SAND2010-2183
Unlimited Release
December 2009
Updated November 30, 2011

DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis

Version 5.2 User's Manual

Brian M. Adams, Keith R. Dalbey, Michael S. Eldred, Laura P. Swiler
Optimization and Uncertainty Quantification Department

William J. Bohnhoff
Radiation Transport Department

John P. Eddy
System Readiness and Sustainment Technologies Department

Dena M. Vigil
Multiphysics Simulation Technologies Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Patricia D. Hough, Sophia Lefantzi
Quantitative Modeling and Analysis Department

Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551

Abstract

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible and extensible interface between simulation codes and iterative analysis methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods; uncertainty quantification with sampling, reliability, and stochastic expansion methods; parameter estimation with nonlinear least squares methods; and sensitivity/variance analysis with design of experiments and parameter study methods. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for design and performance analysis of computational models on high performance computers.

This report serves as a user's manual for the DAKOTA software and provides capability overviews and procedures for software execution, as well as a variety of example studies.

Contents

Preface **17**

1 Introduction **19**

1.1 Motivation for DAKOTA Development . . . 19

1.2 Capabilities of DAKOTA . . . 19

1.3 How Does DAKOTA Work? . . . 20

1.4 Background and Mathematical Formulations . . . 21

1.4.1 Optimization . . . 21

1.4.2 Nonlinear Least Squares for Parameter Estimation . . . 23

1.4.3 Sensitivity Analysis and Parameter Studies . . . 23

1.4.4 Design of Experiments . . . 24

1.4.5 Uncertainty Quantification . . . 25

1.5 Using this Manual . . . 25

2 DAKOTA Tutorial **27**

2.1 Getting Started . . . 27

2.1.1 Quickstart . . . 27

2.1.2 Running DAKOTA . . . 28

2.2 Rosenbrock and Textbook Test Problems . . . 30

2.3 DAKOTA Input File Format . . . 31

2.4 Example Problems . . . 34

2.4.1 Parameter Studies . . . 34

2.4.1.1 Two-Dimensional Grid Parameter Study . . . 34

2.4.1.2 Vector Parameter Study . . . 35

2.4.2 Optimization . . . 35

2.4.2.1 Gradient-based Unconstrained Optimization . . . 36

2.4.2.2 Gradient-based Constrained Optimization . . . 40
2.4.2.3 Nonlinear Least Squares Methods for Optimization . . . 41
2.4.2.4 Nongradient-based Optimization via Pattern Search . . . 42
2.4.2.5 Nongradient-based Optimization via Evolutionary Algorithm . . . 45
2.4.2.6 Multiobjective Optimization . . . 48
2.4.3 Uncertainty Quantification . . . 51
2.4.3.1 Monte Carlo Sampling . . . 51
2.4.3.2 Reliability Methods - via the Mean Value Method . . . 52
2.4.3.3 Polynomial Chaos . . . 55
2.4.3.4 Interval Analysis . . . 56
2.4.4 User Supplied Simulation Code Examples . . . 58
2.4.4.1 Optimization with a User-Supplied Simulation Code - Case 1 . . . 58
2.4.4.2 Optimization with a User-Supplied Simulation Code - Case 2 . . . 62
2.5 Where to Go from Here . . . 63

3 DAKOTA Capability Overview 65
3.1 Purpose . . . 65
3.2 Parameter Study Methods . . . 65
3.3 Design of Experiments . . . 65
3.4 Uncertainty Quantification . . . 66
3.5 Optimization . . . 68
3.6 Additional Optimization Capabilities . . . 70
3.7 Nonlinear Least Squares for Parameter Estimation . . . 70
3.8 Surrogate-Based Minimization . . . 71
3.9 Optimization Strategies . . . 72
3.10 Surrogate Models . . . 72
3.11 Nested Models . . . 74
3.12 Parallel Computing . . . 75
3.13 Summary . . . 75

4 Parameter Study Capabilities 77
4.1 Overview . . . 77
4.1.1 Initial Values . . . 78
4.1.2 Bounds . . . 78
4.2 Vector Parameter Study . . . 78

4.3 List Parameter Study . . . 80
4.4 Centered Parameter Study . . . 80
4.5 Multidimensional Parameter Study . . . 82

5 Design of Experiments Capabilities 85
5.1 Overview . . . 85
5.2 Design of Computer Experiments . . . 85
5.3 DDACE Background . . . 87
5.3.1 Central Composite Design . . . 87
5.3.2 Box-Behnken Design . . . 88
5.3.3 Orthogonal Array Designs . . . 88
5.3.4 Grid Design . . . 89
5.3.5 Monte Carlo Design . . . 90
5.3.6 LHS Design . . . 90
5.3.7 OA-LHS Design . . . 90
5.4 FSUDace Background . . . 90
5.5 PSUADE MOAT Background . . . 91
5.6 Sensitivity Analysis . . . 92

6 Uncertainty Quantification Capabilities 95
6.1 Overview . . . 95
6.2 Sampling Methods . . . 95
6.2.1 Uncertainty Quantification Example using Sampling Methods . . . 97
6.2.2 Incremental Sampling . . . 102
6.3 Reliability Methods . . . 103
6.3.1 Local Reliability Methods . . . 103
6.3.1.1 Method mapping . . . 103
6.3.2 Global Reliability Methods . . . 105
6.3.3 Uncertainty Quantification Example using Reliability Analysis . . . 106
6.4 Stochastic Expansion Methods . . . 107
6.4.1 Uncertainty Quantification Example using Stochastic Collocation . . . 112
6.5 Epistemic Nondeterministic Methods . . . 112
6.5.1 Interval Methods for Epistemic Analysis . . . 114
6.5.2 Dempster-Shafer Theory of Evidence . . . 116
6.6 Future Nondeterministic Methods . . . 119

7 Optimization Capabilities **121**
7.1 Overview . . . 121
7.2 Optimization Software Packages . . . 122
7.2.1 HOPSPACK Library . . . 122
7.2.2 SCOLIB Library . . . 122
7.2.3 Constrained Minimization (CONMIN) Library . . . 123
7.2.4 Design Optimization Tools (DOT) Library . . . 124
7.2.5 dl_solver — Solvers via Shared Libraries . . . 124
7.2.6 JEGA . . . 125
7.2.7 NCSU DIRECT . . . 125
7.2.8 NLPQL Library . . . 125
7.2.9 NPSOL Library . . . 126
7.2.10 OPT++ Library . . . 126
7.3 Additional Optimization Capabilities . . . 127
7.3.1 Multiobjective Optimization . . . 127
7.3.2 Optimization with User-specified or Automatic Scaling . . . 128

8 Nonlinear Least Squares Capabilities **135**
8.1 Overview . . . 135
8.2 Solution Techniques . . . 136
8.2.1 Gauss-Newton . . . 136
8.2.2 NLSSOL . . . 136
8.2.3 NL2SOL . . . 137
8.2.4 Additional Features and Future plans . . . 137
8.3 Examples . . . 137

9 Surrogate-Based Minimization **139**
9.1 Overview . . . 139
9.2 Surrogate-Based Local Minimization . . . 139
9.2.1 SBO with Data Fits . . . 139
9.2.2 SBO with Multifidelity Models . . . 142
9.2.3 SBO with Reduced Order Models . . . 142
9.3 Surrogate-Based Global Minimization . . . 143
9.4 Efficient Global Minimization . . . 144

10 Models **147**
10.1 Overview . . . 147
10.2 Single Models . . . 148
10.3 Recast Models . . . 148
10.4 Surrogate Models . . . 148
10.4.1 Data Fit Surrogate Models . . . 149
10.4.1.1 Procedures for Surface Fitting . . . 150
10.4.1.2 Taylor Series . . . 150
10.4.1.3 Two Point Adaptive Nonlinearity Approximation . . . 150
10.4.1.4 Linear, Quadratic, and Cubic Polynomial Models . . . 151
10.4.1.5 Kriging/Gaussian-Process Spatial Interpolation Models . . . 152
10.4.1.6 Artificial Neural Network (ANN) Models . . . 155
10.4.1.7 Multivariate Adaptive Regression Spline (MARS) Models . . . 155
10.4.1.8 Radial Basis Functions . . . 156
10.4.1.9 Moving Least Squares . . . 156
10.4.2 Multifidelity Surrogate Models . . . 157
10.4.3 Reduced Order Models . . . 157
10.5 Nested Models . . . 157

11 Variables **159**
11.1 Overview . . . 159
11.2 Design Variables . . . 159
11.2.1 Continuous Design Variables . . . 159
11.2.2 Discrete Design Variables . . . 160
11.2.2.1 Discrete Design Integer Variables . . . 160
11.2.2.2 Discrete Design Real Variables . . . 160
11.3 Uncertain Variables . . . 161
11.3.1 Aleatory Uncertain Variables . . . 161
11.3.1.1 Continuous Aleatory Uncertain Variables . . . 161
11.3.1.2 Discrete Aleatory Uncertain Variables . . . 162
11.3.2 Epistemic Uncertain Variables . . . 162
11.3.2.1 Continuous Epistemic Uncertain Variables . . . 163
11.4 State Variables . . . 163
11.5 Mixed Variables . . . 163

11.6 DAKOTA Parameters File Data Format . . . 164
11.6.1 Parameters file format (standard) . . . 164
11.6.2 Parameters file format (APREPRO) . . . 165
11.7 The Active Set Vector . . . 166
11.7.1 Active set vector control . . . 167

12 Interfaces **169**
12.1 Overview . . . 169
12.2 Algebraic Mappings . . . 169
12.3 Simulation Interfaces . . . 172
12.3.1 The Direct Function Simulation Interface . . . 172
12.3.2 The System Call Simulation Interface . . . 173
12.3.3 The Fork Simulation Interface . . . 173
12.3.4 Syntax for Filter and Driver Strings . . . 174
12.3.5 Fork or System Call: Which to Use? . . . 174
12.4 Simulation Interface Components . . . 175
12.4.1 Single analysis driver without filters . . . 175
12.4.2 Single analysis driver with filters . . . 177
12.4.3 Multiple analysis drivers without filters . . . 178
12.4.4 Multiple analysis drivers with filters . . . 178
12.5 Simulation File Management . . . 179
12.5.1 File Saving . . . 180
12.5.2 File Tagging for Evaluations . . . 180
12.5.3 Temporary Files . . . 181
12.5.4 File Tagging for Analysis Drivers . . . 181
12.5.5 Work Directories . . . 182
12.6 Parameter to Response Mappings . . . 183

13 Responses **189**
13.1 Overview . . . 189
13.1.1 Response function types . . . 189
13.1.2 Gradient availability . . . 190
13.1.3 Hessian availability . . . 190
13.2 DAKOTA Results File Data Format . . . 190
13.3 Active Variables for Derivatives . . . 191

14 Inputs to DAKOTA **193**
14.1 Overview of Inputs . . . 193
14.1.1 Tabular Data Formats . . . 193
14.2 Data Imports . . . 194
14.2.1 AMPL algebraic mappings: stub.nl, stub.row, and stub.col . . . 194
14.2.2 Genetic algorithm population import . . . 194
14.2.3 Calibration (least squares) data import . . . 194
14.2.4 PCE coefficient import . . . 195
14.2.5 Surrogate construction from data files . . . 195
14.2.6 Variables/responses import to post-run . . . 195

15 Output from DAKOTA **197**
15.1 Overview of Output Formats . . . 197
15.2 Standard Output . . . 197
15.3 Tabular Output Data . . . 203
15.4 Graphics Output . . . 204
15.5 Error Messages Output . . . 205
15.6 Variables Output from Pre-run . . . 205

16 Advanced Strategies **207**
16.1 Overview . . . 207
16.2 Hybrid Minimization . . . 207
16.3 Multistart Local Minimization . . . 208
16.4 Pareto Optimization . . . 210

17 Advanced Model Recursions **213**
17.1 Mixed Aleatory-Epistemic UQ . . . 213
17.1.1 Interval-valued probability (IVP) . . . 213
17.1.2 Second-order probability (SOP) . . . 216
17.1.3 Dempster-Shafer Theory of Evidence . . . 217
17.2 Optimization Under Uncertainty (OUU) . . . 217
17.2.1 Nested OUU . . . 217
17.2.2 Surrogate-Based OUU (SBOUU) . . . 218
17.2.3 Trust-Region Surrogate-Based OUU (TR-SBOUU) . . . 220
17.2.4 RBDO . . . 220

17.2.5 Stochastic Expansion-Based Design Optimization . . . 221
17.2.6 Epistemic OUU . . . 222
17.3 Surrogate-Based Uncertainty Quantification . . . 222

18 Advanced Simulation Code Interfaces 225
18.1 Building an Interface to a Engineering Simulation Code . . . 225
18.1.1 Generic Script Interface Files . . . 225
18.1.2 Adapting These Scripts to Another Simulation . . . 232
18.1.3 Additional Examples . . . 233
18.2 Developing a Direct Simulation Interface . . . 233
18.2.1 Extension . . . 233
18.2.2 Derivation . . . 234
18.2.3 Sandwich . . . 234
18.3 Existing Direct Interfaces to External Simulators . . . 234
18.3.1 Matlab . . . 235
18.3.1.1 DAKOTA/Matlab input file specification . . . 235
18.3.1.2 Matlab .m file specification . . . 236
18.3.2 Python . . . 237

19 Parallel Computing 241
19.1 Overview . . . 241
19.1.1 Categorization of parallelism . . . 241
19.1.2 Parallel DAKOTA algorithms . . . 243
19.1.2.1 Parallel iterators . . . 243
19.1.2.2 Parallel strategies . . . 244
19.1.2.3 Parallel models . . . 244
19.2 Single-level parallelism . . . 245
19.2.1 Asynchronous Local Parallelism . . . 246
19.2.1.1 Direct function synchronization . . . 246
19.2.1.2 System call synchronization . . . 247
19.2.1.3 Fork synchronization . . . 248
19.2.1.4 Asynchronous Local Example . . . 248
19.2.1.5 Local evaluation scheduling options . . . 250
19.2.2 Message Passing Parallelism . . . 250
19.2.2.1 Partitioning . . . 251

19.2.2.2 Scheduling . . . 251
19.2.2.3 Message Passing Example . . . 252
19.2.3 Hybrid Parallelism . . . 253
19.2.3.1 Hybrid Example . . . 254
19.3 Multilevel parallelism . . . 256
19.3.1 Asynchronous Local Parallelism . . . 257
19.3.2 Message Passing Parallelism . . . 257
19.3.2.1 Partitioning of levels . . . 257
19.3.2.2 Scheduling within levels . . . 258
19.3.3 Hybrid Parallelism . . . 258
19.4 Capability Summary . . . 259
19.5 Running a Parallel DAKOTA Job . . . 259
19.5.1 Single-processor execution . . . 260
19.5.2 Multiprocessor execution . . . 260
19.6 Specifying Parallelism . . . 261
19.6.1 The interface specification . . . 262
19.6.2 The strategy specification . . . 262
19.6.3 Single-processor DAKOTA specification . . . 262
19.6.3.1 Example 1 . . . 263
19.6.3.2 Example 2 . . . 263
19.6.4 Multiprocessor DAKOTA specification . . . 264
19.6.4.1 Example 3 . . . 264
19.6.4.2 Example 4 . . . 266
19.6.4.3 Example 5 . . . 268
19.7 Application Parallelism Use Cases . . . 269
19.7.1 Case 1: Multiple serial analysis jobs . . . 270
19.7.2 Case 2: One simultaneous parallel analysis job . . . 271
19.7.3 Case 3: Multiple simultaneous parallel analysis jobs . . . 271
19.7.3.1 Mpiexec server mode . . . 271
19.7.3.2 Relative node scheduling . . . 272
19.7.3.3 Machinefile management . . . 272
19.7.4 Case 4: Parallel analysis jobs submitted to a queue . . . 272

20 DAKOTA Usage Guidelines **273**

20.1 Problem Exploration . . . 273
20.2 Optimization Method Selection . . . 273
20.3 UQ Method Selection . . . 275
20.4 Parameter Study/DOE/DACE/Sampling Method Selection . . . 278
20.5 Surrogate Model Selection . . . 278

21 Restart Capabilities and Utilities 281
21.1 Restart Management . . . 281
21.2 The DAKOTA Restart Utility . . . 282
21.2.1 Print . . . 283
21.2.2 To/From Neutral File Format . . . 284
21.2.3 To Tabular Format . . . 284
21.2.4 Concatenation of Multiple Restart Files . . . 285
21.2.5 Removal of Corrupted Data . . . 285

22 Simulation Failure Capturing 287
22.1 Failure detection . . . 287
22.2 Failure communication . . . 288
22.3 Failure mitigation . . . 288
22.3.1 Abort (default) . . . 288
22.3.2 Retry . . . 288
22.3.3 Recover . . . 288
22.3.4 Continuation . . . 289
22.4 Special values . . . 289

23 Additional Examples 291
23.1 Textbook Example . . . 291
23.1.1 Methods . . . 292
23.1.2 Optimization Results . . . 292
23.1.3 Least Squares Results . . . 293
23.2 Rosenbrock Example . . . 293
23.2.1 Methods . . . 294
23.2.2 Results . . . 295
23.3 Herbie, Smooth Herbie, and Shubert Example . . . 297
23.4 Cylinder Head Example . . . 299

23.4.1 Methods . . . 301
23.4.2 Optimization Results . . . 301
23.5 Container Example . . . 302
23.6 Log Ratio Example . . . 305
23.7 Steel Section Example . . . 305
23.8 Portal Frame Example . . . 306
23.9 Short Column Example . . . 306
23.9.1 Uncertainty Quantification . . . 307
23.9.2 Reliability-Based Design Optimization . . . 307
23.10 Cantilever Example . . . 308
23.10.1 Deterministic Optimization Results . . . 308
23.10.2 Stochastic Optimization Results . . . 310
23.11 Steel Column Example . . . 310
23.12 Multiobjective Examples . . . 311
23.12.1 Multiobjective Test Problem 1 . . . 311
23.12.2 Multiobjective Test Problem 2 . . . 311
23.12.3 Multiobjective Test Problem 3 . . . 314
23.13 Morris example . . . 314

Preface

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) project started in 1994 as an internal research and development activity at Sandia National Laboratories in Albuquerque, New Mexico. The original goal was to provide a common set of optimization tools for a group of engineers solving structural analysis and design problems. Prior to the DAKOTA project, there was no focused effort to archive optimization methods for reuse on other projects. Thus, engineers found themselves repeatedly building new custom interfaces between the engineering analysis software and optimization software. This was especially burdensome when using parallel computing, as each project developed a unique master program to coordinate concurrent simulations on a network of workstations or a parallel computer. The initial DAKOTA toolkit provided the engineering and analysis community at Sandia access to a variety of optimization algorithms, hiding the complexity of the optimization software interfaces from the users. Engineers could readily switch between optimization software packages by simply changing a few lines in a DAKOTA input file. In addition to structural analysis, DAKOTA has been applied to computational fluid dynamics, nonlinear dynamics, shock physics, heat transfer, electrical circuits, and many other science and engineering models.

DAKOTA has grown significantly beyond an optimization toolkit. In addition to its state-of-the-art optimization methods, DAKOTA includes methods for global sensitivity and variance analysis, parameter estimation, uncertainty quantification, and verification, as well as meta-level strategies for surrogate-based optimization, hybrid optimization, and optimization under uncertainty. Available to all these algorithms is parallel computation support; ranging from desktop multiprocessor computers to massively parallel computers typically found at national laboratories and supercomputer centers.

As of Version 5.0, DAKOTA is publicly released as open source under a GNU Lesser General Public License and is available for free download world-wide. See http://www.gnu.org/licenses/lgpl.html for more information on the LGPL software use agreement. DAKOTA Versions 3.0 through 4.2+ were licensed under the GNU General Public License. DAKOTA public release facilitates research and software collaborations among DAKOTA developers at Sandia National Laboratories and other institutions, including academic, government, and corporate entities. See the DAKOTA FAQ at http://dakota.sandia.gov/faq.html for more information on the public release rationale and ways to contribute.

The DAKOTA leadership team consists of Brian Adams (project lead), Mike Eldred (research lead), Bill Bohnhoff (support manager), and Jim Stewart (business manager). DAKOTA development team members include Keith Dalbey, John Eddy, Patty Hough, Sophia Lefantzi, Laura Swiler, and Dena Vigil. Additional historical contributors to DAKOTA and its third-party libraries are acknowledged on the DAKOTA web page.

Contact Information:
Brian M. Adams, DAKOTA Project Lead
Sandia National Laboratories
P.O. Box 5800, Mail Stop 1318
Albuquerque, NM 87185-1318
Web (including support information): http://dakota.sandia.gov

Chapter 1

Introduction

1.1 Motivation for DAKOTA Development

Computational models are commonly used in engineering design and scientific discovery activities for simulating complex physical systems in disciplines such as fluid mechanics, structural dynamics, heat transfer, nonlinear structural mechanics, shock physics, and many others. These simulators can be an enormous aid to engineers who want to develop an understanding and/or predictive capability for complex behaviors typically observed in the corresponding physical systems. Simulators often serve as virtual prototypes, where a set of predefined system parameters, such as size or location dimensions and material properties, are adjusted to improve the performance of a system, as defined by one or more system performance objectives. Such optimization or tuning of the virtual prototype requires executing the simulator, evaluating performance objective(s), and adjusting the system parameters in an iterative, automated, and directed way. System performance objectives can be formulated, for example, to minimize weight, cost, or defects; to limit a critical temperature, stress, or vibration response; or to maximize performance, reliability, throughput, agility, or design robustness. In addition, one would often like to design computer experiments, run parameter studies, or perform uncertainty quantification. These approaches reveal how system performance changes as a design or uncertain input variable changes. Sampling strategies are often used in uncertainty quantification to calculate a distribution on system performance measures, and to understand which uncertain inputs contribute most to the variance of the outputs.

A primary goal for DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) development is to provide engineers and other disciplinary scientists with a systematic and rapid means to obtain improved or optimal designs or understand sensitivity or uncertainty using simulation-based models. These capabilities generally lead to improved designs and system performance in earlier design stages, alleviating dependence on physical prototypes and testing, shortening design cycles, and reducing product development costs. In addition to providing this practical environment for answering system performance questions, the DAKOTA toolkit provides an extensible platform for the research and rapid prototyping of customized methods and strategies [23].

1.2 Capabilities of DAKOTA

The DAKOTA toolkit provides a flexible, extensible interface between your simulation code and a variety of iterative methods and strategies. While DAKOTA was originally conceived as an easy-to-use interface between simulation codes and optimization algorithms, recent versions have been expanded to interface with other types

Figure 1.1: The loosely-coupled or "black-box" interface between DAKOTA and a user-supplied simulation code.

of iterative analysis methods such as uncertainty quantification with nondeterministic propagation methods, parameter estimation with nonlinear least squares solution methods, and sensitivity/variance analysis with general-purpose design of experiments and parameter study capabilities. These capabilities may be used on their own or as building blocks within more sophisticated strategies such as hybrid optimization, surrogate-based optimization, or optimization under uncertainty.

Thus, one of the primary advantages that DAKOTA has to offer is that access to a broad range of iterative capabilities can be obtained through a single, relatively simple interface between DAKOTA and your simulator. Should you want to try a different type of iterative method or strategy with your simulator, it is only necessary to change a few commands in the DAKOTA input and start a new analysis. The need to learn a completely different style of command syntax and the need to construct a new interface each time you want to use a new algorithm are eliminated.

1.3 How Does DAKOTA Work?

Figure 1.1 depicts a typical loosely-coupled, or "black-box," relationship between DAKOTA and the simulation code(s). Such loose coupling is the simplest and most common interfacing approach DAKOTA users employ. Data is exchanged between DAKOTA and the simulation code by reading and writing short data files, thus using DAKOTA does not require the source code of the simulation software. DAKOTA is executed using commands that the user supplies in an input file (not shown in Figure 1.1) which specify the type of analysis to be performed (e.g., parameter study, optimization, uncertainty quantification, etc.), along with the file names associated with the user's simulation code. During its operation, DAKOTA automatically executes the user's simulation code by creating a separate process external to DAKOTA.

The solid lines in Figure 1.1 denote file input/output (I/O) operations that are part of DAKOTA or the user's simulation code. The dotted lines indicate the passing/conversion of information that must be implemented by the user. As DAKOTA runs, it writes out a parameters file containing the current variable values. DAKOTA then starts the user's simulation code (or, often, a short driver script wrapping it), and when the simulation completes, reads the response data from a results file. This process is repeated until all of the simulation code runs required by the iterative study are complete.

In some cases it is advantageous to have a close coupling between DAKOTA and the simulation code. This close

coupling is an advanced feature of DAKOTA and is accomplished through either a direct interface or a SAND (simultaneous analysis and design) interface. For the direct interface, the user's simulation code is modified to behave as a function or subroutine under DAKOTA. This interface can be considered to be "semi-intrusive" in that it requires relatively minor modifications to the simulation code. Its major advantage is the elimination of the overhead resulting from file I/O and process creation. It can also be a useful tool for parallel processing, by encapsulating everything within a single executable. A SAND interface approach is "fully intrusive" in that it requires further modifications to the simulation code so that an optimizer has access to the internal residual vector and Jacobian matrices computed by the simulation code. In a SAND approach, both the optimization method and a nonlinear simulation code are converged simultaneously. While this approach can greatly reduce the computational expense of optimization, considerable software development effort must be expended to achieve this intrusive coupling between SAND optimization methods and the simulation code. SAND may be supported in future DAKOTA releases.

1.4 Background and Mathematical Formulations

This section provides a basic introduction to the mathematical formulation of optimization, nonlinear least squares, sensitivity analysis, design of experiments, and uncertainty quantification problems. The primary goal of this section is to introduce terms relating to these topics, and is not intended to be a description of theory or numerical algorithms. There are numerous sources of information on these topics ([6], [46], [61], [62], [86], [117]) and the interested reader is advised to consult one or more of these texts.

1.4.1 Optimization

A general optimization problem is formulated as follows:

$$
\begin{aligned}
\text{minimize:} \quad & f(\mathbf{x}) \\
& \mathbf{x} \in \Re^n \\
\text{subject to:} \quad & \mathbf{g}_L \leq \mathbf{g}(\mathbf{x}) \leq \mathbf{g}_U \\
& \mathbf{h}(\mathbf{x}) = \mathbf{h}_t \\
& \mathbf{a}_L \leq \mathbf{A}_i\mathbf{x} \leq \mathbf{a}_U \\
& \mathbf{A}_e\mathbf{x} = \mathbf{a}_t \\
& \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U
\end{aligned}
\tag{1.1}
$$

where vector and matrix terms are marked in bold typeface. In this formulation, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is an n-dimensional vector of real-valued *design variables* or *design parameters*. The n-dimensional vectors, $\mathbf{x}_L$ and $\mathbf{x}_U$, are the lower and upper bounds, respectively, on the design parameters. These bounds define the allowable values for the elements of $\mathbf{x}$, and the set of all allowable values is termed the *design space* or the *parameter space*. A *design point* or a *sample point* is a particular set of values within the parameter space.

The optimization goal is to minimize the *objective function*, $f(\mathbf{x})$, while satisfying the constraints. Constraints can be categorized as either linear or nonlinear and as either inequality or equality. The *nonlinear inequality constraints*, $\mathbf{g}(\mathbf{x})$, are "2-sided," in that they have both lower and upper bounds, $\mathbf{g}_L$ and $\mathbf{g}_U$, respectively. The *nonlinear equality constraints*, $\mathbf{h}(\mathbf{x})$, have target values specified by $\mathbf{h}_t$. The linear inequality constraints create a linear system $\mathbf{A}_i\mathbf{x}$, where $\mathbf{A}_i$ is the coefficient matrix for the linear system. These constraints are also 2-sided as they have lower and upper bounds, $\mathbf{a}_L$ and $\mathbf{a}_U$, respectively. The linear equality constraints create a linear

system $\mathbf{A}_e\mathbf{x}$, where $\mathbf{A}_e$ is the coefficient matrix for the linear system and $\mathbf{a}_t$ are the target values. The constraints partition the parameter space into feasible and infeasible regions. A design point is said to be *feasible* if and only if it satisfies all of the constraints. Correspondingly, a design point is said to be *infeasible* if it violates one or more of the constraints.

Many different methods exist to solve the optimization problem given by Equation 1.1, all of which iterate on $\mathbf{x}$ in some manner. That is, an initial value for each parameter in $\mathbf{x}$ is chosen, the *response quantities*, $f(\mathbf{x})$, $\mathbf{g}(\mathbf{x})$, $\mathbf{h}(\mathbf{x})$, are computed, often by running a simulation, and some algorithm is applied to generate a new $\mathbf{x}$ that will either reduce the objective function, reduce the amount of infeasibility, or both. To facilitate a general presentation of these methods, three criteria will be used in the following discussion to differentiate them: optimization problem type, search goal, and search method.

The **optimization problem type** can be characterized both by the types of constraints present in the problem and by the linearity or nonlinearity of the objective and constraint functions. For constraint categorization, a hierarchy of complexity exists for optimization algorithms, ranging from simple bound constraints, through linear constraints, to full nonlinear constraints. By the nature of this increasing complexity, optimization problem categorizations are inclusive of all constraint types up to a particular level of complexity. That is, an *unconstrained problem* has no constraints, a *bound-constrained problem* has only lower and upper bounds on the design parameters, a *linearly-constrained problem* has both linear and bound constraints, and a *nonlinearly-constrained problem* may contain the full range of nonlinear, linear, and bound constraints. If all of the linear and nonlinear constraints are equality constraints, then this is referred to as an *equality-constrained problem*, and if all of the linear and nonlinear constraints are inequality constraints, then this is referred to as an *inequality-constrained problem*. Further categorizations can be made based on the linearity of the objective and constraint functions. A problem where the objective function and all constraints are linear is called a *linear programming (LP) problem*. These types of problems commonly arise in scheduling, logistics, and resource allocation applications. Likewise, a problem where at least some of the objective and constraint functions are nonlinear is called a *nonlinear programming (NLP) problem*. These NLP problems predominate in engineering applications and are the primary focus of DAKOTA.

The **search goal** refers to the ultimate objective of the optimization algorithm, i.e., either global or local optimization. In *global optimization*, the goal is to find the design point that gives the lowest feasible objective function value over the entire parameter space. In contrast, in *local optimization*, the goal is to find a design point that is lowest relative to a "nearby" region of the parameter space. In almost all cases, global optimization will be more computationally expensive than local optimization. Thus, the user must choose an optimization algorithm with an appropriate search scope that best fits the problem goals and the computational budget.

The **search method** refers to the approach taken in the optimization algorithm to locate a new design point that has a lower objective function or is more feasible than the current design point. The search method can be classified as either *gradient-based* or *nongradient-based*. In a gradient-based algorithm, gradients of the response functions are computed to find the direction of improvement. Gradient-based optimization is the search method that underlies many efficient local optimization methods. However, a drawback to this approach is that gradients can be computationally expensive, inaccurate, or even nonexistent. In such situations, nongradient-based search methods may be useful. There are numerous approaches to nongradient-based optimization. Some of the more well known of these include pattern search methods (nongradient-based local techniques) and genetic algorithms (nongradient-based global techniques). Because of the computational cost of running simulation models, surrogate-based optimization (SBO) methods are often used to reduce the number of actual simulation runs. In SBO, a surrogate or approximate model is constructed based on a limited number of simulation runs. The optimization is then performed on the surrogate model. DAKOTA has an extensive framework for managing a variety of local, multipoint, global, and hierarchical surrogates for use in optimization.

The overview of optimization methods presented above underscores that there is no single optimization method or algorithm that works best for all types of optimization problems. Chapter 20 provides some guidelines on

choosing which DAKOTA optimization algorithm is best matched to your specific optimization problem.

1.4.2 Nonlinear Least Squares for Parameter Estimation

Specialized least squares solution algorithms can exploit the structure of a sum of the squares objective function for problems of the form:

$$\begin{aligned}
\text{minimize:} \quad & f(\mathbf{x}) = \sum_{i=1}^{n} [T_i(\mathbf{x})]^2 \\
& \mathbf{x} \in \Re^n \\
\text{subject to:} \quad & \mathbf{g}_L \leq \mathbf{g}(\mathbf{x}) \leq \mathbf{g}_U \\
& \mathbf{h}(\mathbf{x}) = \mathbf{h}_t \\
& \mathbf{a}_L \leq \mathbf{A}_i \mathbf{x} \leq \mathbf{a}_U \\
& \mathbf{A}_e \mathbf{x} = \mathbf{a}_t \\
& \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U
\end{aligned} \tag{1.2}$$

where $f(\mathbf{x})$ is the objective function to be minimized and $T_i(\mathbf{x})$ is the $\mathrm{i^{th}}$ least squares term. The bound, linear, and nonlinear constraints are the same as described previously for (1.1). Specialized least squares algorithms are generally based on the Gauss-Newton approximation. When differentiating $f(\mathbf{x})$ twice, terms of $T_i(\mathbf{x})T_i''(\mathbf{x})$ and $[T_i'(\mathbf{x})]^2$ result. By assuming that the former term tends toward zero near the solution since $T_i(\mathbf{x})$ tends toward zero, then the Hessian matrix of second derivatives of $f(\mathbf{x})$ can be approximated using only first derivatives of $T_i(\mathbf{x})$. As a result, Gauss-Newton algorithms exhibit quadratic convergence rates near the solution for those cases when the Hessian approximation is accurate, i.e. the residuals tend towards zero at the solution. Thus, by exploiting the structure of the problem, the second order convergence characteristics of a full Newton algorithm can be obtained using only first order information from the least squares terms.

A common example for $T_i(\mathbf{x})$ might be the difference between experimental data and model predictions for a response quantity at a particular location and/or time step, i.e.:

$$T_i(\mathbf{x}) = R_i(\mathbf{x}) - \overline{R_i} \tag{1.3}$$

where $R_i(\mathbf{x})$ is the response quantity predicted by the model and $\overline{R_i}$ is the corresponding experimental data. In this case, $\mathbf{x}$ would have the meaning of model parameters which are not precisely known and are being calibrated to match available data. This class of problem is known by the terms parameter estimation, system identification, model calibration, test/analysis reconciliation, etc.

1.4.3 Sensitivity Analysis and Parameter Studies

In many engineering design applications, sensitivity analysis techniques and parameter study methods are useful in identifying which of the design parameters have the most influence on the response quantities. This information is helpful prior to an optimization study as it can be used to remove design parameters that do not strongly influence the responses. In addition, these techniques can provide assessments as to the behavior of the response functions (smooth or nonsmooth, unimodal or multimodal) which can be invaluable in algorithm selection for optimization, uncertainty quantification, and related methods. In a post-optimization role, sensitivity information is useful is

determining whether or not the response functions are robust with respect to small changes in the optimum design point.

In some instances, the term sensitivity analysis is used in a local sense to denote the computation of response derivatives at a point. These derivatives are then used in a simple analysis to make design decisions. DAKOTA supports this type of study through numerical finite-differencing or retrieval of analytic gradients computed within the analysis code. The desired gradient data is specified in the responses section of the DAKOTA input file and the collection of this data at a single point is accomplished through a parameter study method with no steps. This approach to sensitivity analysis should be distinguished from the activity of augmenting analysis codes to internally compute derivatives using techniques such as direct or adjoint differentiation, automatic differentiation (e.g., ADIFOR), or complex step modifications. These sensitivity augmentation activities are completely separate from DAKOTA and are outside the scope of this manual. However, once completed, DAKOTA can utilize these analytic gradients to perform optimization, uncertainty quantification, and related studies more reliably and efficiently.

In other instances, the term sensitivity analysis is used in a more global sense to denote the investigation of variability in the response functions. DAKOTA supports this type of study through computation of response data sets (typically function values only, but all data sets are supported) at a series of points in the parameter space. The series of points is defined using either a vector, list, centered, or multidimensional parameter study method. For example, a set of closely-spaced points in a vector parameter study could be used to assess the smoothness of the response functions in order to select a finite difference step size, and a set of more widely-spaced points in a centered or multidimensional parameter study could be used to determine whether the response function variation is likely to be unimodal or multimodal. See Chapter 4 for additional information on these methods. These more global approaches to sensitivity analysis can be used to obtain trend data even in situations when gradients are unavailable or unreliable, and they are conceptually similar to the design of experiments methods and sampling approaches to uncertainty quantification described in the following sections.

1.4.4 Design of Experiments

Classical design of experiments (DoE) methods and the more modern design and analysis of computer experiments (DACE) methods are both techniques which seek to extract as much trend data from a parameter space as possible using a limited number of sample points. Classical DoE techniques arose from technical disciplines that assumed some randomness and nonrepeatability in field experiments (e.g., agricultural yield, experimental chemistry). DoE approaches such as central composite design, Box-Behnken design, and full and fractional factorial design generally put sample points at the extremes of the parameter space, since these designs offer more reliable trend extraction in the presence of nonrepeatability. DACE methods are distinguished from DoE methods in that the nonrepeatability component can be omitted since computer simulations are involved. In these cases, space filling designs such as orthogonal array sampling and Latin hypercube sampling are more commonly employed in order to accurately extract trend information. Quasi-Monte Carlo sampling techniques which are constructed to fill the unit hypercube with good uniformity of coverage can also be used for DACE.

DAKOTA supports both DoE and DACE techniques. In common usage, only parameter bounds are used in selecting the samples within the parameter space. Thus, DoE and DACE can be viewed as special cases of the more general probabilistic sampling for uncertainty quantification (see following section), in which the DoE/DACE parameters are treated as having uniform probability distributions. The DoE/DACE techniques are commonly used for investigation of global response trends, identification of significant parameters (e.g., main effects), and as data generation methods for building response surface approximations.

1.4.5 Uncertainty Quantification

Uncertainty quantification (UQ) is the process of determining the effect of input uncertainties on response metrics of interest (forward propagating uncertainties through the model). These input uncertainties may be characterized as either aleatory uncertainties, which are irreducible variabilities inherent in nature, or epistemic uncertainties, which are reducible uncertainties resulting from a lack of knowledge. Since sufficient data is generally available for aleatory uncertainties, probabilistic methods are commonly used for computing response distribution statistics based on input probability distribution specifications. Conversely, for epistemic uncertainties, data is generally sparse, making the use of probability theory questionable and leading to nonprobabilistic methods based on interval specifications.

UQ is related to sensitivity analysis in that the common goal is to gain an understanding of how variations in the parameters affect the response functions of the engineering design problem. However, for UQ, some or all of the components of the parameter vector, $\mathbf{x}$, are considered to be uncertain as specified by particular probability distributions (e.g., normal, exponential, extreme value). By assigning specific distributional structure to the inputs, distributional structure for the outputs (i.e., response statistics) can be inferred.

Current DAKOTA methods for modeling aleatory uncertainty include sampling methods, local and global reliability methods, and stochastic expansion methods (polynomial chaos expansions and stochastic collocation). Current methods for modeling epistemic uncertainties include Dempster-Shafer theory of evidence and local or global interval estimation. The sampling, reliability, stochastic expansion, Dempster-Shafer, and interval UQ approaches are described in more detail in Chapter 6. Current methods for modeling mixed aleatory/epistemic uncertainties include interval-valued probability, second-order probability, and Dempster-Shafer theory of evidence, as described in Section 17.1.

1.5 Using this Manual

The previous sections in this chapter provide a brief overview of the capabilities in DAKOTA, and introduce some of the common terms that are used in the fields of optimization, parameter estimation, sensitivity analysis, design of experiments, and uncertainty quantification. A DAKOTA user new to these techniques and terms is advised to consult the cited references to obtain more detailed descriptions of methods and algorithms in these disciplines.

Chapter 2 provides information on how to obtain, install, and use DAKOTA. In addition, example problems are presented in this tutorial chapter to demonstrate some of DAKOTA's capabilities for parameter studies, optimization, and UQ. Chapter 3 provides a brief overview of all of the different software packages and capabilities in DAKOTA. Chapters 4 through 9 provide details on the iterative algorithms supported in DAKOTA, Chapters 10 through 13 provide information on model components which are involved in parameter to response mappings, and Chapters 14 and 15 describe the inputs to and outputs from DAKOTA. Several advanced topics are covered next, with Chapter 16 on strategies, Chapter 17 on model recursions, Chapter 18 on interfacing DAKOTA with engineering simulation codes, and Chapter 19 on DAKOTA's parallel computing capabilities. Next, Chapter 20 provides some usage guidelines for selecting from among DAKOTA's different algorithms for solving particular classes of problems. Finally, Chapter 21 through Chapter 23 describe restart utilities, failure capturing facilities, and additional test problems, respectively.

Chapter 2

DAKOTA Tutorial

2.1 Getting Started

DAKOTA operates on most systems running Unix or Linux operating systems as well as on Windows with the help of a MinGW or Cygwin emulation layer. DAKOTA is developed and most extensively tested on Redhat Enterprise Linux with GNU compilers, but additional operating systems / compiler combinations are tested nightly as well. See the DAKOTA website for more information on supported platforms for particular DAKOTA versions.

Department of Energy users: DAKOTA may already be available on your target system. Sandia users should visit `http://dakota.sandia.gov/sandia_only/` for information on supported DAKOTA installations on engineering networks and cluster computers, as well as for Sandia-specific downloads. At other DOE institutions, contact your system administrator about DAKOTA availability. If not available for your target platform, you may still download DAKOTA as described below.

2.1.1 Quickstart

Getting started with DAKOTA typically involves the following steps:

- Download DAKOTA: binary executables and source code are available from `http://dakota.sandia.gov/`.
- After downloading, unpack the files, yielding a `Dakota/` directory. For tar.gz or .tgz files, e.g., `Dakota_5_x.OSversion.tar.gz` either invoke

  ```
  gunzip Dakota_5_x.OSversion.tar.gz
  tar -xvf Dakota_5_x.OSversion.tar
  ```

  or (slightly faster and less demanding of disk space) invoke

  ```
  gzip -dc Dakota_5_x.OSversion.tar.gz | tar xf -
  ```

  A similar process applies to windows distributions which are packaged as .zip files. These can be extracted with the Windows extractor or WinZIP, for example.

- For binary packages, `Dakota/bin` contains executable files. Example input files and interfaces are in `Dakota/examples` (see `Dakota/examples/README`) and `Dakota/test`. To get started, open a terminal shell or Command Prompt and set your PATH to include `Dakota/bin` (and optionally DAKOTA's test or example directories). On Linux:

  ```
  export PATH=/path/to/Dakota/bin:$PATH
  ```

  or on Windows:

  ```
  set PATH=C:\path\to\Dakota\bin:%PATH%
  ```

  then proceed with running as described below in Section 2.1.2.

- For source packages, configure and compile using either GNU configure/make or CMake/make. For example, for configure:

  ```
  cd Dakota/
  configure
  make
  make install  # optional
  ```

  or for CMake

  ```
  cd Dakota/
  mkdir build && cd build/
  cmake ..
  make
  make install  # optional
  ```

  Detailed configuration options are available on the DAKOTA website, including the Developer Portal http://dakota.sandia.gov/developer/.

- Verify that DAKOTA runs; see Section 2.1.2

- Joining mailing lists and get help: the most up to date guidance for downloading, compiling, installing, or running DAKOTA, or seeking help is available on the DAKOTA website (http://dakota.sandia.gov/resources.html). In addition, guidance is also included in files included with DAKOTA, including `Dakota/INSTALL`. Further platform/operating system-specific guidance can be found in `Dakota/examples/platforms`. For getting started on Windows, see the files `INSTALL.cygwin` and `INSTALL.mingw` in `Dakota/examples/platforms`.

2.1.2 Running DAKOTA

The DAKOTA executable file is named `dakota` (`dakota.exe` on Windows). If this command is entered at the command prompt without any arguments, a usage message similar to the following appears:

```
usage: dakota [options and <args>]
        -help (Print this summary)
        -version (Print DAKOTA version number)
        -input <$val> (REQUIRED DAKOTA input file $val)
        -output <$val> (Redirect DAKOTA standard output to file $val)
        -error <$val> (Redirect DAKOTA standard error to file $val)
```

```
-parser <$val> (Parsing technology: nidr[strict][:dumpfile])
-check (Perform input checks)
-pre_run [$val] (Perform pre-run (variables generation) phase)
-run [$val] (Perform run (model evaluation) phase)
-post_run [$val] (Perform post-run (final results) phase)
-read_restart [$val] (Read an existing DAKOTA restart file $val)
-stop_restart <$val> (Stop restart file processing at evaluation $val)
-write_restart [$val] (Write a new DAKOTA restart file $val)
```

Of these available command line inputs, only the "`-input`" option is required, and "`-input`" can be omitted if the input file name is the final item on the command line; all other command-line inputs are optional. The "`-help`" option prints the usage message above. The "`-version`" option prints the version number of the executable. The "`-check`" option invokes a dry-run mode in which the input file is processed and checked for errors, but the study is not performed. The "`-input`" option provides the name of the DAKOTA input file. The "`-output`" and "`-error`" options provide file names for redirection of the DAKOTA standard output (stdout) and standard error (stderr), respectively. The "`-parser`" input is for debugging and will not be further described here. The "`-read_restart`" and "`-write_restart`" command line inputs provide the names of restart databases to read from and write to, respectively. The "`-stop_restart`" command line input limits the number of function evaluations read from the restart database (the default is all the evaluations) for those cases in which some evaluations were erroneous or corrupted. Restart management is an important technique for retaining data from expensive engineering applications. This advanced topic is discussed in detail in Chapter 20. Note that these command line inputs can be abbreviated so long as the abbreviation is unique, so the following are valid, unambiguous specifications: "`-h`", "`-v`", "`-c`", "`-i`", "`-o`", "`-e`", "`-re`", "`-s`", "`-w`", "`-pr`", "`-ru`", and "`-po`" and can be used in place of the longer forms of the command line inputs.

To run DAKOTA with a particular input file, the following syntax can be used:

```
dakota -i dakota.in
```

or more simply

```
dakota dakota.in
```

This will echo the standard output (stdout) and standard error (stderr) messages to the terminal. To redirect stdout and stderr to separate files, the `-o` and `-e` command line options may be used:

```
dakota -i dakota.in -o dakota.out -e dakota.err
```

or

```
dakota -o dakota.out -e dakota.err dakota.in
```

Alternatively, any of a variety of Unix redirection variants can be used. The simplest of these redirects stdout to another file:

```
dakota dakota.in > dakota.out
```

To append to a file rather than overwrite it, "`>>`" is used in place of "`>`". The syntax to redirect stderr as well as stdout to the same file depends on the shell you are using. With csh, simply append "`&`" with no embedded space, i.e., "`>&`" or "`>>&`". With the Bourne shell (sh or bash) use "`>dakota.out 2>&1`" or "`>>dakota.out 2>&1`". With csh, if you have the noclobber environment variable set but wish either to overwrite an existing

output file or to append to a file that does not yet exist, append "`!`" to the redirection operators (with no intervening spaces), i.e., "`>!`", "`>&!`", "`>>!`", or "`>>&!`".

To run the dakota process in the background, append an ampersand symbol (&) to the command with an embedded space, e.g.,

```
dakota dakota.in > dakota.out &
```

Refer to [5] for more information on Unix redirection and background commands.

The "`-pre_run`", "`-run`", and "`-post_run`" switches instruct DAKOTA to run one or more execution phases, excluding others. For example pre-run might generate variable sets, run (core run) invoke the simulation to evaluate variables, producing responses, and post-run accepts variable/response sets and analyzes the results (for example, calculate correlations from a set of samples). Currently only two modes are supported and only for sampling, parameter study, and DACE methods: (1) pre-run only with optional tabular output of variables:

```
dakota -i dakota.in -pre_run [::myvariables.dat]
```

and (2) post-run only with required tabular input of variables/responses:

```
dakota -i dakota.in -post_run myvarsresponses.dat::
```

2.2 Rosenbrock and Textbook Test Problems

Many of the example problems in this chapter use the Rosenbrock function [96] (also described in [46], among other places), which has the form:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{2.1}$$

A three-dimensional plot of this function is shown in Figure 2.1(a), where both x_1 and x_2 range in value from -2 to 2. Figure 2.1(b) shows a contour plot for Rosenbrock's function. An optimization problem using Rosenbrock's function is formulated as follows:

$$\begin{array}{rl} \texttt{minimize} & f(x_1, x_2) \\ & \mathbf{x} \in \Re^2 \\ \texttt{subject to} & -2 \leq x_1 \leq 2 \\ & -2 \leq x_2 \leq 2 \end{array} \tag{2.2}$$

Note that there are no linear or nonlinear constraints in this formulation, so this is a bound constrained optimization problem. The unique solution to this problem lies at the point $(x_1, x_2) = (1, 1)$, where the function value is zero.

The two-variable version of the "textbook" example problem provides a nonlinearly constrained optimization test

(a) (b)

Figure 2.1: Rosenbrock's function: (a) 3-D plot and (b) contours with x_1 on the bottom axis.

case. It is formulated as:

$$
\begin{aligned}
\texttt{minimize} \quad & f = (x_1 - 1)^4 + (x_2 - 1)^4 \\
\texttt{subject to} \quad & g_1 = x_1^2 - \frac{x_2}{2} \leq 0 \\
& g_2 = x_2^2 - \frac{x_1}{2} \leq 0 \\
& 0.5 \leq x_1 \leq 5.8 \\
& -2.9 \leq x_2 \leq 2.9
\end{aligned}
\tag{2.3}
$$

Contours of this example problem are illustrated in Figure 2.2(a), with a close-up view of the feasible region given in Figure 2.2(b).

For the textbook example problem, the unconstrained minimum occurs at $(x_1, x_2) = (1, 1)$. However, the inclusion of the constraints moves the minimum to $(x_1, x_2) = (0.5, 0.5)$.

Several other example problems are available. See Chapter 23 for a description of these example problems as well as further discussion of the Rosenbrock and textbook example problems.

2.3 DAKOTA Input File Format

All of the DAKOTA input files for the simple example problems presented here are included in the distribution tar files within directory `Dakota/examples/tutorial`. A simple DAKOTA input file (that is named `dakota_rosenbrock_2d.in`) for a two-dimensional parameter study on Rosenbrock's function is shown in Figure 2.3. This input file will be used to describe the basic format and syntax used in all DAKOTA input files.

There are six specification blocks that may appear in DAKOTA input files. These are identified in the input file using the following keywords: variables, interface, responses, model, method, and strategy. These keyword blocks can appear in any order in a DAKOTA input file. At least one *variables*, *interface*, *responses*, and *method* specification must appear, and no more than one *strategy* specification should appear. In Figure 2.3, one of each

Figure 2.2: Contours of the textbook problem (a) on the $[-3, 4] \times [-3, 4]$ domain and (b) zoomed into an area containing the constrained optimum point $(x_1, x_2) = (0.5, 0.5)$. The feasible region lies at the intersection of the two constraints g_1 (solid) and g_2 (dashed).

of the keyword blocks is used. Additional syntax features include use of the # symbol to indicate a comment, use of single or double quotes for string inputs (e.g., `'x1'`), the use of commas and/or white space for separation of specifications, and the optional use of "=" symbols to indicate supplied data. See the DAKOTA Reference Manual [3] for additional details on this input file syntax.

The first section of the input file shown in Figure 2.3 is the *strategy* section. This keyword section is used to specify some of DAKOTA's advanced meta-procedures such as hybrid optimization, multi-start optimization, and Pareto optimization. See Chapter 16 for more information on these meta-procedures. The *strategy* section also contains the settings for DAKOTA's graphical output (via the `graphics` flag) and the tabular data output (via the `tabular_graphics_data` keyword).

The *method* section of the input file specifies the iterative technique that DAKOTA will employ, such as a parameter study, optimization method, data sampling technique, etc. The keyword `multidim_parameter_study` in Figure 2.3 calls for a multidimensional parameter study, while the keyword `partitions` specifies the number of intervals per variable. In this case, there will be eight intervals (nine data points) evaluated between the lower and upper bounds of both variables (bounds provided subsequently in the *variables* section), for a total of 81 response function evaluations.

The *model* section of the input file specifies the model that DAKOTA will use. A model provides the logical unit for determining how a set of variables is mapped into a set of responses in support of an iterative method. The model allows one to specify a single interface, or to manage more sophisticated mappings involving surrogates or nested iteration. For example, one might want to use an approximate model for optimization or uncertainty quantification, due to the lower computational cost. The `model` keyword allows one to specify if the iterator will be operating on a data fit surrogate (such as a polynomial regression, neural net, etc.), a hierarchical surrogate (which uses the corrected results of a lower fidelity simulation model as an approximation to a higher fidelity simulation), or a nested model. See Chapter 10 for additional model specification details. If these advanced facilities for surrogate modeling or nested iteration are not required, then it is not necessary to specify the `model` keyword at all, since the default behavior is the use of a "single" model constructed with the last set of responses, variables, and interface specified. In Figure 2.3, the keyword `single` explicitly specifies the use of a single

```
## DAKOTA INPUT FILE - dakota_rosenbrock_2d.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        multidim_parameter_study
          partitions = 8 8

model,
        single

variables,
        continuous_design = 2
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

Figure 2.3: Rosenbrock 2-D parameter study example: the DAKOTA input file.

model in the parameter study, even though this is the default.

The *variables* section of the input file specifies the characteristics of the parameters that will be used in the problem formulation. The variables can be continuous or discrete, and can be classified as design variables, uncertain variables, or state variables. See Chapter 11 for more information on the types of variables supported by DAKOTA. The *variables* section shown in Figure 2.3 specifies that there are two continuous design variables. The sub-specifications for continuous design variables provide the descriptors "x1" and "x2" as well as lower and upper bounds for these variables. The information about the variables is organized in column format for readability. So, both variables x_1 and x_2 have a lower bound of -2.0 and an upper bound of 2.0.

The *interface* section of the input file specifies what approach will be used to map variables into responses as well as details on how DAKOTA will pass data to and from a simulation code. In this example, the keyword `direct` is used to indicate the use of a function linked directly into DAKOTA. Alternatively, `fork` or `system` executions can be used to invoke instances of a simulation code that is external to DAKOTA, as explained in Section 2.4.4.2 and Chapter 18. The `analysis_driver` keyword indicates the name of the test function. With `fork` or `system`, default file names would be used for passing data between DAKOTA and the simulation code.

The *responses* section of the input file specifies the types of data that the interface will return to DAKOTA. For the example shown in Figure 2.3, the assignment `num_objective_functions = 1` indicates that there is

only one objective function. Since there are no constraints associated with Rosenbrock's function, the keywords for constraint specifications are omitted. The keywords `no_gradients` and `no_hessians` indicate that no derivatives will be provided to the method; none are needed for a parameter study.

2.4 Example Problems

This section serves to familiarize users about how to perform parameter studies, optimization, and uncertainty quantification through their common DAKOTA interface. The initial examples utilize simple built in driver functions; later we show how to utilize DAKOTA to drive the evaluation of user supplied black box code. The examples presented in this chapter are intended to show the simplest use of DAKOTA for several methods of each type. More advanced examples of using DAKOTA for specific purposes are provided in subsequent, topic-based, chapters.

2.4.1 Parameter Studies

Parameter study methods in the DAKOTA toolkit involve the computation of response data sets at a selection of points in the parameter space. These response data sets are not linked to any specific interpretation, so they may consist of any allowable specification from the responses keyword block, i.e., objective and constraint functions, least squares terms and constraints, or generic response functions. This allows the use of parameter studies in direct coordination with optimization, least squares, and uncertainty quantification studies without significant modification to the input file. The two examples given in this subsection are for a two-dimensional tensor product of sample points and a vector parameter study.

2.4.1.1 Two-Dimensional Grid Parameter Study

The 2-D parameter study example problem listed in Figure 2.3 is executed by DAKOTA using the following command:

```
dakota dakota_rosenbrock_2d.in > 2d.out
```

The output of the DAKOTA run is directed to the file named `2d.out`. For comparison, a file named `2d.out.sav` is included in the `Dakota/examples/tutorial` directory. As for many of the examples, DAKOTA provides a report on the best design point located during the study at the end of these output files.

This 2-D parameter study produces the grid of data samples shown in Figure 2.4. In general, a multidimensional parameter study lets one generate a grid in multiple dimensions. The keyword `multidim_parameter_study` indicates that a grid will be generated over all variables. The keyword `partitions` indicates the number of grid partitions in each dimension. For this example, the number of the grid partitions are the same in each dimension (8 partitions) but it would be possible to specify (partitions = 8 2), and have only two partitions over the second input variable. Note that the `graphics` flag in the *strategy* section of the input file could be commented out since, for this example, the iteration history plots created by DAKOTA are not particularly instructive. More interesting visualizations can be created by importing DAKOTA's tabular data into an external graphics/plotting package. Common graphics and plotting packages include Mathematica, Matlab, Microsoft Excel, Origin, Tecplot, and many others. (Sandia National Laboratories and the DAKOTA developers do not endorse any of these commercial products.)

Figure 2.4: Rosenbrock 2-D parameter study example: location of the design points (dots) evaluated.

2.4.1.2 Vector Parameter Study

In addition to the multidimensional parameter study, DAKOTA can perform a vector parameter study, i.e., a parameter study between any two design points in an n-dimensional parameter space.

An input file for the vector parameter study is shown in Figure 2.5. The primary differences between this input file and the previous input file are found in the *variables* and *method* sections. In the variables section, the keywords for the bounds are removed and replaced with the keyword `initial_point` that specifies the starting point for the parameter study. In the method section, the `vector_parameter_study` keyword is used. The `final_point` keyword indicates the stopping point for the parameter study, and `num_steps` specifies the number of steps taken between the initial and final points in the parameter study.

The vector parameter study example problem is executed using the command

```
dakota dakota_rosenbrock_vector.in > vector.out
```

Figure 2.6(a) shows the graphics output created by DAKOTA. For this study, the simple DAKOTA graphics are more useful for visualizing the results. Figure 2.6(b) shows the locations of the 11 sample points generated in this study. It is evident from these figures that the parameter study starts within the banana-shaped valley, marches up the side of the hill, and then returns to the valley. The output file `vector.out.sav` is provided in the `Dakota/examples/tutorial` directory.

In addition to the vector and multidimensional examples shown, DAKOTA also supports list and centered parameter study methods. Refer to Chapter 4 for additional information.

2.4.2 Optimization

DAKOTA's optimization capabilities include a variety of gradient-based and nongradient-based optimization methods. This subsection demonstrates the use of several such methods through the DAKOTA interface.

```
# DAKOTA INPUT FILE - dakota_rosenbrock_vector.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        vector_parameter_study
          final_point = 1.1  1.3
          num_steps = 10

model,
        single

variables,
        continuous_design = 2
          initial_point   -0.3      0.2
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

Figure 2.5: Rosenbrock vector parameter study example: the DAKOTA input file.

2.4.2.1 Gradient-based Unconstrained Optimization

A DAKOTA input file for a gradient-based optimization of Rosenbrock's function is listed in Figure 2.7. The format of the input file is similar to that used for the parameter studies, but there are some new keywords in the responses and method sections. First, in the responses section of the input file, the keyword block starting with `numerical_gradients` specifies that a finite difference method will be used to compute gradients for the optimization algorithm. Note that the Rosenbrock function evaluation code inside DAKOTA has the ability to give analytical gradient values. (To switch from finite difference gradient estimates to analytic gradients, uncomment the `analytic_gradients` keyword and comment out the four lines associated with the `numerical_gradients` specification.) Next, in the method section of the input file, several new keywords have been added. In this section, the keyword `conmin_frcg` indicates the use of the Fletcher-Reeves conjugate gradient algorithm in the CONMIN optimization software package [116] for bound-constrained optimization. The keyword `max_iterations` is used to indicate the computational budget for this optimization (in this case, a single iteration includes multiple evaluations of Rosenbrock's function for the gradient computation steps and the line search steps). The keyword `convergence_tolerance` is used to specify one of CONMIN's convergence criteria (under which CONMIN terminates if the objective function value differs by less than the absolute value of the convergence tolerance for three successive iterations).

(a)

(b)

Figure 2.6: Rosenbrock vector parameter study example: (a) screen capture of the DAKOTA graphics and (b) location of the design points (dots) evaluated.

```
# DAKOTA INPUT FILE - dakota_rosenbrock_grad_opt.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        conmin_frcg
          max_iterations = 100
          convergence_tolerance = 1e-4

model,
        single

variables,
        continuous_design = 2
          initial_point   -1.2  1.0
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
#       analytic_gradients
        numerical_gradients
          method_source dakota
          interval_type forward
          fd_gradient_step_size = 1.e-5
        no_hessians
```

Figure 2.7: Rosenbrock gradient-based unconstrained optimization example: the DAKOTA input file.

This DAKOTA input file is executed using the following command:

```
dakota dakota_rosenbrock_grad_opt.in > grad_opt.out
```

The sample file `grad_opt.out.sav` is included in `Dakota/examples/tutorial` for comparison. When this example problem is executed, DAKOTA creates some iteration history graphics similar to the screen capture shown in Figure 2.8(a). These plots show how the objective function and design parameters change in value during the optimization steps. The scaling of the horizontal and vertical axes can be changed by moving the scroll knobs on each plot. Also, the "Options" button allows the user to plot the vertical axes using a logarithmic scale. Note that log-scaling is only allowed if the values on the vertical axis are strictly greater than zero.

(a)

(b)

Figure 2.8: Rosenbrock gradient-based unconstrained optimization example: (a) screen capture of the DAKOTA graphics and (b) sequence of design points (dots) evaluated (line search points omitted).

Figure 2.8(b) shows the iteration history of the optimization algorithm. The optimization starts at the point $(x_1, x_2) = (-1.2, 1.0)$ as given in the DAKOTA input file. Subsequent iterations follow the banana-shaped valley that curves around toward the minimum point at $(x_1, x_2) = (1.0, 1.0)$. Note that the function evaluations associated with the line search phase of each CONMIN iteration are not shown on the plot. At the end of the

DAKOTA run, information is written to the output file to provide data on the optimal design point. These data include the optimum design point parameter values, the optimum objective and constraint function values (if any), plus the number of function evaluations that occurred and the amount of time that elapsed during the optimization study.

2.4.2.2 Gradient-based Constrained Optimization

This example demonstrates the use of a gradient-based optimization algorithm on a nonlinearly constrained problem. The "textbook" example problem (see Section 2.2) is used for this purpose and the DAKOTA input file for this example problem is shown in Figure 2.9. This input file is similar to the input file for the unconstrained gradient-based optimization example problem involving the Rosenbrock function. Note the addition of commands in the responses section of the input file that identify the number and type of constraints, along with the upper bounds on these constraints. The commands `direct` and `analysis_driver = 'text_book'` specify that DAKOTA will use its internal version of the textbook problem.

```
strategy,
        single_method
          graphics,tabular_graphics_data

method,
#       DOT performs better, but may not be available
#       dot_mmfd,
        conmin_mfd,
          max_iterations = 50,
          convergence_tolerance = 1e-4

variables,
        continuous_design = 2
          initial_point    0.9    1.1
          upper_bounds     5.8    2.9
          lower_bounds     0.5   -2.9
          descriptors      'x1'   'x2'

interface,
        direct
          analysis_driver =       'text_book'

responses,
        objective_functions = 1
        nonlinear_inequality_constraints = 2
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-4
        no_hessians
```

Figure 2.9: Textbook gradient-based constrained optimization example: the DAKOTA input file.

The following command runs this example problem:

```
dakota dakota_textbook.in > textbook.out
```

The `conmin_mfd` keyword in Figure 2.9 tells DAKOTA to use the CONMIN package's implementation of the Method of Feasible Directions (see Section 7.2.3 for more details). A significantly faster alternative is the DOT package's Modified Method of Feasible Directions, i.e. `dot_mmfd` (see Section 7.2.4 for more details). However, DOT is licensed software that may not be available on a particular system. If it is installed on your system and DAKOTA has been compiled without the `--without-dot` flag, you may use it by commenting out the line with `conmin_mfd` and uncommenting the line with `dot_mmfd`.

The file `textbook.out.sav` is included in `Dakota/examples/tutorial` for comparison purposes. The results of the optimization example problem are listed at the end of the `textbook.out` file. This information shows that the optimizer stopped at the point $(x_1, x_2) = (0.5, 0.5)$, where both constraints are approximately satisfied, and where the objective function value is 0.128. The progress of the optimization algorithm is shown in Figure 2.10(a) where the dots correspond to the end points of each iteration in the algorithm. The starting point is $(x_1, x_2) = (0.9, 1.1)$, where both constraints are violated. The optimizer takes a sequence of steps to minimize the objective function while reducing the infeasibility of the constraints. The optimization graphics are also shown in Figure 2.10(b).

2.4.2.3 Nonlinear Least Squares Methods for Optimization

Both the Rosenbrock and textbook example problems can be formulated as least-squares minimization problems (see Section 23.1 and Section 23.2). For example, the Rosenbrock problem can be cast as:

$$\texttt{minimize} \;\; (f_1)^2 + (f_2)^2 \tag{2.4}$$

where $f_1 = 10(x_2 - x_1^2)$ and $f_2 = (1 - x_1)$. When using a least-squares approach to minimize a function, each of the least-squares terms $f_1, f_2, \ldots$ is driven toward zero. This formulation permits the use of specialized algorithms that can be more efficient than general purpose optimization algorithms. See Chapter 8 for more detail on the algorithms used for least-squares minimization, as well as a discussion on the types of engineering design problems (e.g., parameter estimation) that can make use of the least-squares approach.

Figure 2.11 is a listing of the DAKOTA input file `dakota_rosenbrock_ls.in`. This input file differs from the input file shown in Figure 2.7 in several key areas. The responses section of the input file uses the keyword `num_least_squares_terms = 2` instead of the `num_objective_functions = 1`. The method section of the input file shows that the NL2SOL algorithm [17] (`nl2sol`) is used in this example. (The Gauss-Newton, NL2SOL, and NLSSOL SQP algorithms are currently available for exploiting the special mathematical structure of least squares minimization problems).

The input file listed in Figure 2.11 is executed using the command:

```
dakota dakota_rosenbrock_ls.in > leastsquares.out
```

The file `leastsquares.out.sav` is included `Dakota/examples/tutorial` for comparison purposes. The optimization results at the end of this file show that the least squares minimization approach has found the same optimum design point, $(x1, x2) = (1.0, 1.0)$, as was found using the conventional gradient-based optimization approach. The iteration history of the least squares minimization is given in Figure 2.12, and shows that 14 function evaluations were needed for convergence. In this example the least squares approach required about half the number of function evaluations as did conventional gradient-based optimization. In many cases a good least squares algorithm will converge more rapidly in the vicinity of the solution.

(a)

(b)

Figure 2.10: Textbook gradient-based constrained optimization example: (a) screen capture of the DAKOTA graphics shows how the objective function was reduced during the search for a feasible design point and (b) iteration history (iterations marked by solid dots).

2.4.2.4 Nongradient-based Optimization via Pattern Search

In addition to gradient-based optimization algorithms, DAKOTA also contains a variety of nongradient-based algorithms. One particular nongradient-based algorithm for local optimization is known as pattern search (see Chapter 1 for a discussion of local versus global optimization). The DAKOTA input file shown in Figure 2.13

```
# DAKOTA INPUT FILE - dakota_rosenbrock_ls.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        nl2sol
          max_iterations = 100
          convergence_tolerance = 1e-4

model,
        single

variables,
        continuous_design = 2
          initial_point   -1.2  1.0
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        calibration_terms = 2
        analytic_gradients
        no_hessians
```

Figure 2.11: Rosenbrock nonlinear least squares example: the DAKOTA input file.

applies a pattern search method to minimize the Rosenbrock function. While this provides for an interesting comparison to the previous example problems in this chapter, the Rosenbrock function is not the best test case for a pattern search method. That is, pattern search methods are better suited to problems where the gradients are too expensive to evaluate, inaccurate, or nonexistent — situations common among many engineering optimization problems. It also should be noted that nongradient-based algorithms generally are applicable only to unconstrained or bound-constrained optimization problems, although the inclusion of general linear and nonlinear constraints in nongradient-based algorithms is an active area of research in the optimization community. For most users who wish to use nongradient-based algorithms on constrained optimization problems, the easiest route is to create a penalty function, i.e., a composite function that contains the objective function and the constraints, external to DAKOTA and then optimize on this penalty function. Most optimization textbooks will provide guidance on selecting and using penalty functions.

The DAKOTA input file shown in Figure 2.13 is similar to the input file for the gradient-based optimization, except it has a different set of keywords in the method section of the input file, and the gradient specification in the responses section has been changed to `no_gradients`. The pattern search optimization algorithm used is part of the SCOLIB library [63]. See the DAKOTA Reference Manual [3] for more information on the *methods* section commands that can be used with SCOLIB algorithms.

Figure 2.12: Rosenbrock nonlinear least squares example: iteration history for least squares terms f_1 and f_2.

This DAKOTA input file is executed using the following command:

```
dakota dakota_rosenbrock_ps_opt.in > ps_opt.out
```

The file `ps_opt.out.sav` is included in the `Dakota/examples/tutorial` directory. For this run, the optimizer was given an initial design point of $(x_1, x_2) = (0.0, 0.0)$ and was limited to 2000 function evaluations. In this case, the pattern search algorithm stopped short of the optimum at $(x_1, x_2) = (1.0, 1, 0)$, although it was making progress in that direction when it was terminated. (It would have reached the minimum point eventually.)

The iteration history is provided in Figures 2.14(a) and (b), which show the locations of the function evaluations used in the pattern search algorithm. Figure 2.14(c) provides a close-up view of the pattern search function evaluations used at the start of the algorithm. The coordinate pattern is clearly visible at the start of the iteration history, and the decreasing size of the coordinate pattern is evident at the design points move toward $(x_1, x_2) = (1.0, 1.0)$.

While pattern search algorithms are useful in many optimization problems, this example shows some of the drawbacks to this algorithm. While a pattern search method may make good initial progress towards an optimum, it is often slow to converge. On a smooth, differentiable function such as Rosenbrock's function, a nongradient-based method will not be as efficient as a gradient-based method. However, there are many engineering design applications where gradient information is inaccurate or unavailable, which renders gradient-based optimizers ineffective. Thus, pattern search algorithms (and other nongradient-based algorithms such as genetic algorithms as discussed in the next section) are often good choices in complex engineering applications when the quality of gradient data is suspect.

```
# DAKOTA INPUT FILE - dakota_rosenbrock_ps_opt.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        coliny_pattern_search
          max_iterations = 1000
          max_function_evaluations = 2000
          solution_accuracy = 1e-4
          initial_delta = 0.5
          threshold_delta = 1e-4
          exploratory_moves basic_pattern
          contraction_factor = 0.75

model,
        single

variables,
        continuous_design = 2
          initial_point    0.0  0.0
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

Figure 2.13: Rosenbrock pattern search optimization example: the DAKOTA input file.

2.4.2.5 Nongradient-based Optimization via Evolutionary Algorithm

In contrast to pattern search algorithms, which are local optimization methods, evolutionary algorithms (EA) are global optimization methods. As was described above for the pattern search algorithm, the Rosenbrock function is not an ideal test problem for showcasing the capabilities of evolutionary algorithms. Rather, EAs are best suited to optimization problems that have multiple local optima, and where gradients are either too expensive to compute or are not readily available.

Evolutionary algorithms are based on Darwin's theory of survival of the fittest. The EA algorithm starts with a randomly selected population of design points in the parameter space, where the values of the design parameters form a "genetic string," analogous to DNA in a biological system, that uniquely represents each design point in the population. The EA then follows a sequence of generations, where the best design points in the population

Figure 2.14: Rosenbrock pattern search optimization example: (a) screen capture of the DAKOTA graphics, (b) sequence of design points (dots) evaluated and (c) close-up view illustrating the shape of the coordinate pattern used.

(i.e., those having low objective function values) are considered to be the most "fit" and are allowed to survive and reproduce. The EA simulates the evolutionary process by employing the mathematical analogs of processes such as natural selection, breeding, and mutation. Ultimately, the EA identifies a design point (or a family of design points) that minimizes the objective function of the optimization problem. An extensive discussion of EAs is beyond the scope of this text, but may be found in a variety of sources (cf., [61] pp. 149-158; [53]). Currently, the EAs available in DAKOTA include a genetic algorithm for problems involving discrete variables and an evolution strategy with self-adaptation for problems with continuous variables. Details of these algorithms are given in the DAKOTA Reference Manual [3]. The SCOLIB library, which provides the EA software that has been linked into DAKOTA, is described in [63].

Figure 2.15 shows a DAKOTA input file that uses an EA to minimize the Rosenbrock function. For this example the EA has a population size of 50. At the start of the first generation, a random number generator is used to select 50 design points that will comprise the initial population. *[A specific seed value is used in this example to generate repeatable results, although, in general, one should use the default setting which allows the EA to choose a random seed.]* A two-point crossover technique is used to exchange genetic string values between the members of the population during the EA breeding process. The result of the breeding process is a population comprised

```
# DAKOTA INPUT FILE - dakota_rosenbrock_ea_opt.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        coliny_ea
          max_iterations = 100
          max_function_evaluations = 2000
          seed = 11011011
          population_size = 50
          fitness_type merit_function
          mutation_type offset_normal
          mutation_rate 1.0
          crossover_type two_point
          crossover_rate 0.0
          replacement_type chc = 10

model,
        single

variables,
        continuous_design = 2
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

Figure 2.15: Rosenbrock evolutionary algorithm optimization example: the DAKOTA input file.

of the 10 best "parent" design points (elitist strategy) plus 40 new "child" design points. The EA optimization process will be terminated after either 100 iterations (generations of the EA) or 2,000 function evaluations. The EA software available in DAKOTA provides the user with much flexibility in choosing the settings used in the optimization process. See [3] and [63] for details on these settings.

The following command runs DAKOTA on the input file:

```
dakota dakota_rosenbrock_ea_opt.in > ea_opt.out
```

A corresponding output file named `ea_opt.out.sav` appears in `Dakota/examples/tutorial`. The EA optimization results printed at the end of this file show that the best design point found was $(x_1, x_2) =$

$(0.98, 0.95)$. The file `ea_tabular.dat.sav` provides a listing of the design parameter values and objective function values for all 2,000 design points evaluated during the running of the EA. Figure 2.16(a) shows the population of 50 randomly selected design points that comprise the first generation of the EA, and Figure 2.16(b) shows the final population of 50 design points, where most of the 50 points are clustered near $(x_1, x_2) = (0.98, 0.95)$.

(a) (b)

Figure 2.16: Rosenbrock evolutionary algorithm optimization example: 50 design points in the (a) initial and (b) final populations selected by the evolutionary algorithm.

As described above, an EA is not well-suited to an optimization problem involving a smooth, differentiable objective such as the Rosenbrock function. Rather, EAs are better suited to optimization problems where conventional gradient-based optimization fails, such as situations where there are multiple local optima and/or gradients are not available. In such cases, the computational expense of an EA is warranted since other optimization methods are not applicable or impractical. In many optimization problems, EAs often quickly identify promising regions of the design space where the global minimum may be located. However, an EA can be slow to converge to the optimum. For this reason, it can be an effective approach to combine the global search capabilities of a EA with the efficient local search of a gradient-based algorithm in a *hybrid optimization* strategy. In this approach, the optimization starts by using a few iterations of a EA to provide the initial search for a good region of the parameter space (low objective function and/or feasible constraints), and then it switches to a gradient-based algorithm (using the best design point found by the EA as its starting point) to perform an efficient local search for an optimum design point. More information on this hybrid approach is provided in Chapter 16.

In addition to the evolutionary algorithm capabilities in the `coliny_ea` method, there is a single-objective genetic algorithm method called `soga`. For more information on `soga`, see Chapter 7.

2.4.2.6 Multiobjective Optimization

Multiobjective optimization means that there are two or more objective functions that you wish to optimize simultaneously. Often these are conflicting objectives, such as cost and performance. The answer to a multi-objective problem is usually not a single point. Rather, it is a set of points called the Pareto front. Each point on the Pareto front satisfies the Pareto optimality criterion, i.e., locally there exists no other feasible vector that would improve some objective without causing a simultaneous worsening in at least one other objective. Thus a feasible point X' from which small moves improve one or more objectives without worsening any others is not Pareto optimal: it is said to be "dominated" and the points along the Pareto front are said to be "non-dominated".

Often multi-objective problems are addressed by simply assigning weights to the individual objectives, summing the weighted objectives, and turning the problem into a single-objective one which can be solved with a variety of optimization techniques. While this approach provides a useful "first cut" analysis (and is supported within DAKOTA—see Section 7.3), this approach has many limitations. The major limitation is that a local solver with a weighted sum objective will only find one point on the Pareto front; if one wants to understand the effects of changing weights, this method can be computationally expensive. Since each optimization of a single weighted objective will find only one point on the Pareto front, many optimizations must be performed to get a good parametric understanding of the influence of the weights and to achieve a good sampling of the entire Pareto frontier.

Starting with version 3.2 of DAKOTA, a capability to perform multi-objective optimization based on a genetic algorithm method has been available. This method is called `moga`. It is based on the idea that as the population evolves in a GA, solutions that are non-dominated are chosen to remain in the population. Until version 4.0 of DAKOTA, there was a selection_type choice of domination_count that performed a custom fitness assessment and selection operation together. As of version 4.0 of DAKOTA, that functionality has been broken into separate, more generally usable fitness assessment and selection operators called the domination_count fitness assessor and below_limit selector respectively. The effect of using these two operators is the same as the previous behavior of the domination_count selector. This means of selection works especially well on multi-objective problems because it has been specifically designed to avoid problems with aggregating and scaling objective function values and transforming them into a single objective. Instead, the fitness assessor works by ranking population members such that their resulting fitness is a function of the number of other designs that dominate them. The below_limit selector then chooses designs by considering the fitness of each. If the fitness of a design is above a certain limit, which in this case corresponds to a design being dominated by more than a specified number of other designs, then it is discarded. Otherwise it is kept and selected to go to the next generation. The one catch is that this selector will require that a minimum number of selections take place. The `shrinkage_percentage` determines the minimum number of selections that will take place if enough designs are available. It is interpreted as a percentage of the population size that must go on to the subsequent generation. To enforce this, the below_limit selector makes all the selections it would make anyway and if that is not enough, it relaxes its limit and makes selections from the remaining designs. It continues to do this until it has made enough selections. The moga method has many other important features. Complete descriptions can be found in the DAKOTA Reference Manual [3].

Figure 2.17 shows an example input file that demonstrates some of the multi-objective capabilities available with the moga method.

This example has three input variables and two objectives. The example uses objectives different from the Rosenbrock function because we wanted to demonstrate the capability on a problem with two conflicting objectives. This example is taken from a testbed of multi-objective problems [13]. In this example, the three best solutions (as specified by `final_solutions` =3) are written to the output. Additionally, final results from moga are output to a file called `finaldata1.dat` in the directory in which you are running. This `finaldata1.dat` file is simply a list of inputs and outputs. Plotting the output columns against each other allows one to see the Pareto front generated by `moga`. Figure 2.18 shows an example of the Pareto front for this problem. Note that a Pareto front easily shows the tradeoffs between Pareto optimal solutions. For example, look at the point with f1 and f2 values equal to (0.9, 0.23). One cannot improve (minimize) the value of objective function f1 without increasing the value of f2: another point on the Pareto front, (0.63, 0.63) represents a better value of objective f1 but a worse value of objective f2.

Sections 7.2 and 7.3 provide more information on multiobjective optimization. There are three detailed examples provided in Section 23.12.

```
##  DAKOTA INPUT FILE - dakota_mogatest1.in



strategy,
        single
        graphics tabular_graphics_data

method,
        moga
        output silent
        seed = 10983
        final_solutions = 3
        max_function_evaluations = 2500
        initialization_type unique_random
        crossover_type shuffle_random
          num_offspring = 2 num_parents = 2
          crossover_rate = 0.8
        mutation_type replace_uniform
          mutation_rate = 0.1
        fitness_type domination_count
        replacement_type below_limit = 6
          shrinkage_percentage = 0.9
        convergence_type metric_tracker
          percent_change = 0.05 num_generations = 40

variables,
        continuous_design = 3
          initial_point     0            0       0
          upper_bounds      4    4       4
          lower_bounds     -4   -4      -4
          descriptors      'x1'         'x2'    'x3'

interface,
        system
          analysis_driver = 'mogatest1'

responses,
        objective_functions = 2
        no_gradients
        no_hessians
```

Figure 2.17: Multiple objective genetic algorithm (MOGA) example: the DAKOTA input file.

Figure 2.18: Multiple objective genetic algorithm (MOGA) example: Pareto front showing tradeoffs between functions f1 and f2.

2.4.3 Uncertainty Quantification

Uncertainty quantification (UQ) is the process of determining the effect of input uncertainties on response metrics of interest. These input uncertainties may be characterized as either aleatory uncertainties, which are irreducible variabilities inherent in nature, or epistemic uncertainties, which are reducible uncertainties resulting from a lack of knowledge. Since sufficient data is generally available for aleatory uncertainties, probabilistic methods are commonly used for computing response distribution statistics based on input probability distribution specifications. Conversely, for epistemic uncertainties, data is generally sparse, making the use of probability theory questionable and leading to nonprobabilistic methods based on interval specifications.

The subsection demonstrates the use of several methods of uncertainty quantification methods built into DAKOTA. These examples include Monte Carlo random sampling, reliability methods, the representation of a stochastic process by a polynomial chaos expansion, and interval analysis.

2.4.3.1 Monte Carlo Sampling

Figure 2.19 shows the DAKOTA input file for an example problem that demonstrates some of the random sampling capabilities available in DAKOTA. In this example, the design parameters, x1 and x2, will be treated as uncertain parameters that have uniform distributions over the interval [-2, 2]. This is specified in the variables section of the

input file, beginning with the keyword `uniform_uncertain`. Another change from earlier input files, such as Figure 2.5, occurs in the responses section, where the keyword `num_response_functions` is used in place of `num_objective_functions`. The final changes to the input file occur in the method section, where the keyword `sampling` is used; "nond" is an abbreviation for nondeterministic. The other keywords in the methods section of the input file specify the number of samples (200), the seed for the random number generator (17), the sampling method (random), and the response threshold (100.0). The `seed` specification allows a user to obtain repeatable results from multiple runs. If a seed value is not specified, then DAKOTA's sampling methods are designed to generate nonrepeatable behavior (by initializing the seed using a system clock). The keyword `response_thresholds` allows the user to specify threshold values for which DAKOTA will compute statistics on the response function output. Note that a unique threshold value can be specified for each response function.

In this example, DAKOTA will select 200 design points from within the parameter space, evaluate the value of Rosenbrock's function at all 200 points, and then perform some basic statistical calculations on the 200 response values.

This DAKOTA input file is executed using the following command:

```
dakota dakota_rosenbrock_nond.in > nond.out
```

Figure 2.20 shows example results from this sampling method. See the `nond.out.sav` file in directory `Dakota/examples/tutorial` for comparison with results produced by DAKOTA. Note that your results will differ from those in this file if your `seed` value differs or if no `seed` is specified.

As shown in Figure 2.20, the statistical data on the 200 Monte Carlo samples is printed at the end of the output file in the section that starts with "Statistics based on 200 samples." In this section summarizing moment-based statistics, DAKOTA outputs the mean, standard deviation, skewness, and kurtosis estimates for each of the response functions. For example, the mean of the Rosenbrock function given uniform input uncertainties on the input variables is 455.4 and the standard deviation is 536.8. This is a very large standard deviation, due to the fact that the Rosenbrock function varies by three orders of magnitude over the input domain. The skewness is positive, meaning this is a right-tailed distribution, not a symmetric distribution. Finally, the kurtosis (a measure of the "peakedness" of the distribution) indicates that this is a strongly peaked distribution (note that we use a central, standardized kurtosis so that the kurtosis of a normal is zero). After the moment-related statistics, the 95% confidence intervals on the mean and standard deviations are printed. This is followed by the fractions ("Probability Level") of the response function values that are below the response threshold values specified in the input file. For example, 34 percent of the sample inputs resulted in a Rosenbrock function value that was less than or equal to 100, as shown in the line listing the cumulative distribution function values. Finally, there are several correlation matrices printed at the end, showing simple and partial raw and rank correlation matrices. Correlations provide an indication of the strength of a monotonic relationship between input and outputs. More detail on correlation coefficients and their interpretation can be found in Section 6.2.1. More detail about sampling methods in general can be found in Section 6.2. Finally, Figure 2.21 shows the locations of the 200 sample sites within the parameter space of the Rosenbrock function for this example.

2.4.3.2 Reliability Methods - via the Mean Value Method

Reliability methods provide an alternative approach to uncertainty quantification which can be less computationally demanding than sampling techniques. Reliability methods for uncertainty quantification are based on probabilistic approaches that compute approximate response function distribution statistics based on specified uncertain variable distributions. These response statistics include response mean, response standard deviation, and cumulative or complementary cumulative distribution functions (CDF/CCDF). These methods are often more efficient at computing statistics in the tails of the response distributions (events with low probability) than sampling based approaches since the number of samples required to resolve a low probability can be prohibitive.

```
# DAKOTA INPUT FILE - dakota_rosenbrock_nond.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        sampling
          samples = 200 seed = 17
          sample_type random
          response_levels = 100.0

model,
        single

variables,
        uniform_uncertain = 2
          lower_bounds -2.0  -2.0
          upper_bounds  2.0   2.0
          descriptors        'x1'  'x2'

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        response_functions = 1
        no_gradients
        no_hessians
```

Figure 2.19: Monte Carlo sampling example: the DAKOTA input file.

Figure 2.22 shows the DAKOTA input file for an example problem that demonstrates the simplest reliability method, called the mean value method (also referred to as the Mean Value First Order Second Moment method). It is specified with method keyword `local_reliability`. This method calculates the mean and variance of the response function based on information about the mean and variance of the inputs and gradient information at the mean of the inputs. The mean value method is extremely cheap computationally (only five runs were required for the textbook function), but can be quite inaccurate, especially for nonlinear problems and/or problems with uncertain inputs that are significantly non-normal. More detail on the mean value method can be found in the Local Reliability Methods section of the DAKOTA Theory Manual [2], and more detail on reliability methods in general (including the more advanced methods) is found in Section 6.3.

Example output from the mean value method is displayed in Figure 2.23. Note that since the mean of both inputs is 1, the mean value of the output for response 1 is zero. However, the mean values of the constraints are both 0.5. The mean value results indicate that variable x1 is more important in constraint 1 while x2 is more important in constraint 2, which is the case based on Equation 2.3.

This DAKOTA input file is executed using the following command:

```
Statistics based on 200 samples:

Moment-based statistics for each response function:
                            Mean           Std Dev          Skewness          Kurtosis
 response_fn_1  4.5540183516e+02  5.3682678089e+02  1.6661798252e+00  2.7925726822e+00

95% confidence intervals for each response function:
                    LowerCI_Mean      UpperCI_Mean    LowerCI_StdDev    UpperCI_StdDev
 response_fn_1  3.8054757609e+02  5.3025609422e+02  4.8886795789e+02  5.9530059589e+02

Level mappings for each response function:
Cumulative Distribution Function (CDF) for response_fn_1:
     Response Level  Probability Level  Reliability Index  General Rel Index
     --------------  -----------------  -----------------  -----------------
   1.0000000000e+02   3.4000000000e-01

Probability Density Function (PDF) histograms for each response function:
PDF for response_fn_1:
          Bin Lower          Bin Upper      Density Value
          ---------          ---------      -------------
   1.1623549854e-01   1.0000000000e+02   3.4039566059e-03
   1.0000000000e+02   2.7101710856e+03   2.5285698843e-04

Simple Correlation Matrix among all inputs and outputs:
                       x1           x2 response_fn_1
          x1  1.00000e+00
          x2 -5.85097e-03  1.00000e+00
response_fn_1 -9.57746e-02 -5.08193e-01  1.00000e+00

Partial Correlation Matrix between input and output:
             response_fn_1
          x1 -1.14659e-01
          x2 -5.11111e-01

Simple Rank Correlation Matrix among all inputs and outputs:
                       x1           x2 response_fn_1
          x1  1.00000e+00
          x2 -6.03315e-03  1.00000e+00
response_fn_1 -1.15360e-01 -5.04661e-01  1.00000e+00

Partial Rank Correlation Matrix between input and output:
             response_fn_1
          x1 -1.37154e-01
          x2 -5.08762e-01
```

Figure 2.20: Results of Monte Carlo Sampling on the Rosenbrock Function

Figure 2.21: Monte Carlo sampling example: locations in the parameter space of the 200 Monte Carlo samples using a uniform distribution for both x_1 and x_2.

```
dakota dakota_mv.in > mv.out
```

See the file `mv.out.sav` in `Dakota/examples/tutorial` for comparison with results from DAKOTA.

2.4.3.3 Polynomial Chaos

The term "Polynomial Chaos" refers to the representation of a stochastic process as a polynomial expansion in random (or stochastic) variables. This representation acts as a response surface that maps stochastic inputs to stochastic outputs. Desired statistics can then be obtained from the response surface either analytically or by re-sampling the fast surrogate. Exponential convergence of the error with increasing polynomial order can be obtained by using (an) orthogonal polynomial series whose weighting function(s) is/are the probability density functions of the stochastic inputs. Coefficients in the Chaos expansion are determined through orthogonal projection. For non-intrusive implementations, such as in DAKOTA, numerical integration via quadrature or cubature is used to evaluate the orthogonal projections. Additional details regarding the method are provided in Section 6.4.

A typical DAKOTA input file for performing an uncertainty quantification using polynomial chaos expansions is shown in Figure 2.24, `dakota_pce.in`. In this example, we compute CDF probabilities for six response levels of Rosenbrock's function. Since Rosenbrock is a fourth order polynomial and we employ a fourth-order expansion using an optimal basis (Legendre for uniform random variables), we can readily obtain a polynomial expansion which exactly matches the Rosenbrock function. In this example, we select Gaussian quadratures using an anisotropic approach (fifth-order quadrature in x_1 and third-order quadrature in x_2), resulting in a total of 15 function evaluations to compute the PCE coefficients.

The tensor product quadature points upon which the expansion is calculated are shown in Figure 2.25. The tensor product generates all combinations of values from each individual dimension: it is an all-way pairing of points.

Once the expansion coefficients have been calculated, some statistics are available analytically and others must be evaluated numerically. For the numerical portion, the input file specifies the use of 10000 samples, which will be evaluated on the expansion to compute the CDF probabilities. In Figure 2.26, excerpts from the results summary are presented, where we first see a summary of the PCE coefficients which exactly reproduce Rosenbrock for a Legendre polynomial basis. The analytic statistics for mean, standard deviation, and COV are then presented.

```
# test file with a specific test.  The  is used to designate lines

interface,
        system asynch
          analysis_driver = 'text_book'

variables,
        lognormal_uncertain = 2
          means                 =  1.      1.
          std_deviations        =  0.5     0.5
          descriptors           =  'TF1ln' 'TF2ln'

responses,
        response_functions = 3
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-4
        no_hessians

strategy,
        single_method #graphics

method,
        local_reliability
```

Figure 2.22: Mean Value Reliability Method: the DAKOTA input file.

For example, the mean is 455.66 and the standard deviation is 606.56. The moments are followed by global sensitivity indices (Sobol indices).This example shows that variable x1 has the largest main effect (0.497) as compared with variable x2 (0.296) or the interaction between x1 and x2 (0.206). After the global sensitivity indices, the local, analytic random variable sensitivities are presented, evaluated at the mean values. Finally, we see the numerical results for the CDF probabilities based on 10000 samples performed on the expansion. For example, the probability that the Rosenbrock function is less than 100 over these two uncertain variables is 0.342. Note that this is a very similar estimate to what was obtained using 200 Monte Carlo samples, with fewer function evaluations.

2.4.3.4 Interval Analysis

Interval analysis is often used to model epistemic uncertainty. In interval analysis, one assumes that nothing is known about an epistemic uncertain variable except that its value lies somewhere within an interval. In this situation, it is NOT assumed that the value has a uniform probability of occuring within the interval. Instead, the interpretation is that any value within the interval is a possible value or a potential realization of that variable. In interval analysis, the uncertainty quantification problem is one of determining the resulting bounds on the output (defining the output interval) given interval bounds on the inputs. Again, any output response that falls within the output interval is a possible output with no frequency information assigned to it.

We can do interval analysis using either `global_interval_est` or `local_interval_est`. In the global

```
-----------------------------------------------------------------
MV Statistics for response_fn_1:
  Approximate Mean Response                  =  0.0000000000e+00
  Approximate Standard Deviation of Response =  0.0000000000e+00
  Importance Factors not available.
MV Statistics for response_fn_2:
  Approximate Mean Response                  =  5.0000000000e-01
  Approximate Standard Deviation of Response =  1.0307764064e+00
  Importance Factor for variable TF1ln       =  9.4117647059e-01
  Importance Factor for variable TF2ln       =  5.8823529412e-02
MV Statistics for response_fn_3:
  Approximate Mean Response                  =  5.0000000000e-01
  Approximate Standard Deviation of Response =  1.0307764064e+00
  Importance Factor for variable TF1ln       =  5.8823529412e-02
  Importance Factor for variable TF2ln       =  9.4117647059e-01
-----------------------------------------------------------------
```

Figure 2.23: Results of the Mean Value Method on the Textbook Function

```
strategy,
        single_method #graphics

method,
        polynomial_chaos
          quadrature_order   = 5
          dimension_preference = 5 3
          samples = 10000
          seed = 12347 rng rnum2
          response_levels =
          .1 1. 50. 100. 500. 1000.
          variance_based_decomp #univariate_effects

variables,
        uniform_uncertain = 2
          lower_bounds      = -2.  -2.
          upper_bounds      =  2.   2.
          descriptors       = 'x1' 'x2'

interface,
        direct
        analysis_driver = 'rosenbrock'

responses,
        response_functions = 1
        no_gradients
        no_hessians
```

Figure 2.24: DAKOTA input file for performing UQ using polynomial chaos expansions.

Figure 2.25: Rosenbrock polynomial chaos example: tensor product quadrature points.

approach, one uses either a global optimization method or a sampling method to assess the bounds, whereas the local method uses gradient information in a derivative-based optimization approach.

An example of interval estimation is found in the test file `dakota_uq_interval.in`, and also in Figure 2.27, with example results in Figure 2.28. This example is a demonstration of calculating interval bounds for three outputs of the cantilever beam problem. The cantilever beam problem is described in detail in Section 23.10. Given input intervals of [1,10] on beam width and beam thickness, we can see that the interval estimate of beam weight is approximately [1,100].

2.4.4 User Supplied Simulation Code Examples

This subsection provides examples of how to use DAKOTA to drive user supplied black box code.

2.4.4.1 Optimization with a User-Supplied Simulation Code - Case 1

Many of the previous examples made use of the direct interface to access the Rosenbrock and textbook test functions that are compiled into DAKOTA. In engineering applications, it is much more common to use the `system` or `fork` interface approaches within DAKOTA to manage external simulation codes. In both of these cases, the communication between DAKOTA and the external code is conducted through the reading and writing of short text files. For this example, the C++ program `rosenbrock.C` in `Dakota/test` is used as the simulation code. This file is compiled to create the stand-alone `rosenbrock` executable that is referenced as the `analysis_driver` in Figure 2.29. This stand-alone program performs the same function evaluations as DAKOTA's internal Rosenbrock test function.

Figure 2.29 shows the text of the DAKOTA input file named `dakota_rosenbrock_syscall.in` that is provided in the directory `Dakota/examples/tutorial`. The only differences between this input file and the one in Figure 2.7 occur in the *interface* keyword section. The keyword `system` indicates that DAKOTA will use system calls to create separate Unix processes for executions of the user-supplied simulation code. The name of the simulation code, and the names for DAKOTA's parameters and results file are specified using the `analysis_driver`, `parameters_file`, and `results_file` keywords, respectively.

```
Polynomial Chaos coefficients for response_fn_1:
        coefficient   u1   u2
        ----------- ---- ----
   4.5566666667e+02   P0   P0
  -4.0000000000e+00   P1   P0
   9.1695238095e+02   P2   P0
  -9.9475983006e-14   P3   P0
   3.6571428571e+02   P4   P0
  -5.3333333333e+02   P0   P1
  -3.9968028887e-14   P1   P1
  -1.0666666667e+03   P2   P1
  -3.3573144265e-13   P3   P1
   1.2829737273e-12   P4   P1
   2.6666666667e+02   P0   P2
   2.2648549702e-13   P1   P2
   4.8849813084e-13   P2   P2
   2.8754776338e-13   P3   P2
  -2.8477220582e-13   P4   P2
-------------------------------------------------------------------
Statistics derived analytically from polynomial expansion:

Moment-based statistics for each response function:
                            Mean           Std Dev          Skewness          Kurtosis
response_fn_1
  expansion:    4.5566666667e+02  6.0656024184e+02
  numerical:    4.5566666667e+02  6.0656024184e+02  1.9633285271e+00  3.3633861456e+00

Covariance among response functions:
[[  3.6791532698e+05 ]]

Local sensitivities for each response function evaluated at uncertain variable means:
response_fn_1:
 [ -2.0000000000e+00  2.4055757386e-13 ]

Global sensitivity indices for each response function:
response_fn_1 Sobol indices:
                                   Main             Total
                       4.9746891383e-01  7.0363551328e-01 x1
                       2.9636448672e-01  5.0253108617e-01 x2
                            Interaction
                       2.0616659946e-01 x1 x2

Statistics based on 10000 samples performed on polynomial expansion:

Probability Density Function (PDF) histograms for each response function:
PDF for response_fn_1:
          Bin Lower          Bin Upper      Density Value
          ---------          ---------      -------------
   6.8311107124e-03   1.0000000000e-01   2.0393073423e-02
   1.0000000000e-01   1.0000000000e+00   1.3000000000e-02
   1.0000000000e+00   5.0000000000e+01   4.7000000000e-03
   5.0000000000e+01   1.0000000000e+02   1.9680000000e-03
   1.0000000000e+02   5.0000000000e+02   9.2150000000e-04
   5.0000000000e+02   1.0000000000e+03   2.8300000000e-04
   1.0000000000e+03   3.5755437782e+03   5.7308286215e-05

Level mappings for each response function:
Cumulative Distribution Function (CDF) for response_fn_1:
     Response Level  Probability Level  Reliability Index  General Rel Index
     --------------  -----------------  -----------------  -----------------
   1.0000000000e-01   1.9000000000e-03
   1.0000000000e+00   1.3600000000e-02
   5.0000000000e+01   2.4390000000e-01
   1.0000000000e+02   3.4230000000e-01
   5.0000000000e+02   7.1090000000e-01
   1.0000000000e+03   8.5240000000e-01
-------------------------------------------------------------------
```

Figure 2.26: Excerpt of UQ output for polynomial chaos example.

```
strategy,
        single_method tabular_graphics_data

method,
        global_interval_est ego
          seed = 1234567 rng rnum2
          output verbose

variables,
        interval_uncertain = 2
          num_intervals   = 1 1
          interval_probs  = 1.0 1.0
          interval_bounds = 1. 10. 1. 10
          descriptors 'w' 't'
        continuous_state = 4
          initial_state     =  40000. 29.E+6 500. 1000.
          descriptors       =  'R' 'E' 'X' 'Y'

interface,
        direct
          analysis_driver = 'cantilever'

responses,
        response_functions = 3
        response_descriptors = 'weight' 'stress' 'displ'
        no_gradients
        no_hessians
```

Figure 2.27: DAKOTA input file for performing UQ using interval analysis.

```
-----------------------------------------------------------------
Min and Max estimated values for each response function:
weight:  Min = 1.0000169352e+00  Max = 9.9999491948e+01
stress:  Min = -9.7749994284e-01  Max = 2.1499428450e+01
displ:  Min = -9.9315672724e-01  Max = 6.7429714485e+01
----------------------------------------------------------------
```

Figure 2.28: Excerpt of UQ output for interval example.

This example problem is executed using the command:

```
dakota dakota_rosenbrock_syscall.in > syscall.out
```

This run of DAKOTA takes longer to complete than the previous gradient-based optimization example since the `system` interface method has additional process creation and file I/O overhead, as compared to the internal communication that occurs when the `direct` interface method is used. File `syscall.out.sav` in the `Dakota/examples/tutorial` directory permits comparison with output results you get by executing the command given above.

To gain a better understanding of what exactly DAKOTA is doing with the `system` interface approach, add the keywords `file_tag` and `file_save` to the interface specification and re-run DAKOTA. Check the listing of the local directory and you will see many new files with names such as `params.in.1`, `params.in.2`, etc., and `results.out.1`, `results.out.2`, etc. There is one `params.in.X` file and one `results.out.X` file for each of the function evaluations performed by DAKOTA. This is the file listing for `params.in.1`:

```
                     2 variables
-1.200000000000000e+00 x1
 1.000000000000000e+00 x2
                     1 functions
                     1 ASV_1:obj_fn
                     2 derivative_variables
                     1 DVV_1:x1
                     2 DVV_2:x2
                     0 analysis_components
```

The basic pattern is that of array lengths and string identifiers followed by listings of the array entries, where the arrays consist of the variables, the active set vector (ASV), the derivative values vector (DVV), and the analysis components (AC). For the variables array, the first line gives the total number of variables (2) and the "variables" string identifier, and the subsequent two lines provide the array listing for the two variable values (-1.2 and 1.0) and descriptor tags ("x1" and "x2" from the DAKOTA input file). The next array conveys the ASV, which indicates what simulator outputs are needed. The first line of the array gives the total number of response functions (1) and the "functions" string identifier, followed by one ASV code and descriptor tag ("ASV_1") for each function. In this case, the ASV value of 1 indicates that DAKOTA is requesting that the simulation code return the response function value in the file `results.out.X`. (Possible ASV values: 1 = value of response function value, 2 = response function gradient, 4 = response function Hessian, and any sum of these for combinations up to 7 = response function value, gradient, and Hessian; see 11.7 for more detail.) The next array provides the DVV, which defines the variable identifiers used in computing derivatives. The first line of the array gives the number of derivative variables (2) and the "derivative_variables" string identifier, followed by the listing of the two DVV variable identifiers (the first and second variables) and descriptor tags ("DVV_1" and "DVV_2"). The final array provides the AC array used to provide additional strings for use by the simulator (e.g., to provide the name of a particular mesh file). The first line of the array gives the total number of analysis components (0) and the "analysis_components" string identifier, followed by the listing of the array, which is empty in this case.

The executable program rosenbrock reads in the `params.in.X` file and evaluates the objective function at the given values for x_1 and x_2. Then, rosenbrock writes out the objective function data to the `results.out.X` file. Here is the listing for the file `results.out.1`:

```
2.420000000000000e+01 f
```

The value shown above is the value of the objective function, and the descriptor 'f' is an optional tag returned by the simulation code. When the system call has completed, DAKOTA reads in the data from the `results.in.X`

file and processes the results. DAKOTA then continues with additional executions of the rosenbrock program until the optimization process is complete.

2.4.4.2 Optimization with a User-Supplied Simulation Code - Case 2

In many situations the user-supplied simulation code cannot be modified to read and write the `params.in.X` `file` and the `results.out.X` file, as described above. Typically, this occurs when the simulation code is a commercial or proprietary software product that has specific input file and output file formats. In such cases, it is common to replace the executable program name in the DAKOTA input file with the name of a Unix shell script containing a sequence of commands that read and write the necessary files and run the simulation code. For

```
# DAKOTA INPUT FILE - dakota_rosenbrock_syscall.in


strategy,
        single_method
          graphics,tabular_graphics_data

method,
        conmin_frcg
          max_iterations = 100
          convergence_tolerance = 1e-4

model,
        single

variables,
        continuous_design = 2
          initial_point   -1.2  1.0
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"

interface,
        system
          analysis_driver = 'rosenbrock'
          parameters_file = 'params.in'
          results_file    = 'results.out'

responses,
        objective_functions = 1
        numerical_gradients
          method_source dakota
          interval_type forward
          fd_gradient_step_size = 1.e-5
        no_hessians
```

Figure 2.29: DAKOTA input file for gradient-based optimization using the system call interface to an external rosenbrock simulator.

example, the executable program named `rosenbrock` listed in Figure 2.29 could be replaced by a Unix C-shell script named `simulator_script`, with the script containing a sequence of commands to perform the following steps: insert the data from the `parameters.in.X` file into the input file of the simulation code, execute the simulation code, post-process the files generated by the simulation code to compute response data, and return the response data to DAKOTA in the `results.out.X` file. The steps that are typically used in constructing and using a Unix shell script are described in Section 18.1.

2.5 Where to Go from Here

This chapter has provided an introduction to the basic capabilities of DAKOTA including parameter studies, various types of optimization, and uncertainty quantification sampling. More information on the DAKOTA input file syntax is provided in the remaining chapters in this manual and in the DAKOTA Reference Manual [3]. Additional example problems that demonstrate some of DAKOTA's advanced capabilities are provided in Chapter 6, Chapter 9, Chapter 16, Chapter 18, and Chapter 23.

Here are a few pointers to sections of this manual that many new users find useful:

- Chapter 15 describes the different DAKOTA output file formats, including commonly encountered error messages.
- Chapter 18 demonstrates how to employ DAKOTA with a user-supplied simulation code.
  Most DAKOTA users will follow the approach described in Chapter 18.
- Chapter 20 provides guidelines on how to choose an appropriate optimization, uncertainty quantification, or parameter study method based on the characteristics of your application.
- Chapter 21 describes the file restart and data re-use capabilities of DAKOTA.

Chapter 3

DAKOTA Capability Overview

3.1 Purpose

This chapter provides a brief overview of DAKOTA's capabilities. Emerging capabilities in solution verification and Bayesian calibration/uncertainty quantification are not yet documented here. Additional details and example problems are provided in subsequent chapters in this manual.

3.2 Parameter Study Methods

Parameter studies are often performed to explore the effect of parametric changes within simulation models. DAKOTA users may select from four parameter study methods.

Multidimensional: Forms a regular lattice or grid in an n-dimensional parameter space, where the user specifies the number of intervals used for each parameter.

Vector: Performs a parameter study along a line between any two points in an n-dimensional parameter space, where the user specifies the number of steps used in the study.

Centered: Given a point in an n-dimensional parameter space, this method evaluates nearby points along the coordinate axes of the parameter space. The user selects the number of steps and the step size.

List: The user supplies a list of points in an n-dimensional space where DAKOTA will evaluate response data from the simulation code.

Additional information on these methods is provided in Chapter 4.

3.3 Design of Experiments

Design of experiments are often used to explore the parameter space of an engineering design problem, for example to perform global sensitivity analysis. In design of experiments, especially design of computer experiments, one wants to generate input points that provide good coverage of the input parameter space. There is significant overlap between design of experiments and sampling methods, and both techniques can yield similar results about response function behavior and the relative importance of the input variables. We consider design of experiment

methods to generate sets of uniform random variables on the interval $[0, 1]$, with the goal of characterizing the behavior of the response functions over the input parameter ranges of interest. Uncertainty quantification, in contrast, involves characterizing the uncertain input variables with probability distributions such as normal, Weibull, triangular, etc., sampling from the input distributions, and propagating the input uncertainties to obtain a cumulative distribution function on the output or system response. We typically use the Latin Hypercube Sampling software (also developed at Sandia) for generating samples on input distributions used in uncertainty quantification. LHS is explained in more detail in the subsequent section 3.4. Two software packages are available in DAKOTA for design of computer experiments, DDACE (developed at Sandia Labs) and FSUDACE (developed at Florida State University).

DDACE (Distributed Design and Analysis of Computer Experiments): The DACE package includes both stochastic sampling methods and classical design of experiments methods [114]. The stochastic methods are Monte Carlo (random) sampling, Latin Hypercube sampling, orthogonal array sampling, and orthogonal array-latin hypercube sampling. The orthogonal array sampling allows for the calculation of main effects. The DDACE package currently supports variables that have either normal or uniform distributions. However, only the uniform distribution is available in the DAKOTA interface to DDACE. The classical design of experiments methods in DDACE are central composite design (CCD) and Box-Behnken (BB) sampling. A grid-based sampling method also is available. DDACE is available under a GNU Lesser General Public License and is distributed with DAKOTA.

FSUDace (Florida State University Design and Analysis of Computer Experiments): The FSUDace package provides quasi-Monte Carlo sampling (Halton and Hammersley) and Centroidal Voronoi Tessellation (CVT) methods. The quasi-Monte Carlo and CVT methods are designed with the goal of low discrepancy. Discrepancy refers to the non-uniformity of the sample points within the unit hypercube. Low discrepancy sequences tend to cover the unit hypercube reasonably uniformly. Quasi-Monte Carlo methods produce low discrepancy sequences, especially if one is interested in the uniformity of projections of the point sets onto lower dimensional faces of the hypercube. CVT does very well volumetrically: it spaces the points fairly equally throughout the space, so that the points cover the region and are isotropically distributed with no directional bias in the point placement. FSUDace is available under a GNU Lesser General Public License and is distributed with DAKOTA.

PSUADE (Problem Solving Environment for Uncertainty Analysis and Design Exploration): PSUADE is a Lawrence Livermore National Laboratory tool for metamodeling, sensitivity analysis, uncertainty quantification, and optimization. Its features include non-intrusive and parallel function evaluations, sampling and analysis methods, an integrated design and analysis framework, global optimization, numerical integration, response surfaces (MARS and higher order regressions), graphical output with Pgplot or Matlab, and fault tolerance [113]. DAKOTA includes a prototype interface to its MOAT sampling method, a valuable tool for global sensitivity analysis.

Additional information on these methods is provided in Chapter 5.

3.4 Uncertainty Quantification

Uncertainty quantification methods (also referred to as nondeterministic analysis methods) involve the computation of probabilistic information about response functions based on sets of simulations taken from the specified probability distributions for uncertain input parameters. Put another way, these methods perform a forward uncertainty propagation in which probability information for input parameters is mapped to probability information for output response functions. We usually distinguish the UQ methods in terms of their capability to handle aleatory or epistemic uncertainty. Input uncertainties may be characterized as either aleatory uncertainties, which are irreducible variabilities inherent in nature, or epistemic uncertainties, which are reducible uncertainties resulting from a lack of knowledge. Since sufficient data is generally available for aleatory uncertainties, probabilistic methods

are commonly used for computing response distribution statistics based on input probability distribution specifications. Conversely, for epistemic uncertainties, data is generally sparse, making the use of probability theory questionable and leading to non-probabilistic methods based on interval specifications. The aleatory UQ methods in DAKOTA include various sampling-based approaches (e.g., Monte Carlo and Latin Hypercube sampling), local and global reliability methods, and stochastic expansion approaches. The epistemic UQ methods include interval analysis and Dempster-Shafer evidence theory.

LHS (Latin Hypercube Sampling): This package provides both Monte Carlo (random) sampling and Latin Hypercube sampling methods, which can be used with probabilistic variables in DAKOTA that have the following distributions: normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, poisson, binomial, negative binomial, geometric, hypergeometric, and user-supplied histograms. In addition, LHS accounts for correlations among the variables [69], which can be used to accommodate a user-supplied correlation matrix or to minimize correlation when a correlation matrix is not supplied. The LHS package currently serves two purposes: (1) it can be used for uncertainty quantification by sampling over uncertain variables characterized by probability distributions, or (2) it can be used in a DACE mode in which any design and state variables are treated as having uniform distributions (see the `all_variables` flag in the DAKOTA Reference Manual [3]). The LHS package historically came in two versions: "old" (circa 1980) and "new" (circa 1998), but presently only the latter is supported in DAKOTA, requiring a Fortran 90 compiler. This "new" LHS is available under a separate GNU Lesser General Public License and is distributed with DAKOTA. In addition to a standard sampling study, we support the capability to perform "incremental" LHS, where a user can specify an initial LHS study of N samples, and then re-run an additional incremental study which will double the number of samples (to 2N, with the first N being carried from the initial study). The full incremental sample of size 2N is also a Latin Hypercube, with proper stratification and correlation. Finally, DAKOTA offers preliminary support for importance sampling using LHS, specified with `importance`.

Reliability Methods: This suite of methods includes both local and global reliability methods. Local methods include first- and second-order versions of the Mean Value method (MVFOSM and MVSOSM) and a variety of most probable point (MPP) search methods, including the Advanced Mean Value method (AMV and AMV^2), the iterated Advanced Mean Value method (AMV+ and AMV^2+), the Two-point Adaptive Nonlinearity Approximation method (TANA-3), and the traditional First Order and Second Order Reliability Methods (FORM and SORM) [62]. Each of the MPP search techniques solve local optimization problems in order to locate the MPP, which is then used as the point about which approximate probabilities are integrated (using first- or second-order integrations in combination with refinements based on importance sampling). Reliability mappings may involve computing reliability and probability levels for prescribed response levels (forward reliability analysis, commonly known as the reliability index approach or RIA) or computing response levels for prescribed reliability and probability levels (inverse reliability analysis, commonly known as the performance measure approach or PMA). Approximation-based MPP search methods (AMV, AMV^2, AMV+, AMV^2+, and TANA) may be applied in either x-space or u-space, and mappings may involve either cumulative or complementary cumulative distribution functions. Global reliability methods are designed to handle nonsmooth and multimodal failure surfaces, by creating global approximations based on Gaussian process models. They accurately resolve a particular contour of a response function and then estimate probabilities using multimodal adaptive importance sampling.

Stochastic Expansion Methods: The objective of these techniques is to characterize the response of systems whose governing equations involve stochastic coefficients. The development of these techniques mirrors that of deterministic finite element analysis utilizing the notions of projection, orthogonality, and weak convergence [43], [44]. Rather than estimating point probabilities, they form an approximation to the functional relationship between response functions and their random inputs, which provides a more complete uncertainty representation for use in multi-code simulations. Expansion methods include the Wiener-Askey generalized polynomial chaos expansion (PCE), which employs a family of multivariate orthogonal polynomials that are well matched to particular input probability distributions, and stochastic collocation (SC), which employs multivariate Lagrange interpolation polynomials. For PCE, expansion coefficients may be evaluated using a spectral projection approach (based on

sampling, quadrature, or sparse grid methods for integration) or a point collocation approach (based on linear regression). For SC, interpolants may be formed over tensor-product quadrature grids or Smolyak sparse grids. Both methods provide analytic response moments; however, CDF/CCDF probabilities are evaluated by sampling on the expansion.

Interval Analysis: Interval analysis is often used to model epistemic uncertainty. In interval analysis, one assumes that nothing is known about an epistemic uncertain variable except that its value lies somewhere within an interval. In this situation, it is NOT assumed that the value has a uniform probability of occurring within the interval. Instead, the interpretation is that any value within the interval is a possible value or a potential realization of that variable. In interval analysis, the uncertainty quantification problem is one of determining the resulting bounds on the output (defining the output interval) given interval bounds on the inputs. Again, any output response that falls within the output interval is a possible output with no frequency information assigned to it.

We have the capability to perform interval analysis using either global or local methods. In the global approach, one uses either a global optimization method (based on a Gaussian process surrogate model) or a sampling method to assess the bounds. The local method uses gradient information in a derivative-based optimization approach, using either SQP (sequential quadratic programming) or a NIP (nonlinear interior point) method to obtain bounds.

Dempster-Shafer Theory of Evidence: The objective of Evidence theory is to model the effects of epistemic uncertainties. Epistemic uncertainty refers to the situation where one does not know enough to specify a probability distribution on a variable. Sometimes epistemic uncertainty is referred to as subjective, reducible, or lack of knowledge uncertainty. In contrast, aleatory uncertainty refers to the situation where one does have enough information to specify a probability distribution. In Dempster-Shafer theory of evidence, the uncertain input variables are modeled as sets of intervals. The user assigns a basic probability assignment (BPA) to each interval, indicating how likely it is that the uncertain input falls within the interval. The intervals may be overlapping, contiguous, or have gaps. The intervals and their associated BPAs are then propagated through the simulation to obtain cumulative distribution functions on belief and plausibility. Belief is the lower bound on a probability estimate that is consistent with the evidence, and plausibility is the upper bound on a probability estimate that is consistent with the evidence. In addition to the full evidence theory structure, we have a simplified capability for users wanting to perform pure interval analysis (e.g. what is the interval on the output given intervals on the input) using either global or local optimization methods. Interval analysis is often used to model epistemic variables in nested analyses, where probability theory is used to model aleatory variables.

Additional information on these methods is provided in Chapter 6.

3.5 Optimization

Several optimization software packages have been integrated with DAKOTA. These include freely-available software packages developed by research groups external to Sandia Labs, Sandia-developed software that has been released to the public under GNU licenses, and commercially-developed software. These optimization software packages provide the DAKOTA user with access to well-tested, proven methods for use in engineering design applications, as well as access to some of the newest developments in optimization algorithm research.

HOPSPACK: is a library that enables the implementation of hybrid optimization algorithms [91]. Currently the only method exposed is an asynchronous implementation of generating set search known as asynchronous parallel pattern search (APPS) [56]. It can handle unconstrained problems as well as those with bound constraints, linear constraints [58], and general nonlinear constraints [57]. APPS was previously integrated with DAKOTA via the APPSPACK software, but is now integrated through it's successor (HOPSPACK). HOPSPACK is available to the public under the GNU LGPL and the source code is included with DAKOTA (web page: `https://software.sandia.gov/trac/hopspack`).

SCOLIB: Methods for nongradient-based local and global optimization which utilize the Common Optimization Library INterface (COLIN). SCOLIB (formerly known as COLINY) currently includes evolutionary algorithms (including several genetic algorithms and Evolutionary Pattern Search), simple pattern search, Monte Carlo sampling, and the DIRECT and Solis-Wets algorithms. SCOLIB also include interfaces to third-party optimizer COBYLA2. This software is available to the public under a BSD license through ACRO (A Common Repository for Optimizers) and the source code for SCOLIB is included with DAKOTA (web page: `http://software.sandia.gov/trac/acro`).

CONMIN (CONstrained MINimization): Methods for gradient-based constrained and unconstrained optimization [116]. The constrained optimization algorithm is the method of feasible directions (MFD) and the unconstrained optimization algorithm is the Fletcher-Reeves conjugate gradient (CG) method. This software is freely available to the public from NASA, and the CONMIN source code is included with DAKOTA.

DOT (Design Optimization Tools): Methods for gradient-based optimization for constrained and unconstrained optimization problems [118]. The algorithms available for constrained optimization are modified-MFD, SQP, and sequential linear programming (SLP). The algorithms available for unconstrained optimization are the Fletcher-Reeves CG method and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) quasi-Newton technique. DOT is a commercial software product of Vanderplaats Research and Development, Inc. (web page: `http://www.vrand.com`). Sandia National Laboratories and Los Alamos National Laboratory have limited seats for DOT. *Other users may obtain their own copy of DOT and compile it with the DAKOTA source code by following the steps given in the file* `Dakota/INSTALL`.

JEGA: provides SOGA and MOGA (single- and multi-objective genetic algorithms) optimization methods. The SOGA method provides a basic GA optimization capability that uses many of the same software elements as the MOGA method. The MOGA package allows for the formulation of multiobjective optimization problems without the need to specify weights on the various objective function values. The MOGA method directly identifies nondominated design points that lie on the Pareto front through tailoring of its genetic search operators. The advantage of the MOGA method versus conventional multiobjective optimization with weight factors (see Section 3.6), is that MOGA finds points along the entire Pareto front whereas the multiobjective optimization method produces only a single point on the Pareto front. The advantage of the MOGA method versus the Pareto-set optimization strategy (see Section 3.9) is that MOGA is better able to find points on the Pareto front when the Pareto front is nonconvex. However, the use of a GA search method in MOGA causes the MOGA method to be much more computationally expensive than conventional multiobjective optimization using weight factors.

NCSUOpt: Nongradient-based optimizers from North Carolina State University, including DIRECT and, eventually, impicit filtering (web site: `http://www4.ncsu.edu/~ctk/matlab_darts.html`). We currently incorporate only an implementation of the DIRECT (DIviding RECTangles) algorithm [41]. While this is somewhat redundant with DIRECT supplied by SCOLIB, we have found that NCCSU DIRECT performs better in some cases, and presently we maintain both versions in DAKOTA.

NLPQLP: Methods for gradient-based constrained and unconstrained optimization problems using a sequential quadratic programming (SQP) algorithm [99]. NLPQLP is a commercial software product of Prof. Klaus Schittkowski (web site: `http://www.uni-bayreuth.de/departments/math/~kschittkowski/nlpqlp20.htm`). *Users may obtain their own copy of NLPQLP and compile it with the DAKOTA source code by following the steps given in the file* `Dakota/INSTALL`.

NPSOL: Methods for gradient-based constrained and unconstrained optimization problems using a sequential quadratic programming (SQP) algorithm [45]. NPSOL is a commercial software product of Stanford University (web site: www.sbsi-sol-optimize.com). Sandia National Laboratories, Lawrence Livermore National Laboratory, and Los Alamos National Laboratory all have site licenses for NPSOL. *Other users may obtain their own copy of NPSOL and compile it with the DAKOTA source code by following the steps given in the file* `Dakota/INSTALL`.

OPT++: Methods for gradient-based and nongradient-based optimization of unconstrained, bound-constrained,

and nonlinearly constrained optimization problems [81]. OPT++ includes a variety of Newton-based methods (quasi-Newton, finite-difference Newton, Gauss-Newton, and full-Newton), as well as the Polak-Ribeire CG method and the parallel direct search (PDS) method. OPT++ now contains a nonlinear interior point algorithm for handling general constraints. OPT++ is available to the public under the GNU LGPL and the source code is included with DAKOTA (web page: `http://csmr.ca.sandia.gov/opt++`).

Additional information on these methods is provided in Chapter 7, as is a facility for using other solvers made available to DAKOTA via shared libraries.

3.6 Additional Optimization Capabilities

The optimization software packages described above provide algorithms to handle a wide variety of optimization problems. This includes algorithms for constrained and unconstrained optimization, as well as algorithms for gradient-based and nongradient-based optimization. Listed below are additional optimization capabilities that are available in DAKOTA.

Multiobjective Optimization: There are three capabilities for multiobjective optimization in DAKOTA. First, there is the MOGA capability described previously in Section 3.5. This is a specialized algorithm capability. The second capability involves the use of response data transformations to recast a multiobjective problem as a single-objective problem. Currently, DAKOTA supports the weighting factor approach for this transformation, in which a composite objective function is constructed from a set of individual objective functions using a user-specified set of weighting factors. This approach is optimization algorithm independent, in that it works with any of the optimization methods listed in Section 3.5. Constraints are not affected by the weighting factor mapping; therefore, both constrained and unconstrained multiobjective optimization problems can be formulated and solved with DAKOTA, assuming selection of an appropriate constrained or unconstrained single-objective optimization algorithm. Future multiobjective response data transformations for goal programming, normal boundary intersection, etc. are planned. The third capability is the Pareto-set optimization strategy described in Section 3.9. This capability also utilizes the multiobjective response data transformations to allow optimization algorithm independence; however, it builds upon the basic approach by computing sets of optima in order to generate a Pareto trade-off surface.

User-Specified or Automatic Scaling: Some optimization algorithms are sensitive to the relative scaling of problem inputs and outputs. With any optimizer or least squares solver, user-specified (and in some cases automatic or logarithmic) scaling may be applied to continuous design variables, responses (objectives or residuals), nonlinear inequality and equality constraints, and/or linear inequality and equality constraints.

Additional information on these capabilities is provided in Chapter 7.

3.7 Nonlinear Least Squares for Parameter Estimation

Nonlinear least squares methods are optimization algorithms which exploit the special structure of a least squares objective function (see Section 1.4.2). These problems commonly arise in parameter estimation and test/analysis reconciliation. In practice, least squares solvers will tend to converge more rapidly than general-purpose optimization algorithms when the residual terms in the least squares formulation tend towards zero at the solution. Least squares solvers may experience difficulty when the residuals at the solution are significant, although experience has shown that the NL2SOL method can handle some problems that are highly nonlinear and have nonzero residuals at the solution.

NL2SOL: The NL2SOL algorithm [17] uses a secant-based algorithm to solve least-squares problems. In prac-

tice, it is more robust to nonlinear functions and nonzero residuals than conventional Gauss-Newton algorithms.

Gauss-Newton: DAKOTA's Gauss-Newton algorithm utilizes the Hessian approximation described in Section 1.4.2. The exact objective function value, exact objective function gradient, and the approximate objective function Hessian are defined from the least squares term values and gradients and are passed to the full-Newton optimizer from the OPT++ software package. As for all of the Newton-based optimization algorithms in OPT++, unconstrained, bound-constrained, and generally-constrained problems are supported. However, for the generally-constrained case, a derivative order mismatch exists in that the nonlinear interior point full Newton algorithm will require second-order information for the nonlinear constraints whereas the Gauss-Newton approximation only requires first order information for the least squares terms.

NLSSOL: The NLSSOL algorithm is a commercial software product of Stanford University (web site: `http://www.sbsi-sol-optimize.com`) that is bundled with current versions of the NPSOL library. It uses an SQP-based approach to solve generally-constrained nonlinear least squares problems. It periodically employs the Gauss-Newton Hessian approximation to accelerate the search. It requires only first-order information for the least squares terms and nonlinear constraints. Sandia National Laboratories, Lawrence Livermore National Laboratory, and Los Alamos National Laboratory all have site licenses for NLSSOL. *Other users may obtain their own copy of NLSSOL and compile it with the DAKOTA source code by following the NPSOL installation steps given in the file* `Dakota/INSTALL`.

Additional information on these methods is provided in Chapter 8.

3.8 Surrogate-Based Minimization

Surrogate-Based Local Minimization: This method combines the design of experiments methods, surrogate models, and optimization capabilities of DAKOTA. In SBO, the optimization algorithm operates on a surrogate model instead of directly operating on the computationally expensive simulation model. The surrogate model can be formed from data fitting methods (local, multipoint, or global), from a lower fidelity version of the computational model, or from a mathematically-generated reduced-order model (see Section 3.10). For each of these surrogate model types, the SBO algorithm periodically validates the progress using the surrogate model against the original high-fidelity model. The SBO strategy in DAKOTA can be configured to employ heuristic rules (less expensive) or to be provably convergent to the optimum of the original model (more expensive). The development of SBO strategies is an area of active research in the DAKOTA project.

Surrogate-Based Global Minimization: Similar to surrogate-based local minimization, this method combines design of experiments and surrogate modeling with optimization. However, rather than employing trust region model management to localize and control the extent of the approximation in order to ensure convergence to a local minimum, the surrogate-based global method sequentially refines the full range of a global approximation using global optimizers.

Efficient Global Minimization: Methods for nongradient-based constrained and unconstrained optimization and nonlinear least squares based on Gaussian process models. This approach uses an expected improvement function derived from the expected value and variance estimators in Gaussian process models, and is designed to balance exploitation of regions with good solutions and exploration of regions with limited data. [70]

Additional information on these methods is provided in Chapter 9.

3.9 Optimization Strategies

Due to the flexibility of DAKOTA's object-oriented design, it is relatively easy to create algorithms that combine several of DAKOTA's capabilities. These algorithms are referred to as *strategies*:

Multilevel Hybrid Optimization: This strategy allows the user to specify a sequence of optimization methods, with the results from one method providing the starting point for the next method in the sequence. An example which is useful in many engineering design problems involves the use of a nongradient-based global optimization method (e.g., genetic algorithm) to identify a promising region of the parameter space, which feeds its results into a gradient-based method (quasi-Newton, SQP, etc.) to perform an efficient local search for the optimum point.

Multistart Local Optimization: This strategy uses many local optimization runs (often gradient-based), each of which is started from a different initial point in the parameter space. This is an attractive strategy in situations where multiple local optima are known to exist or may potentially exist in the parameter space. This approach combines the efficiency of local optimization methods with the parameter space coverage of a global stratification technique.

Pareto-Set Optimization: The Pareto-set optimization strategy allows the user to specify different sets of weights for the individual objective functions in a multiobjective optimization problem. DAKOTA executes each of these weighting sets as a separate optimization problem, serially or in parallel, and then outputs the set of optimal designs which define the Pareto set. Pareto set information can be useful in making trade-off decisions in engineering design problems. *[Refer to 3.6 for additional information on multiobjective optimization methods.]*

These strategies are covered in more detail in Chapter 16.

3.10 Surrogate Models

Surrogate models are inexpensive approximate models that are intended to capture the salient features of an expensive high-fidelity model. They can be used to explore the variations in response quantities over regions of the parameter space, or they can serve as inexpensive stand-ins for optimization or uncertainty quantification studies (see, for example, the surrogate-based optimization strategy in Section 3.9). The surrogate models supported in DAKOTA can be categorized into three types: data fits, multifidelity, and reduced-order model surrogates.

Data fitting methods involve construction of an approximation or surrogate model using data (response values, gradients, and Hessians) generated from the original truth model. Data fit methods can be further categorized as local, multipoint, and global approximation techniques, based on the number of points used in generating the data fit. Local methods involve response data from a single point in parameter space. Available techniques currently include:

Taylor Series Expansion: This is a local first-order or second-order expansion centered at a single point in the parameter space.

Multipoint approximations involve response data from two or more points in parameter space, often involving the current and previous iterates of a minimization algorithm. Available techniques currently include:

TANA-3: This multipoint approximation uses a two-point exponential approximation [127, 35] built with response value and gradient information from the current and previous iterates.

Global methods, often referred to as *response surface methods*, involve many points spread over the parameter ranges of interest. These surface fitting methods work in conjunction with the sampling methods and design of experiments methods described in Section 3.3.

Polynomial Regression: First-order (linear), second-order (quadratic), and third-order (cubic) polynomial re-

sponse surfaces computed using linear least squares regression methods. Note: there is currently no use of forward- or backward-stepping regression methods to eliminate unnecessary terms from the polynomial model.

Gaussian Process (GP) or Kriging Interpolation DAKOTA contains two implementations of Gaussian process, also known as Kriging [52], spatial interpolation. One of these resides in the Surfpack sub-package of DAKOTA, the other resides in DAKOTA itself. Both versions use the Gaussian correlation function with parameters that are selected by Maximum Likelihood Estimation (MLE). This correlation function results in a response surface that is C^∞-continuous. Prior to DAKOTA 5.2, the Surfpack GP was referred to as the "Kriging" model and the DAKOTA version was labeled as the "Gaussian Process." These terms are now used interchangeably. As of DAKOTA 5.2,the Surfpack GP is used by default. For now the user still has the option to select the DAKOTA GP, but the DAKOTA GP is deprecated and will be removed in a future release.

- **Surfpack GP**: Ill-conditioning due to a poorly spaced sample design is handled by discarding points that contribute the least unique information to the correlation matrix. Therefore, the points that are discarded are the ones that are easiest to predict. The resulting surface will exactly interpolate the data values at the retained points but is not guaranteed to interpolate the discarded points.

- **DAKOTA GP**: Ill-conditioning is handled by adding a jitter term or "nugget" to diagonal elements of the correlation matrix. When this happens, the DAKOTA GP may not exactly interpolate the data values.

Artificial Neural Networks: An implementation of the stochastic layered perceptron neural network developed by Prof. D. C. Zimmerman of the University of Houston [128]. This neural network method is intended to have a lower training (fitting) cost than typical back-propagation neural networks.

Multivariate Adaptive Regression Splines (MARS): Software developed by Prof. J. H. Friedman of Stanford University [39]. The MARS method creates a C^2-continuous patchwork of splines in the parameter space.

Radial Basis Functions (RBF): Radial basis functions are functions whose value typically depends on the distance from a center point, called the centroid. The surrogate model approximation is constructed as the weighted sum of individual radial basis functions.

Moving Least Squares (MLS): Moving Least Squares can be considered a more specialized version of linear regression models. MLS is a weighted least squares approach where the weighting is "moved" or recalculated for every new point where a prediction is desired. [85]

In addition to data fit surrogates, DAKOTA supports multifidelity and reduced-order model approximations:

Multifidelity Surrogates: Multifidelity modeling involves the use of a low-fidelity physics-based model as a surrogate for the original high-fidelity model. The low-fidelity model typically involves a coarser mesh, looser convergence tolerances, reduced element order, or omitted physics. It is a separate model in its own right and does not require data from the high-fidelity model for construction. Rather, the primary need for high-fidelity evaluations is for defining correction functions that are applied to the low-fidelity results.

Reduced Order Models: A reduced-order model (ROM) is mathematically derived from a high-fidelity model using the technique of Galerkin projection. By computing a set of basis functions (e.g., eigenmodes, left singular vectors) that capture the principal dynamics of a system, the original high-order system can be projected to a much smaller system, of the size of the number of retained basis functions.

Additional information on these surrogate methods is provided in Sections 10.4.1 through 10.4.3.

3.11 Nested Models

Nested models utilize a sub-iterator and a sub-model to perform a complete iterative study as part of every evaluation of the model. This sub-iteration accepts variables from the outer level, performs the sub-level analysis, and computes a set of sub-level responses which are passed back up to the outer level. The nested model constructs admit a wide variety of multi-iterator, multi-model solution approaches. For example, optimization within optimization (for hierarchical multidisciplinary optimization), uncertainty quantification within uncertainty quantification (for second-order probability), uncertainty quantification within optimization (for optimization under uncertainty), and optimization within uncertainty quantification (for uncertainty of optima) are all supported, with and without surrogate model indirection. Three important examples are highlighted: mixed epistemic-aleatory uncertainty quantification, optimization under uncertainty, and surrogate-based uncertainty quantification.

Mixed Epistemic-Aleatory Uncertainty Quantification: Mixed uncertainty quantification (UQ) refers to capabilities for performing UQ calculations on both epistemic uncertainties (also known as reducible uncertainties resulting from a lack of knowledge) and aleatory uncertainties (also known as irreducible uncertainties that are inherent variabilities). Mixed UQ approaches employ nested models to embed one uncertainty quantification within another. The outer level UQ is commonly linked to epistemic uncertainties, and the inner UQ is commonly linked to aleatory uncertainties. We have three main approaches: interval-valued probability, second-order probability, and nested Dempster-Shafer. In interval-valued probability, the outer level generates sets of realizations, typically from sampling within interval distributions. These realizations define values for the epistemic variables used in a probabilistic analysis for the inner level UQ (e.g. which may involve sampling over aleatory variables). In interval-valued probability, we generate intervals on statistics from the inner loop. In second-order probability, the outer level also generates sets of realizations, from sampling from distributions on the epistemic variables. These outer loop values then define values for distribution parameters used in the inner level UQ. The term "second-order" derives from this use of distributions on distributions and the generation of statistics on statistics. DAKOTA includes capability to use interval analysis to perform the outer loop calculations (e.g. find intervals on inner loop statistics). Interval analysis can use efficient optimization methods to obtain interval bound estimates. Nested Dempster-Shafer refers to using Dempster-Shafer evidence theory on the outer loop, and an aleatory UQ method such as sampling or stochastic expansions on the inner loop. Evidence theory results in measures of belief and plausibility, so in a nested context, this produces belief and plausibility bounds on inner loop statistics. More on mixed UQ approaches can be found in [32] and [33].

Optimization Under Uncertainty (OUU): Many real-world engineering design problems contain stochastic features and must be treated using OUU methods such as robust design and reliability-based design. For OUU, the uncertainty quantification methods of DAKOTA are combined with optimization algorithms. This allows the user to formulate problems where one or more of the objective and constraints are stochastic. Due to the computational expense of both optimization and UQ, the simple nesting of these methods in OUU can be computationally prohibitive for real-world design problems. For this reason, surrogate-based optimization under uncertainty (SBOUU), reliability-based design optimization (RBDO), polynomial chaos-based design optimization (PCBDO), and stochastic collocation-based design optimization (SCBDO) methods have been developed which can reduce the overall expense by orders of magnitude. OUU methods are an active research area.

Surrogate-Based Uncertainty Quantification (SBUQ): Since many uncertainty quantification (UQ) methods are computationally costly, requiring many function evaluations to obtain accurate estimates of moments or percentile values of an output distribution, one may wish to embed surrogate models within the UQ process in order to reduce expense. By evaluating the true function on a fixed, small set of samples and using these sample evaluations to create a response surface approximation (e.g. a surrogate model or meta-model) of the underlying "true" function, the subsequent evaluation of the UQ results (using thousands or millions of samples) based on the approximation can obtain estimates of the mean, variance, and percentiles of the response at much lower overall cost.

Additional information on these nested approaches is provided in Section 10.5 and Chapter 17.

3.12 Parallel Computing

The methods and strategies in DAKOTA are designed to exploit parallel computing resources such as those found in a desktop multiprocessor workstation, a network of workstations, or a massively parallel computing platform. This parallel computing capability is a critical technology for rendering real-world engineering design problems computationally tractable. DAKOTA employs the concept of *multilevel parallelism*, which takes simultaneous advantage of opportunities for parallel execution from multiple sources:

Parallel Simulation Codes: DAKOTA works equally well with both serial and parallel simulation codes.

Concurrent Execution of Analyses within a Function Evaluation: Some engineering design applications call for the use of multiple simulation code executions (different disciplinary codes, the same code for different load cases or environments, etc.) in order to evaluate a single response data set (e.g., objective functions and constraints) for a single set of parameters. If these simulation code executions are independent (or if coupling is enforced at a higher level), DAKOTA can perform them in parallel.

Concurrent Execution of Function Evaluations within an Iterator: With very few exceptions, the iterative algorithms described in Section 3.2 through Section 3.7 all provide opportunities for the concurrent evaluation of response data sets for different parameter sets. Whenever there exists a set of design point evaluations that are independent, DAKOTA can perform them in parallel.

Concurrent Execution of Iterators within a Strategy: Some of the DAKOTA strategies described in Section 3.9 generate a sequence of iterator subproblems. For example, the Pareto-set and multi-start strategies generate sets of optimization subproblems, and the optimization under uncertainty strategy generates sets of uncertainty quantification subproblems. Whenever these subproblems are independent, DAKOTA can perform them in parallel.

It is important to recognize that these four parallelism levels are nested, in that a strategy can schedule and manage concurrent iterators, each of which may manage concurrent function evaluations, each of which may manage concurrent analyses, each of which may execute on multiple processors. Additional information on parallel computing with DAKOTA is provided in Chapter 19.

3.13 Summary

DAKOTA is both a production tool for engineering design and analysis activities and a research tool for the development of new algorithms in optimization, uncertainty quantification, and related areas. Because of the extensible, object-oriented design of DAKOTA, it is relatively easy to add new iterative algorithms, strategies, simulation interfacing approaches, surface fitting methods, etc. In addition, DAKOTA can serve as a rapid prototyping tool for algorithm development. That is, by having a broad range of building blocks available (i.e., parallel computing, surrogate models, simulation interfaces, fundamental algorithms, etc.), new capabilities can be assembled rapidly which leverage the previous software investments. For additional discussion on framework extensibility, refer to the DAKOTA Developers Manual [4].

The capabilities of DAKOTA have been used to solve engineering design and optimization problems at Sandia Labs, at other Department of Energy labs, and by our industrial and academic collaborators. Often, this real-world experience has provided motivation for research into new areas of optimization. The DAKOTA development team welcomes feedback on the capabilities of this software toolkit, as well as suggestions for new areas of research.

Chapter 4

Parameter Study Capabilities

4.1 Overview

Parameter study methods in the DAKOTA toolkit involve the computation of response data sets at a selection of points in the parameter space. These response data sets are not linked to any specific interpretation, so they may consist of any allowable specification from the responses keyword block, i.e., objective and constraint functions, least squares terms and constraints, or generic response functions. This allows the use of parameter studies in direct coordination with optimization, least squares, and uncertainty quantification studies without significant modification to the input file. In addition, response data sets are not restricted to function values only; gradients and Hessians of the response functions can also be catalogued by the parameter study. This allows for several different approaches to "sensitivity analysis": (1) the variation of function values over parameter ranges provides a global assessment as to the sensitivity of the functions to the parameters, (2) derivative information can be computed numerically, provided analytically by the simulator, or both (mixed gradients) in directly determining local sensitivity information at a point in parameter space, and (3) the global and local assessments can be combined to investigate the variation of derivative quantities through the parameter space by computing sensitivity information at multiple points.

In addition to sensitivity analysis applications, parameter studies can be used for investigating nonsmoothness in simulation response variations (so that models can be refined or finite difference step sizes can be selected for computing numerical gradients), interrogating problem areas in the parameter space, or performing simulation code verification (verifying simulation robustness) through parameter ranges of interest. A parameter study can also be used in coordination with minimization methods as either a pre-processor (to identify a good starting point) or a post-processor (for post-optimality analysis).

Parameter study methods will iterate any combination of design, uncertain, and state variables defined over continuous and discrete domains into any set of responses (any function, gradient, and Hessian definition). Parameter studies draw no distinction among the different types of continuous variables (design, uncertain, or state) or among the different types of response functions. They simply pass all of the variables defined in the variables specification into the interface, from which they expect to retrieve all of the responses defined in the responses specification. As described in Section 13.3, when gradient and/or Hessian information is being catalogued in the parameter study, it is assumed that derivative components will be computed with respect to all of the *continuous* variables (continuous design, continuous uncertain, and continuous state variables) specified, since derivatives with respect to discrete variables are assumed to be undefined.

DAKOTA currently supports four types of parameter studies. Vector parameter studies compute response data

sets at selected intervals along an n-dimensional vector in parameter space. List parameter studies compute response data sets at a list of points in parameter space, defined by the user. A centered parameter study computes multiple coordinate-based parameter studies, one per parameter, centered about the initial parameter values. A multidimensional parameter study computes response data sets for an n-dimensional hypergrid of points. More detail on these parameter studies is found in Sections 4.2 through 4.5 below.

4.1.1 Initial Values

The vector and centered parameter studies use the initial values of the variables from the `variables` keyword block as the starting point and the central point of the parameter studies, respectively. In the case of design variables, the `initial_point` is used, and in the case of state variables, the `initial_state` is used (see the DAKOTA Reference Manual [3] for default values when `initial_point` or `initial_state` are unspecified). In the case of uncertain variables, initial values are inferred from the distribution specification: all uncertain initial values are set to their means, where mean values for bounded normal and bounded lognormal are repaired of needed to satisfy the specified distribution bounds, mean values for discrete integer range distributions are rounded down to the nearest integer, and mean values for discrete set distributions are rounded to the nearest set value. These parameter study starting values for design, uncertain, and state variables are referenced in the following sections using the identifier "Initial Values."

4.1.2 Bounds

The multidimensional parameter study uses the bounds of the variables from the `variables` keyword block to define the range of parameter values to study. In the case of design and state variables, the `lower_bounds` and `upper_bounds` specifications are used (see the DAKOTA Reference Manual [3] for default values when `lower_bounds` or `upper_bounds` are unspecified). In the case of uncertain variables, these values are either drawn or inferred from the distribution specification. Distribution lower and upper bounds can be drawn directly from required bounds specifications for uniform, loguniform, triangular, and beta distributions, as well as from optional bounds specifications for normal and lognormal. Distribution bounds are implicitly defined for histogram bin, histogram point, and interval variables (from the extreme values within the bin/point/interval specifications) as well as for binomial (0 to `num_trials`) and hypergeometric (0 to min(`num_drawn`,`num_selected`)) variables. Finally, distribution bounds are inferred for normal and lognormal if optional bounds are unspecified, as well as for exponential, gamma, gumbel, frechet, weibull, poisson, negative binomial, and geometric (which have no bounds specifications); these bounds are [0, $\mu+3\sigma$] for exponential, gamma, frechet, weibull, poisson, negative binomial, geometric, and unspecified lognormal, and [$\mu - 3\sigma$, $\mu + 3\sigma$] for gumbel and unspecified normal.

4.2 Vector Parameter Study

The vector parameter study computes response data sets at selected intervals along an n-dimensional vector in parameter space. This capability encompasses both single-coordinate parameter studies (to study the effect of a single variable on a response set) as well as multiple coordinate vector studies (to investigate the response variations along some arbitrary vector; e.g., to investigate a search direction failure).

DAKOTA's vector parameter study includes two possible specification formulations which are used in conjunction with the Initial Values (see Section 4.1.1) to define the vector and steps of the parameter study:

```
final_point (vector of reals) and num_steps (integer)
step_vector (vector of reals) and num_steps (integer)
```

In both of these cases, the Initial Values are used as the parameter study starting point and the specification selection above defines the orientation of the vector and the increments to be evaluated along the vector. In the former case, the vector from initial to final point is partitioned by `num_steps`, and in the latter case, the `step_vector` is added `num_steps` times. In the case of discrete range variables, both `final_point` and `step_vector` are specified in the actual values; and in the case of discrete sets (integer or real), `final_point` is specified in the actual values but `step_vector` must instead specify index offsets for the (ordered, unique) set. In all cases, the number of evaluations is `num_steps`+1. Two examples are included below:

Three continuous parameters with initial values of (1.0, 1.0, 1.0), `num_steps` = 4, and either `final_point` = (1.0, 2.0, 1.0) or `step_vector` = (0, .25, 0):

```
Parameters for function evaluation 1:
                     1.0000000000e+00 c1
                     1.0000000000e+00 c2
                     1.0000000000e+00 c3
Parameters for function evaluation 2:
                     1.0000000000e+00 c1
                     1.2500000000e+00 c2
                     1.0000000000e+00 c3
Parameters for function evaluation 3:
                     1.0000000000e+00 c1
                     1.5000000000e+00 c2
                     1.0000000000e+00 c3
Parameters for function evaluation 4:
                     1.0000000000e+00 c1
                     1.7500000000e+00 c2
                     1.0000000000e+00 c3
Parameters for function evaluation 5:
                     1.0000000000e+00 c1
                     2.0000000000e+00 c2
                     1.0000000000e+00 c3
```

Two continuous parameters with initial values of (1.0, 1.0), one discrete range parameter with initial value of 5, one discrete real set parameter with set values of (10., 12., 18., 30., 50.) and initial value of 10., `num_steps` = 4, and either `final_point` = (2.0, 1.4, 13, 50.) or `step_vector` = (.25, .1, 2, 1):

```
Parameters for function evaluation 1:
                     1.0000000000e+00 c1
                     1.0000000000e+00 c2
                                    5 di1
                     1.0000000000e+01 dr1
Parameters for function evaluation 2:
                     1.2500000000e+00 c1
                     1.1000000000e+00 c2
                                    7 di1
                     1.2000000000e+01 dr1
Parameters for function evaluation 3:
                     1.5000000000e+00 c1
                     1.2000000000e+00 c2
                                    9 di1
                     1.8000000000e+01 dr1
Parameters for function evaluation 4:
                     1.7500000000e+00 c1
                     1.3000000000e+00 c2
```

```
                                    11 di1
                     3.0000000000e+01 dr1
Parameters for function evaluation 5:
                     2.0000000000e+00 c1
                     1.4000000000e+00 c2
                                    13 di1
                     5.0000000000e+01 dr1
```

4.3 List Parameter Study

The list parameter study computes response data sets at selected points in parameter space. These points are explicitly specified by the user and are not confined to lie on any line or surface. Thus, this parameter study provides a general facility that supports the case where the desired set of points to evaluate does not fit the prescribed structure of the vector, centered, or multidimensional parameter studies.

The user input consists of a `list_of_points` specification which lists the requested parameter sets in succession. The list parameter study simply performs a simulation for the first parameter set (the first n entries in the list), followed by a simulation for the next parameter set (the next n entries), and so on, until the list of points has been exhausted. Since the Initial Values will not be used, they need not be specified. In the case of discrete range or discrete set variables, list values are specified using the actual values (not set indices).

An example specification that would result in the same parameter sets as in the second example in Section 4.2 would be:

```
list_of_points = 1.0  1.0  5 10.
                 1.25 1.1  7 12.
                 1.5  1.2  9 18.
                 1.75 1.3 11 30.
                 2.0  1.4 13 50.
```

4.4 Centered Parameter Study

The centered parameter study executes multiple coordinate-based parameter studies, one per parameter, centered about the specified Initial Values. This is useful for investigation of function contours in the vicinity of a specific point. For example, after computing an optimum design, this capability could be used for post-optimality analysis in verifying that the computed solution is actually at a minimum or constraint boundary and in investigating the shape of this minimum or constraint boundary.

This method requires `step_vector` (list of reals) and `steps_per_variable` (list of integers) specifications, where the former specifies the size of the increments per variable (employed sequentially, not all at once as for the vector study in Section 4.2) and the latter specifies the number of increments per variable (employed sequentially, not all at once) for each of the positive and negative step directions. As for the vector study described in Section 4.2, `step_vector` includes actual variable steps for continuous and discrete range variables, but employs index offsets for discrete set variables (integer or real).

For example, with Initial Values of (1.0, 1.0), a `step_vector` of (0.1, 0.1), and a `steps_per_variable` of (2, 2), the center point is evaluated followed by four function evaluations (two negative deltas and two positive deltas) per variable:

```
Parameters for function evaluation 1:
```

Figure 4.1: Example centered parameter study.

```
                     1.0000000000e+00 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 2:
                     8.0000000000e-01 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 3:
                     9.0000000000e-01 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 4:
                     1.1000000000e+00 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 5:
                     1.2000000000e+00 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 6:
                     1.0000000000e+00 d1
                     8.0000000000e-01 d2
Parameters for function evaluation 7:
                     1.0000000000e+00 d1
                     9.0000000000e-01 d2
Parameters for function evaluation 8:
                     1.0000000000e+00 d1
                     1.1000000000e+00 d2
Parameters for function evaluation 9:
                     1.0000000000e+00 d1
                     1.2000000000e+00 d2
```

This set of points in parameter space is depicted in Figure 4.1.

Figure 4.2: Example multidimensional parameter study

4.5 Multidimensional Parameter Study

The multidimensional parameter study computes response data sets for an n-dimensional hypergrid of points. Each variable is partitioned into equally spaced intervals between its upper and lower bounds (see Section 4.1.2), and each combination of the values defined by these partitions is evaluated. As for the vector and centered studies described in Sections 4.2 and 4.4, partitioning occurs using the actual variable values for continuous and discrete range variables, but occurs within the space of valid indices for discrete set variables (integer or real). The number of function evaluations performed in the study is:

$$\prod_{i=1}^{n}(\texttt{partitions}_i + 1) \tag{4.1}$$

The partitions information is specified using the `partitions` specification, which provides an integer list of the number of partitions for each variable (i.e., $\texttt{partitions}_i$). Since the Initial Values will not be used, they need not be specified.

In a two variable example problem with `d1` $\in$ [0,2] and `d2` $\in$ [0,3] (as defined by the upper and lower bounds from the variables specification) and with `partitions` = (2,3), the interval [0,2] is divided into two equal-sized partitions and the interval [0,3] is divided into three equal-sized partitions. This two-dimensional grid, shown in Figure 4.2, would result in the following twelve function evaluations:

```
Parameters for function evaluation 1:
```

```
                     0.0000000000e+00 d1
                     0.0000000000e+00 d2
Parameters for function evaluation 2:
                     1.0000000000e+00 d1
                     0.0000000000e+00 d2
Parameters for function evaluation 3:
                     2.0000000000e+00 d1
                     0.0000000000e+00 d2
Parameters for function evaluation 4:
                     0.0000000000e+00 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 5:
                     1.0000000000e+00 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 6:
                     2.0000000000e+00 d1
                     1.0000000000e+00 d2
Parameters for function evaluation 7:
                     0.0000000000e+00 d1
                     2.0000000000e+00 d2
Parameters for function evaluation 8:
                     1.0000000000e+00 d1
                     2.0000000000e+00 d2
Parameters for function evaluation 9:
                     2.0000000000e+00 d1
                     2.0000000000e+00 d2
Parameters for function evaluation 10:
                     0.0000000000e+00 d1
                     3.0000000000e+00 d2
Parameters for function evaluation 11:
                     1.0000000000e+00 d1
                     3.0000000000e+00 d2
Parameters for function evaluation 12:
                     2.0000000000e+00 d1
                     3.0000000000e+00 d2
```

Chapter 5

Design of Experiments Capabilities

5.1 Overview

DAKOTA contains several software packages for sampling and design of experiments: LHS (Latin hypercube sampling), DDACE (distributed design and analysis for computer experiments), FSUDace (Florida State University's Design and Analysis of Computer Experiments package), and PSUADE (Problem Solving Environment for Uncertainty Analysis and Design Exploration). LHS [109] is a general-purpose sampling package developed at Sandia that has been used by the DOE national labs for several decades. DDACE is a more recent package for computer experiments that is under development by staff at Sandia Labs [114]. DDACE provides the capability for generating orthogonal arrays, Box-Behnken designs, Central Composite designs, and random designs. The FSUDace package provides the following sampling techniques: quasi-Monte Carlo sampling based on Halton or Hammersley sequences, and Centroidal Voronoi Tessellation. Lawrence Livermore National Lab's PSUADE [113] includes several methods for model exploration, but only the Morris screening method is exposed in DAKOTA.

This chapter describes DDACE, FSUDace, and PSUADE, with the primary goal of designing computer experiments. Latin Hypercube Sampling, also used in uncertainty quantification, is discussed in Section 6.2. The differences between sampling used in design of experiments and sampling used in uncertainty quantification is discussed in more detail in the following paragraphs. In brief, we consider design of experiment methods to generate sets of uniform random variables on the interval $[0, 1]$. These sets are mapped to the lower/upper bounds of the problem variables and then the response functions are evaluated at the sample input points with the goal of characterizing the behavior of the response functions over the input parameter ranges of interest. Uncertainty quantification via LHS sampling, in contrast, involves characterizing the uncertain input variables with probability distributions such as normal, Weibull, triangular, etc., sampling from the input distributions, and propagating the input uncertainties to obtain a cumulative distribution function on the output. There is significant overlap between design of experiments and sampling. Often, both techniques can be used to obtain similar results about the behavior of the response functions and about the relative importance of the input variables.

5.2 Design of Computer Experiments

Computer experiments are often different from physical experiments, such as those performed in agriculture, manufacturing, or biology. In physical experiments, one often applies the same *treatment* or *factor level* in an

experiment several times to get an understanding of the variability of the output when that treatment is applied. For example, in an agricultural experiment, several fields (e.g., 8) may be subject to a low level of fertilizer and the same number of fields may be subject to a high level of fertilizer to see if the amount of fertilizer has a significant effect on crop output. In addition, one is often interested in the variability of the output within a treatment group: is the variability of the crop yields in the low fertilizer group much higher than that in the high fertilizer group, or not?

In physical experiments, the process we are trying to examine is stochastic: that is, the same treatment may result in different outcomes. By contrast, in computer experiments, often we have a deterministic code. If we run the code with a particular set of input parameters, the code will always produce the same output. There certainly are stochastic codes, but the main focus of computer experimentation has been on deterministic codes. Thus, in computer experiments we often do not have the need to do replicates (running the code with the exact same input parameters several times to see differences in outputs). Instead, a major concern in computer experiments is to create an experimental design which can sample a high-dimensional space in a representative way with a minimum number of samples. The number of factors or parameters that we wish to explore in computer experiments is usually much higher than physical experiments. In physical experiments, one may be interested in varying a few parameters, usually five or less, while in computer experiments we often have dozens of parameters of interest. Choosing the levels of these parameters so that the samples adequately explore the input space is a challenging problem. There are many experimental designs and sampling methods which address the issue of adequate and representative sample selection. Classical experimental designs which are often used in physical experiments include Central Composite designs and Box-Behnken designs.

There are many goals of running a computer experiment: one may want to explore the input domain or the design space and get a better understanding of the range in the outputs for a particular domain. Another objective is to determine which inputs have the most influence on the output, or how changes in the inputs change the output. This is usually called *sensitivity analysis*. Another goal is to compare the relative importance of model input uncertainties on the uncertainty in the model outputs, *uncertainty analysis*. Yet another goal is to use the sampled input points and their corresponding output to create a *response surface approximation* for the computer code. The response surface approximation (e.g., a polynomial regression model, a Gaussian-process/Kriging model, a neural net) can then be used to emulate the computer code. Constructing a response surface approximation is particularly important for applications where running a computational model is extremely expensive: the computer model may take 10 or 20 hours to run on a high performance machine, whereas the response surface model may only take a few seconds. Thus, one often optimizes the response surface model or uses it within a framework such as surrogate-based optimization. Response surface models are also valuable in cases where the gradient (first derivative) and/or Hessian (second derivative) information required by optimization techniques are either not available, expensive to compute, or inaccurate because the derivatives are poorly approximated or the function evaluation is itself noisy due to roundoff errors. Furthermore, many optimization methods require a good initial point to ensure fast convergence or to converge to good solutions (e.g. for problems with multiple local minima). Under these circumstances, a good design of computer experiment framework coupled with response surface approximations can offer great advantages.

In addition to the sensitivity analysis, uncertainty analysis, and response surface modeling mentioned above, we also may want to do *uncertainty quantification* on a computer model. Uncertainty quantification (UQ) refers to taking a particular set of distributions on the inputs, and propagating them through the model to obtain a distribution on the outputs. For example, if input parameter A follows a normal with mean 5 and variance 1, the computer produces a random draw from that distribution. If input parameter B follows a weibull distribution with alpha = 0.5 and beta = 1, the computer produces a random draw from that distribution. When all of the uncertain variables have samples drawn from their input distributions, we run the model with the sampled values as inputs. We do this repeatedly to build up a distribution of outputs. We can then use the cumulative distribution function of the output to ask questions such as: what is the probability that the output is greater than 10? What is the 99th percentile of the output?

Note that sampling-based uncertainty quantification and design of computer experiments are very similar. *There is significant overlap* in the purpose and methods used for UQ and for DACE. We have attempted to delineate the differences within DAKOTA as follows: we use the methods DDACE, FSUDACE, and PSUADE primarily for design of experiments, where we are interested in understanding the main effects of parameters and where we want to sample over an input domain to obtain values for constructing a response surface. We use the nondeterministic sampling methods `(sampling)` for uncertainty quantification, where we are propagating specific input distributions and interested in obtaining (for example) a cumulative distribution function on the output. If one has a problem with no distributional information, we recommend starting with a design of experiments approach. Note that DDACE, FSUDACE, and PSUADE currently do *not* support distributional information: they take an upper and lower bound for each uncertain input variable and sample within that. The uncertainty quantification methods in `sampling` (primarily Latin Hypercube sampling) offer the capability to sample from many distributional types. The distinction between UQ and DACE is somewhat arbitrary: both approaches often can yield insight about important parameters and both can determine sample points for response surface approximations.

5.3 DDACE Background

The DACE package includes both classical design of experiments methods [114] and stochastic sampling methods. The classical design of experiments methods in DDACE are central composite design (CCD) and Box-Behnken (BB) sampling. A grid-based sampling method also is available. The stochastic methods are orthogonal array sampling [73], Monte Carlo (random) sampling, and Latin hypercube sampling. Note that the DDACE version of LHS available through the DAKOTA interface only supports uniform distributions. DDACE does not currently support enforcement of user-specified correlation structure among the variables.

The sampling methods in DDACE can be used alone or in conjunction with other methods. For example, DDACE sampling can be used with both the surrogate-based optimization strategy and the optimization under uncertainty strategy. See Figure 17.5 for an example of how the DDACE settings are used in DAKOTA.

More information on DDACE is available on the web at: `http://csmr.ca.sandia.gov/projects/ddace`

The following sections provide more detail about the sampling methods available for design of experiments in DDACE.

5.3.1 Central Composite Design

A Box-Wilson Central Composite Design, commonly called a central composite design (CCD), contains an embedded factorial or fractional factorial design with center points that is augmented with a group of 'star points' that allow estimation of curvature. If the distance from the center of the design space to a factorial point is ± 1 unit for each factor, the distance from the center of the design space to a star point is $\pm\alpha$ with $\mid \alpha \mid > 1$. The precise value of α depends on certain properties desired for the design and on the number of factors involved. The CCD design is specified in DAKOTA with the method command `dace central_composite`.

As an example, with two input variables or factors, each having two levels, the factorial design is shown in Table 5.1 .

With a CCD, the design in Table 5.1 would be augmented with the following points shown in Table 5.2 if α = 1.3. These points define a circle around the original factorial design.

Note that the number of sample points specified in a CCD,`samples`, is a function of the number of variables in the problem:

Table 5.1: Simple Factorial Design

Input 1	Input 2
-1	-1
-1	+1
+1	-1
+1	+1

Table 5.2: Additional Points to make the factorial design a CCD

Input 1	Input 2
0	+1.3
0	-1.3
1.3	0
-1.3	0
0	0

$$samples = 1 + 2 * NumVar + 2^{NumVar}$$

5.3.2 Box-Behnken Design

The Box-Behnken design is similar to a Central Composite design, with some differences. The Box-Behnken design is a quadratic design in that it does not contain an embedded factorial or fractional factorial design. In this design the treatment combinations are at the midpoints of edges of the process space and at the center, as compared with CCD designs where the extra points are placed at 'star points' on a circle outside of the process space. Box-Behken designs are rotatable (or near rotatable) and require 3 levels of each factor. The designs have limited capability for orthogonal blocking compared to the central composite designs. Box-Behnken requires fewer runs than CCD for 3 factors, but this advantage goes away as the number of factors increases. The Box-Behnken design is specified in DAKOTA with the method command `dace box_behnken`.

Note that the number of sample points specified in a Box-Behnken design, `samples`, is a function of the number of variables in the problem:

$$samples = 1 + 4 * NumVar + (NumVar - 1)/2$$

5.3.3 Orthogonal Array Designs

Orthogonal array (OA) sampling is a widely used technique for running experiments and systematically testing factor effects [64]. An orthogonal array sample can be described as a 4-tuple (m, n, s, r), where m is the number of sample points, n is the number of input variables, s is the number of symbols, and r is the strength of the orthogonal array. The number of sample points, m, must be a multiple of the number of symbols, s. The number of symbols refers to the number of levels per input variable. The strength refers to the number of columns where we are guaranteed to see all the possibilities an equal number of times.

For example, Table 5.3 shows an orthogonal array of strength 2 for m = 8, with 7 variables:

Table 5.3: Orthogonal Array for Seven Variables

Input 1	Input 2	Input 3	Input 4	Input 5	Input 6	Input 7
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	0	1	0
1	1	0	0	1	1	0
1	1	0	1	0	0	1

If one picks any two columns, say the first and the third, note that each of the four possible rows we might see there, 0 0, 0 1, 1 0, 1 1, appears exactly the same number of times, twice in this case.

DDACE creates orthogonal arrays of strength 2. Further, the OAs generated by DDACE do not treat the factor levels as one fixed value (0 or 1 in the above example). Instead, once a level for a variable is determined in the array, DDACE samples a random variable from within that level. The orthogonal array design is specified in DAKOTA with the method command `dace oas`.

The orthogonal array method in DDACE is the only method that allows for the calculation of main effects, specified with the command `main_effects`. Main effects is a sensitivity analysis method which identifies the input variables that have the most influence on the output. In main effects, the idea is to look at the mean of the response function when variable A (for example) is at level 1 vs. when variable A is at level 2 or level 3. If these mean responses of the output are statistically significantly different at different levels of variable A, this is an indication that variable A has a significant effect on the response. The orthogonality of the columns is critical in performing main effects analysis, since the column orthogonality means that the effects of the other variables 'cancel out' when looking at the overall effect from one variable at its different levels. There are ways of developing orthogonal arrays to calculate higher order interactions, such as two-way interactions (what is the influence of Variable A * Variable B on the output?), but this is not available in DDACE currently. At present, one way interactions are supported in the calculation of orthogonal array main effects within DDACE. The main effects are presented as a series of ANOVA tables. For each objective function and constraint, the decomposition of variance of that objective or constraint is presented as a function of the input variables. The p-value in the ANOVA table is used to indicate if the input factor is significant. The p-value is the probability that you would have obtained samples more extreme than you did if the input factor has no effect on the response. For example, if you set a level of significance at 0.05 for your p-value, and the actual p-value is 0.03, then the input factor has a significant effect on the response.

5.3.4 Grid Design

In a grid design, a grid is placed over the input variable space. This is very similar to a multi-dimensional parameter study where the samples are taken over a set of partitions on each variable (see Section 4.5). The main difference is that in grid sampling, a small random perturbation is added to each sample value so that the grid points are not on a perfect grid. This is done to help capture certain features in the output such as periodic functions. A purely structured grid, with the samples exactly on the grid points, has the disadvantage of not being able to capture important features such as periodic functions with relatively high frequency (due to aliasing).

Adding a random perturbation to the grid samples helps remedy this problem.

Another disadvantage with grid sampling is that the number of sample points required depends exponentially on the input dimensions. In grid sampling, the number of samples is the number of symbols (grid partitions) raised to the number of variables. For example, if there are 2 variables, each with 5 partitions, the number of samples would be 5^2. In this case, doubling the number of variables squares the sample size. The grid design is specified in DAKOTA with the method command `dace grid`.

5.3.5 Monte Carlo Design

Monte Carlo designs simply involve pure Monte-Carlo random sampling from uniform distributions between the lower and upper bounds on each of the input variables. Monte Carlo designs, specified by `dace random`, are a way to generate a set of random samples over an input domain.

5.3.6 LHS Design

DDACE offers the capability to generate Latin Hypercube designs. For more information on Latin Hypercube sampling, see Section 6.2. Note that the version of LHS in DDACE generates uniform samples (uniform between the variable bounds). The version of LHS offered with nondeterministic sampling can generate LHS samples according to a number of distribution types, including normal, lognormal, weibull, beta, etc. To specify the DDACE version of LHS, use the method command `dace lhs`.

5.3.7 OA-LHS Design

DDACE offers a hybrid design which is combination of an orthogonal array and a Latin Hypercube sample. This design is specified with the method command `dace oa_lhs`. This design has the advantages of both orthogonality of the inputs as well as stratification of the samples (see [89]).

5.4 FSUDace Background

The FSUDace package includes quasi-Monte Carlo sampling methods (Halton and Hammersley sequences) and Centroidal Voronoi Tesselation sampling. All three methods generate sets of uniform random variables on the interval $[0, 1]$. The quasi-Monte Carlo and CVT methods are designed with the goal of low discrepancy. Discrepancy refers to the nonuniformity of the sample points within the unit hypercube. Low discrepancy sequences tend to cover the unit hypercube reasonably uniformly. Quasi-Monte Carlo methods produce low discrepancy sequences, especially if one is interested in the uniformity of projections of the point sets onto lower dimensional faces of the hypercube (usually 1-D: how well do the marginal distributions approximate a uniform?) CVT does very well volumetrically: it spaces the points fairly equally throughout the space, so that the points cover the region and are isotropically distributed with no directional bias in the point placement. There are various measures of volumetric uniformity which take into account the distances between pairs of points, regularity measures, etc. Note that CVT does not produce low-discrepancy sequences in lower dimensions, however: the lower-dimension (such as 1-D) projections of CVT can have high discrepancy.

The quasi-Monte Carlo sequences of Halton and Hammersley are deterministic sequences determined by a set of prime bases. A Halton design is specified in DAKOTA with the method command `fsu_quasi_mc halton`, and the Hammersley design is specified with the command `fsu_quasi_mc hammersley`. For more details

about the input specification, see the Reference Manual. CVT points tend to arrange themselves in a pattern of cells that are roughly the same shape. To produce CVT points, an almost arbitrary set of initial points is chosen, and then an internal set of iterations is carried out. These iterations repeatedly replace the current set of sample points by an estimate of the centroids of the corresponding Voronoi subregions [22]. A CVT design is specified in DAKOTA with the method command `fsu_cvt`.

The methods in FSUDace are useful for design of experiments because they provide good coverage of the input space, thus allowing global sensitivity analysis.

5.5 PSUADE MOAT Background

This section describes the Morris One-At-A-Time (MOAT) screening method as implemented in LLNL's PSUADE (Problem Solving Environment for Uncertainty Analysis and Design Exploration) [113]. DAKOTA includes a prototype interface to this DACE method, which is useful for global sensitivity analysis, including interaction studies.

The Morris One-At-A-Time method, originally proposed by M. D. Morris [82], is a screening method, designed to explore a computational model to distinguish between input variables that have negligible, linear and additive, or nonlinear or interaction effects on the output. The computer experiments performed consist of individually randomized designs which vary one input factor at a time to create a sample of its elementary effects.

With MOAT, each dimension of a $k-$dimensional input space is uniformly partitioned into p levels, creating a grid of p^k points $\mathbf{x} \in \mathbb{R}^k$ at which evaluations of the model $y(\mathbf{x})$ might take place. An elementary effect corresponding to input i is computed by a forward difference

$$d_i(\mathbf{x}) = \frac{y(\mathbf{x} + \Delta \mathbf{e}_i) - y(\mathbf{x})}{\Delta}, \tag{5.1}$$

where e_i is the ith coordinate vector, and the step Δ is typically taken to be large (this is not intended to be a local derivative approximation). In the present implementation of MOAT, for an input variable scaled to $[0, 1]$, $\Delta = \frac{p}{2(p-1)}$, so the step used to find elementary effects is slightly larger than half the input range.

The distribution of elementary effects d_i over the input space characterizes the effect of input i on the output of interest. After generating r samples from this distribution, their mean,

$$\mu_i = \frac{1}{r} \sum_{j=1}^{r} d_i^{(j)}, \tag{5.2}$$

modified mean

$$\mu_i^* = \frac{1}{r} \sum_{j=1}^{r} |d_i^{(j)}|, \tag{5.3}$$

(using absolute value) and standard deviation

$$\sigma_i = \sqrt{\frac{1}{r} \sum_{j=1}^{r} \left(d_i^{(j)} - \mu_i \right)^2} \tag{5.4}$$

are computed for each input i. The mean and modified mean give an indication of the overall effect of an input on the output. Standard deviation indicates nonlinear effects or interactions, since it is an indicator of elementary effects varying throughout the input space.

The MOAT method is selected with method keyword `psuade_moat` as shown in the sample DAKOTA input deck in Figure 5.1. The number of samples (`samples`) must be a positive integer multiple of (number of continuous design variables k + 1) and will be automatically adjusted if misspecified. The number of partitions (`partitions`) applies to each variable being studied and must be odd (the number of MOAT levels p per variable is partitions + 1, similar to DAKOTA multidimensional parameter studies). This will also be adjusted at runtime as necessary. Finite user-specified lower and upper bounds are required and will be scaled as needed by the method. For more information on use of MOAT sampling, see the Morris example in Section 23.13, or Saltelli, et al. [97].

```
strategy,
        single_method

method,
        psuade_moat
          samples = 84
          partitions = 3
          seed = 5

variables,
        continuous_design = 20
          lower_bounds =    0.0 0.0 0.0 0.0 0.0
                                0.0 0.0 0.0 0.0 0.0
                                0.0 0.0 0.0 0.0 0.0
                                0.0 0.0 0.0 0.0 0.0
          upper_bounds =    1.0 1.0 1.0 1.0 1.0
                                1.0 1.0 1.0 1.0 1.0
                                1.0 1.0 1.0 1.0 1.0
                                1.0 1.0 1.0 1.0 1.0

interface,
        system asynchronous evaluation_concurrency = 5
          analysis_driver = 'morris'

responses,
        num_objective_functions = 1
        no_gradients
        no_hessians
```

Figure 5.1: DAKOTA input file `dakota_psuade.in` to invoke the PSUADE MOAT method on the Morris test problem described in Section 23.13.

5.6 Sensitivity Analysis

Like parameter studies (see Chapter 4), the DACE techniques are useful for characterizing the behavior of the response functions of interest through the parameter ranges of interest. In addition to direct interrogation and visualization of the sampling results, a number of techniques have been developed for assessing the parameters which are most influential in the observed variability in the response functions. One example of this is the well-known technique of scatter plots, in which the set of samples is projected down and plotted against one parameter dimension, for each parameter in turn. Scatter plots with a uniformly distributed cloud of points indicate

parameters with little influence on the results, whereas scatter plots with a defined shape to the cloud indicate parameters which are more significant. Related techniques include analysis of variance (ANOVA) [83] and main effects analysis, in which the parameters which have the greatest influence on the results are identified from sampling results. Scatter plots and ANOVA may be accessed through import of DAKOTA tabular results (see Section 15.3) into external statistical analysis programs such as S-plus, Minitab, etc.

Running any of the design of experiments or sampling methods allows the user to save the results in a tabular data file, which then can be read into a spreadsheet or statistical package for further analysis. In addition, we have provided some functions to help determine the most important variables.

We take the definition of uncertainty analysis from [97]: "The study of how uncertainty in the output of a model can be apportioned to different sources of uncertainty in the model input."

As a default, DAKOTA provides correlation analyses when running LHS. Correlation tables are printed with the simple, partial, and rank correlations between inputs and outputs. These can be useful to get a quick sense of how correlated the inputs are to each other, and how correlated various outputs are to inputs. The correlation analyses are explained further in Chapter 6.2.

We also have the capability to calculate sensitivity indices through Variance-based Decomposition (VBD). Variance-based decomposition is a global sensitivity method that summarizes how the uncertainty in model output can be apportioned to uncertainty in individual input variables. VBD uses two primary measures, the main effect sensitivity index S_i and the total effect index T_i. The main effect sensitivity index corresponds to the fraction of the uncertainty in the output, Y, that can be attributed to input x_i alone. The total effects index corresponds to the fraction of the uncertainty in the output, Y, that can be attributed to input x_i and its interactions with other variables. The main effect sensitivity index compares the variance of the conditional expectation $Var_{x_i}[E(Y|x_i)]$ against the total variance $Var(Y)$. Formulas for the indices are:

$$S_i = \frac{Var_{x_i}[E(Y|x_i)]}{Var(Y)} \tag{5.5}$$

and

$$T_i = \frac{E(Var(Y|x_{-i}))}{Var(Y)} = \frac{Var(Y) - Var(E[Y|x_{-i}])}{Var(Y)} \tag{5.6}$$

where $Y = f(\mathbf{x})$ and $x_{-i} = (x_1, ..., x_{i-1}, x_{i+1}, ..., x_m)$.

The calculation of S_i and T_i requires the evaluation of m-dimensional integrals which are typically approximated by Monte-Carlo sampling. More details on the calculations and interpretation of the sensitivity indices can be found in [97]. In DAKOTA version 5.1, we have improved calculations for the calculation of the S_i and T_i indices when using sampling. The implementation details of these calculatiosn are provided in [124]. VBD can be specified for any of the sampling or DACE methods using the command `variance_based_decomposition`. Note that VBD is extremely computationally intensive when using sampling since replicated sets of sample values are evaluated. If the user specified a number of samples, N, and a number of nondeterministic variables, M, variance-based decomposition requires the evaluation of $N(M + 2)$ samples. To obtain sensitivity indices that are reasonably accurate, we recommend that N, the number of samples, be at least one hundred and preferably several hundred or thousands. Because of the computational cost, variance-based decomposition is turned off as a default for sampling or DACE. Another alternative, however, is to obtain these indices using one of the stochastic expansion methods described in Section 6.4. The calculation of the indices using expansion methods is much more efficient since the VBD indices are analytic functions of the coefficients in the stochastic expansion. The paper by Weirs et al. [124] compares different methods for calculating the sensitivity indices for nonlinear problems with significant interaction effects.

In terms of interpretation of the sensitivity indices, a larger value of the sensitivity index, S_i, means that the

uncertainty in the input variable i has a larger effect on the variance of the output. Note that the sum of the main effect indices will be less than or equal to one. If the sum of the main effect indices is much less than one, it indicates that there are significant two-way, three-way, or higher order interactions that contribute significantly to the variance. There is no requirement that the sum of the total effect indices is one: in most cases, the sum of the total effect indices will be greater than one. An example of the Main and Total effects indices as calculated by DAKOTA using sampling is shown in Figure 5.2

```
Global sensitivity indices for each response function:
response_fn_1 Sobol indices:
                                 Main             Total
                     4.7508913283e-01  5.3242162037e-01 uuv_1
                     3.8112392892e-01  4.9912486515e-01 uuv_2
```

Figure 5.2: DAKOTA output for Variance-based Decomposition

Finally, we have the capability to calculate a set of quality metrics for a particular input sample. These quality metrics measure various aspects relating to the volumetric spacing of the samples: are the points equally spaced, do they cover the region, are they isotropically distributed, do they have directional bias, etc.? The quality metrics are explained in more detail in the Reference Manual.

Chapter 6

Uncertainty Quantification Capabilities

6.1 Overview

Uncertainty quantification (UQ) is the process of determining the effect of input uncertainties on response metrics of interest. These input uncertainties may be characterized as either aleatory uncertainties, which are irreducible variabilities inherent in nature, or epistemic uncertainties, which are reducible uncertainties resulting from a lack of knowledge. Since sufficient data is generally available for aleatory uncertainties, probabilistic methods are commonly used for computing response distribution statistics based on input probability distribution specifications. Conversely, for epistemic uncertainties, data is generally sparse, making the use of probability distribution assertions questionable and typically leading to nonprobabilistic methods based on interval specifications.

DAKOTA contains capabilities for performing nondeterministic analysis. The methods for uncertainty quantification in DAKOTA have been developed by a group of researchers at Sandia Labs, in conjunction with collaborators in academia [43, 44, 24, 111]. These methods perform a forward uncertainty propagation in which probability or interval information for input parameters is mapped to probability or interval information for output response functions. The m functions in the DAKOTA response data set are interpreted as m general response functions by the DAKOTA methods (with no specific interpretation of the functions as for optimization and least squares).

Within the variables specification, uncertain variable descriptions are employed to define the parameter probability distributions (see Section 11.3). The continuous aleatory distribution types include: normal (Gaussian), lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, and histogram bin. The discrete aleatory distribution types include: poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point. The epistemic distribution type is interval. When gradient and/or Hessian information is used in an uncertainty assessment, derivative components are normally computed with respect to the active continuous variables, or in this case, the *continuous uncertain variables* (aleatory, epistemic, or both, excluding `all_variables` mode; see Section 13.3).

6.2 Sampling Methods

Sampling techniques are selected using the `sampling` method selection. This method generates sets of samples according to the probability distributions of the uncertain variables and maps them into corresponding sets of response functions, where the number of samples is specified by the `samples` integer specification. Means, standard deviations, coefficients of variation (COVs), and 95% confidence intervals are computed for the response

functions. Probabilities and reliabilities may be computed for `response_levels` specifications, and response levels may be computed for either `probability_levels` or `reliability_levels` specifications (refer to the Method Commands chapter in the DAKOTA Reference Manual [3] for additional information).

Currently, traditional Monte Carlo (MC) and Latin hypercube sampling (LHS) are supported by DAKOTA and are chosen by specifying `sample_type` as `random` or `lhs`. In Monte Carlo sampling, the samples are selected randomly according to the user-specified probability distributions. Latin hypercube sampling is a stratified sampling technique for which the range of each uncertain variable is divided into N_s segments of equal probability, where N_s is the number of samples requested. The relative lengths of the segments are determined by the nature of the specified probability distribution (e.g., uniform has segments of equal width, normal has small segments near the mean and larger segments in the tails). For each of the uncertain variables, a sample is selected randomly from each of these equal probability segments. These N_s values for each of the individual parameters are then combined in a shuffling operation to create a set of N_s parameter vectors with a specified correlation structure. A feature of the resulting sample set is that *every row and column in the hypercube of partitions has exactly one sample*. Since the total number of samples is exactly equal to the number of partitions used for each uncertain variable, an arbitrary number of desired samples is easily accommodated (as compared to less flexible approaches in which the total number of samples is a product or exponential function of the number of intervals for each variable, i.e., many classical design of experiments methods).

Advantages of sampling-based methods include their relatively simple implementation and their independence from the scientific disciplines involved in the analysis. The main drawback of these techniques is the large number of function evaluations needed to generate converged statistics, which can render such an analysis computationally very expensive, if not intractable, for real-world engineering applications. LHS techniques, in general, require fewer samples than traditional Monte Carlo for the same accuracy in statistics, but they still can be prohibitively expensive. For further information on the method and its relationship to other sampling techniques, one is referred to the works by McKay, et al. [80], Iman and Shortencarier [69], and Helton and Davis [65]. Note that under certain separability conditions associated with the function to be sampled, Latin hypercube sampling provides a more accurate estimate of the mean value than does random sampling. That is, given an equal number of samples, the LHS estimate of the mean will have less variance than the mean value obtained through random sampling.

Figure 6.1 demonstrates Latin hypercube sampling on a two-variable parameter space. Here, the range of both parameters, x_1 and x_2, is $[0, 1]$. Also, for this example both x_1 and x_2 have uniform statistical distributions. For Latin hypercube sampling, the range of each parameter is divided into p "bins" of equal probability. For parameters with uniform distributions, this corresponds to partitions of equal size. For n design parameters, this partitioning yields a total of p^n bins in the parameter space. Next, p samples are randomly selected in the parameter space, with the following restrictions: (a) each sample is randomly placed inside a bin, and (b) for all one-dimensional projections of the p samples and bins, there will be one and only one sample in each bin. In a two-dimensional example such as that shown in Figure 6.1, these LHS rules guarantee that only one bin can be selected in each row and column. For $p = 4$, there are four partitions in both x_1 and x_2. This gives a total of 16 bins, of which four will be chosen according to the criteria described above. Note that there is more than one possible arrangement of bins that meet the LHS criteria. The dots in Figure 6.1 represent the four sample sites in this example, where each sample is randomly located in its bin. There is no restriction on the number of bins in the range of each parameter, however, all parameters must have the same number of bins.

The actual algorithm for generating Latin hypercube samples is more complex than indicated by the description given above. For example, the Latin hypercube sampling method implemented in the LHS code [109] takes into account a user-specified correlation structure when selecting the sample sites. For more details on the implementation of the LHS algorithm, see Reference [109].

Figure 6.1: An example of Latin hypercube sampling with four bins in design parameters x_1 and x_2. The dots are the sample sites.

6.2.1 Uncertainty Quantification Example using Sampling Methods

The two-variable Textbook example problem (see Equation 2.3) will be used to demonstrate the application of sampling methods for uncertainty quantification where it is assumed that x_1 and x_2 are uniform uncertain variables on the interval $[0, 1]$. The DAKOTA input file for this problem is shown in Figure 6.2. This input file is `dakota_uq_sampling.in` in the `Dakota/examples/methods` directory. The number of samples to perform is controlled with the `samples` specification, the type of sampling algorithm to use is controlled with the `sample_type` specification, the levels used for computing statistics on the response functions is specified with the `response_levels` input, and the `seed` specification controls the sequence of the pseudo-random numbers generated by the sampling algorithms. The input samples generated are shown in Figure 6.3 for the case where `samples` = 5 and `samples` = 10 for both `random` (o) and `lhs` (+) sample types.

Latin hypercube sampling ensures full coverage of the range of the input variables, which is often a problem with Monte Carlo sampling when the number of samples is small. In the case of `samples = 5`, poor stratification is evident in x_1 as four out of the five Monte Carlo samples are clustered in the range $0.35 < x_1 < 0.55$, and the regions $x_1 < 0.3$ and $0.6 < x_1 < 0.9$ are completely missed. For the case where `samples = 10`, some clustering in the Monte Carlo samples is again evident with `4` samples in the range $0.5 < x_1 < 0.55$. In both cases, the stratification with LHS is superior. The response function statistics returned by DAKOTA are shown in Figure 6.4. The first two blocks of output specify the response sample means and sample standard deviations and confidence intervals for these statistics, as well as coefficients of variation. The last section of the output defines CDF pairs (`distribution cumulative` was specified) for the response functions by presenting the probability levels corresponding to the specified response levels (`response_levels` were set and the default `compute probabilities` was used). Alternatively, DAKOTA could have provided CCDF pairings, reliability levels corresponding to prescribed response levels, or response levels corresponding to prescribed probability or reliability levels.

In addition to obtaining statistical summary information of the type shown in Figure 6.4, the results of LHS

```
strategy
        single_method
          tabular_graphics_data
          method_pointer = 'UQ'

method,
        id_method = 'UQ'
        sampling,
          samples = 10
          seed = 98765 rng rnum2
          response_levels = 0.1 0.2 0.6
                            0.1 0.2 0.6
                            0.1 0.2 0.6
          sample_type lhs
          distribution cumulative




variables,
        uniform_uncertain = 2
          lower_bounds =  0.   0.
          upper_bounds =  1.   1.
          descriptors  = 'x1' 'x2'

interface,
        id_interface = 'I1'
        system asynch evaluation_concurrency = 5
          analysis_driver = 'text_book'

responses,
        response_functions = 3
        no_gradients
        no_hessians
```

Figure 6.2: DAKOTA input file for UQ example using LHS sampling.

Figure 6.3: Distribution of input sample points for random (△) and lhs (+) sampling for `samples=5` and `10`.

sampling also include correlations. Four types of correlations are returned in the output: simple and partial "raw" correlations, and simple and partial "rank" correlations. The raw correlations refer to correlations performed on the actual input and output data. Rank correlations refer to correlations performed on the ranks of the data. Ranks are obtained by replacing the actual data by the ranked values, which are obtained by ordering the data in ascending order. For example, the smallest value in a set of input samples would be given a rank 1, the next smallest value a rank 2, etc. Rank correlations are useful when some of the inputs and outputs differ greatly in magnitude: then it is easier to compare if the smallest ranked input sample is correlated with the smallest ranked output, for example.

Correlations are always calculated between two sets of sample data. One can calculate correlation coefficients between two input variables, between an input and an output variable (probably the most useful), or between two output variables. The simple correlation coefficients presented in the output tables are Pearson's correlation coefficient, which is defined for two variables x and y as: $\texttt{Corr}(x,y) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$. Partial correlation coefficients are similar to simple correlations, but a partial correlation coefficient between two variables measures their correlation while adjusting for the effects of the other variables. For example, say one has a problem with two inputs and one output; and the two inputs are highly correlated. Then the correlation of the second input and the output may be very low after accounting for the effect of the first input. The rank correlations in DAKOTA are obtained using Spearman's rank correlation. Spearman's rank is the same as the Pearson correlation coefficient except that it is calculated on the rank data.

Figure 6.5 shows an example of the correlation output provided by DAKOTA for the input file in Figure 6.2. Note that these correlations are presently only available when one specifies `lhs` as the sampling method under `sampling`. Also note that the simple and partial correlations should be similar in most cases (in terms of values of correlation coefficients). This is because we use a default "restricted pairing" method in the LHS routine which forces near-zero correlation amongst uncorrelated inputs.

Finally, note that the LHS package can be used for design of experiments over design and state variables by including the `all_variables` flag in the method specification section of the DAKOTA input file. Then, instead of iterating on only the uncertain variables, the LHS package will sample over all of the variables. In

```
Statistics based on 10 samples:

Moments for each response function:
response_fn_1:  Mean = 3.83840e-01  Std. Dev. = 4.02815e-01
                Coeff. of Variation = 1.04944e+00
response_fn_2:  Mean = 7.47987e-02  Std. Dev. = 3.46861e-01
                Coeff. of Variation = 4.63726e+00
response_fn_3:  Mean = 7.09462e-02  Std. Dev. = 3.41532e-01
                Coeff. of Variation = 4.81397e+00

95% confidence intervals for each response function:
response_fn_1:  Mean = (  9.56831e-02, 6.71997e-01 ),
             Std Dev = (  2.77071e-01, 7.35384e-01 )
response_fn_2:  Mean = ( -1.73331e-01, 3.22928e-01 ),
             Std Dev = (  2.38583e-01, 6.33233e-01 )
response_fn_3:  Mean = ( -1.73371e-01, 3.15264e-01 ),
             Std Dev = (  2.34918e-01, 6.23505e-01 )

Probabilities for each response function:
Cumulative Distribution Function (CDF) for response_fn_1:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   1.0000000000e-01   3.0000000000e-01
   2.0000000000e-01   5.0000000000e-01
   6.0000000000e-01   7.0000000000e-01
Cumulative Distribution Function (CDF) for response_fn_2:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   1.0000000000e-01   5.0000000000e-01
   2.0000000000e-01   7.0000000000e-01
   6.0000000000e-01   9.0000000000e-01
Cumulative Distribution Function (CDF) for response_fn_3:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   1.0000000000e-01   6.0000000000e-01
   2.0000000000e-01   6.0000000000e-01
   6.0000000000e-01   9.0000000000e-01
```

Figure 6.4: DAKOTA response function statistics from UQ sampling example.

```
Simple Correlation Matrix between input and output:
                       x1           x2 response_fn_1 response_fn_2 response_fn_3
          x1  1.00000e+00
          x2 -7.22482e-02  1.00000e+00
response_fn_1 -7.04965e-01 -6.27351e-01  1.00000e+00
response_fn_2  8.61628e-01 -5.31298e-01 -2.60486e-01  1.00000e+00
response_fn_3 -5.83075e-01  8.33989e-01 -1.23374e-01 -8.92771e-01  1.00000e+00

Partial Correlation Matrix between input and output:
             response_fn_1 response_fn_2 response_fn_3
          x1 -9.65994e-01  9.74285e-01 -9.49997e-01
          x2 -9.58854e-01 -9.26578e-01  9.77252e-01

Simple Rank Correlation Matrix between input and output:
                       x1           x2 response_fn_1 response_fn_2 response_fn_3
          x1  1.00000e+00
          x2 -6.66667e-02  1.00000e+00
response_fn_1 -6.60606e-01 -5.27273e-01  1.00000e+00
response_fn_2  8.18182e-01 -6.00000e-01 -2.36364e-01  1.00000e+00
response_fn_3 -6.24242e-01  7.93939e-01 -5.45455e-02 -9.27273e-01  1.00000e+00

Partial Rank Correlation Matrix between input and output:
             response_fn_1 response_fn_2 response_fn_3
          x1 -8.20657e-01  9.74896e-01 -9.41760e-01
          x2 -7.62704e-01 -9.50799e-01  9.65145e-01
```

Figure 6.5: Correlation results using LHS Sampling.

`all_variables` mode, continuous design and continuous state variables are treated as having uniform probability distributions within their upper and lower bounds, discrete values are sampled uniformly from within their sets or ranges, and any uncertain variables are sampled within their specified probability distributions.

6.2.2 Incremental Sampling

In many situations, one may run an initial sample set and then need to perform further sampling to get better estimates of the mean, variance, and percentiles, and to obtain more comprehensive sample coverage. We call this capability incremental sampling. Currently, the LHS incremental sampling capability we have in DAKOTA requires that the incremental samples are double the sample size of the previous sample. That is, if one starts with a very small sample size of 10, then one can use the incremental sampling capability to generate sample sizes of 20, 40, 80, etc. Also, a DAKOTA restart file (dakota.rst) must be available from the original sample. There are two cases, random incremental and Latin Hypercube incremental sampling. We assume that LHS incremental will be most frequently used. One major advantage of LHS incremental sampling is that it maintains the stratification and correlation structure of the original LHS sample. That is, if one generated 2 independent LHS samples and just merged them, the calculation of the accuracy of statistical measures such as the mean and the variance would be slightly incorrect. However, in the incremental case, the full sample (double the original size) is a Latin Hypercube sample itself and statistical measures and their accuracy can be properly calculated. The incremental sampling capability is most useful when one is starting off with very small samples. Once the sample size is more than a few hundred, the benefit of incremental sampling diminishes.

1. Incremental Random Sampling. With incremental random sampling, the original sample set with N samples must be generated using `sample_type` as `random`. Then, the user can create a new DAKOTA input file that is very similar to the original one except the `sample_type` should be defined to be `incremental_random`. Random incremental sampling does not require a doubling of samples each time. Thus, the user can specify the number of samples (`samples`) to be a desired number (it can range from an additional one sample to a large integer), and the `previous_samples` should be specified as N. For example, if the first sample has 50 samples, and 10 more samples are desired, in the second DAKOTA run, the number of samples should be set to 60 and the number of previous samples set to 50. In this situation, only 10 new samples will be generated and the final statistics will be reported on the full sample of 60. The command line syntax for running the second sample is `dakota -i input2.in -r dakota.rst` where input2.in is the input file with the incremental sampling specification and dakota.rst is the restart file. Note that if the restart file has a different name, that is fine; the correct restart file name should be used.

2. Incremental Latin Hypercube Sampling. With incremental LHS sampling, the original sample set with N samples must be generated using `sample_type` as `lhs`. Then, the user can create a new DAKOTA input file that is very similar to the original one except the `sample_type` should be defined to be `incremental_lhs`, the number of samples (`samples`) should be 2N (twice the number of original samples), and `previous_samples` should be specified as N. For example, if the first sample has 50 samples, in the second DAKOTA run, the number of samples should be set to 100 and the number of previous samples set to 50. In this situation, only 50 new samples will be generated and the final statistics will be reported on the full sample of 100. The command line syntax for running the second sample is `dakota -i input2.in -r dakota.rst`, where input2.in is the input file with the incremental sampling specification and dakota.rst is the restart file. Note that if the restart file has a different name, that is fine; the correct restart file name should be used.

6.3 Reliability Methods

Reliability methods provide an alternative approach to uncertainty quantification which can be less computationally demanding than sampling techniques. Reliability methods for uncertainty quantification are based on probabilistic approaches that compute approximate response function distribution statistics based on specified uncertain variable distributions. These response statistics include response mean, response standard deviation, and cumulative or complementary cumulative distribution functions (CDF/CCDF). These methods are often more efficient at computing statistics in the tails of the response distributions (events with low probability) than sampling based approaches since the number of samples required to resolve a low probability can be prohibitive.

The methods all answer the fundamental question: "Given a set of uncertain input variables, $\mathbf{X}$, and a scalar response function, g, what is the probability that the response function is below or above a certain level, $\bar{z}$?" The former can be written as $P[g(\mathbf{X}) \leq \bar{z}] = F_g(\bar{z})$ where $F_g(\bar{z})$ is the cumulative distribution function (CDF) of the uncertain response $g(\mathbf{X})$ over a set of response levels. The latter can be written as $P[g(\mathbf{X}) > \bar{z}]$ and defines the complementary cumulative distribution function (CCDF).

This probability calculation involves a multi-dimensional integral over an irregularly shaped domain of interest, $\mathbf{D}$, where $g(\mathbf{X}) < z$ as displayed in Figure 6.6 for the case of two variables. The reliability methods all involve the transformation of the user-specified uncertain variables, $\mathbf{X}$, with probability density function, $p(x_1, x_2)$, which can be non-normal and correlated, to a space of independent Gaussian random variables, $\mathbf{u}$, possessing a mean value of zero and unit variance (i.e., standard normal variables). The region of interest, $\mathbf{D}$, is also mapped to the transformed space to yield, $\mathbf{D_u}$, where $g(\mathbf{U}) < z$ as shown in Figure 6.7. The Nataf transformation [20], which is identical to the Rosenblatt transformation [95] in the case of independent random variables, is used in DAKOTA to accomplish this mapping. This transformation is performed to make the probability calculation more tractable. In the transformed space, probability contours are circular in nature as shown in Figure 6.7 unlike in the original uncertain variable space, Figure 6.6. Also, the multi-dimensional integrals can be approximated by simple functions of a single parameter, β, called the reliability index. β is the minimum Euclidean distance from the origin in the transformed space to the response surface. This point is also known as the most probable point (MPP) of failure. Note, however, the methodology is equally applicable for generic functions, not simply those corresponding to failure criteria; this nomenclature is due to the origin of these methods within the disciplines of structural safety and reliability. Note that there are local and global reliability methods. The majority of the methods available are local, meaning that a local optimization formulation is used to locate one MPP. In contrast, global methods can find multiple MPPs if they exist.

6.3.1 Local Reliability Methods

The DAKOTA Theory Manual [2] provides the algorithmic details for the local reliability methods, including the Mean Value method and the family of most probable point (MPP) search methods.

6.3.1.1 Method mapping

Given settings for limit state approximation, approximation order, integration approach, and other details presented to this point, it is evident that the number of algorithmic combinations is high. Table 6.1 provides a succinct mapping for some of these combinations to common method names from the reliability literature, where blue indicates the most well-known combinations and gray indicates other supported combinations.

Within the DAKOTA specification (refer to the Method Commands chapter within the Reference Manual), the MPP search and integration order selections are explicit in the method specification, but the order of the approximation is inferred from the associated response specification (as is done with local taylor series approximations

$$P[g(\mathbf{X}) < z] = \iint_{\mathbf{x} \in D} p(x_1, x_2) d\mathbf{x} = P[(\mathbf{x} \in \mathbf{D})]$$

Figure 6.6: Graphical depiction of calculation of cumulative distribution function in the original uncertain variable space.

$$P(\mathbf{X} \in D) = P(U \in D_U) \approx f(\beta)$$

Figure 6.7: Graphical depiction of integration for the calculation of cumulative distribution function in the transformed uncertain variable space.

Table 6.1: Mapping from DAKOTA options to standard reliability methods.

MPP search	Order of approximation and integration	
	First order	Second order
none	MVFOSM	MVSOSM
x_taylor_mean	AMV	AMV^2
u_taylor_mean	u-space AMV	u-space AMV^2
x_taylor_mpp	AMV+	AMV^2+
u_taylor_mpp	u-space AMV+	u-space AMV^2+
x_two_point	TANA	
u_two_point	u-space TANA	
no_approx	FORM	SORM

described in Section 10.4.1.2). Thus, reliability methods do not have to be synchronized in order as shown in the table; however, it is often desirable to do so.

6.3.2 Global Reliability Methods

Global reliability methods are designed to handle nonsmooth and multimodal failure surfaces, by creating global approximations based on Gaussian process models. They accurately resolve a particular contour of a response function and then estimate probabilities using multimodal adaptive importance sampling.

The global reliability method in DAKOTA is called Efficient Global Reliability Analysis (EGRA) [9]. The name is due to its roots in efficient global optimization (EGO) [70, 68]. The main idea in EGO-type optimization methods is that a global approximation is made of the underlying function. This approximation, which is a Gaussian process model, is used to guide the search by finding points which maximize the expected improvement function (EIF). The EIF is used to select the location at which a new training point should be added to the Gaussian process model by maximizing the amount of improvement in the objective function that can be expected by adding that point. A point could be expected to produce an improvement in the objective function if its predicted value is better than the current best solution, or if the uncertainty in its prediction is such that the probability of it producing a better solution is high. Because the uncertainty is higher in regions of the design space with fewer observations, this provides a balance between exploiting areas of the design space that predict good solutions, and exploring areas where more information is needed.

The general procedure of these EGO-type methods is:

1. Build an initial Gaussian process model of the objective function.
2. Find the point that maximizes the EIF. If the EIF value at this point is sufficiently small, stop.
3. Evaluate the objective function at the point where the EIF is maximized. Update the Gaussian process model using this new point. Go to Step 2.

Gaussian process (GP) models are used because they provide not just a predicted value at an unsampled point, but also an estimate of the prediction variance. This variance gives an indication of the uncertainty in the GP model, which results from the construction of the covariance function. This function is based on the idea that when input points are near one another, the correlation between their corresponding outputs will be high. As a result, the uncertainty associated with the model's predictions will be small for input points which are near the points used to train the model, and will increase as one moves further from the training points.

The expected improvement function is used in EGO algorithms to select the location at which a new training point should be added. The EIF is defined as the expectation that any point in the search space will provide a better solution than the current best solution based on the expected values and variances predicted by the GP model. It is important to understand how the use of this EIF leads to optimal solutions. The EIF indicates how much the objective function value at a new potential location is expected to be less than the predicted value at the current best solution. Because the GP model provides a Gaussian distribution at each predicted point, expectations can be calculated. Points with good expected values and even a small variance will have a significant expectation of producing a better solution (exploitation), but so will points that have relatively poor expected values and greater variance (exploration).

The application of EGO to reliability analysis, however, is made more complicated due to the inclusion of equality constraints. In forward reliability analysis, the response function appears as a constraint rather than the objective. That is, we want to satisfy the constraint that the response equals a threshold value and is on the limit state: $G(\mathbf{u}) = \bar{z}$. Therefore, the EIF function was modified to focus on feasibility, and instead of using an expected improvement function, we use an expected feasibility function (EFF) [9]. The EFF provides an indication of how well the response is expected to satisfy the equality constraint. Points where the expected value is close to the threshold ($\mu_G \approx \bar{z}$) and points with a large uncertainty in the prediction will have large expected feasibility values.

The general outline of the EGRA algorithm is as follows: LHS sampling is used to generate a small number of samples from the true response function. Then, an initial Gaussian process model is constructed. Based on the EFF, the point with maximum EFF is found using the global optimizer DIRECT. The true response function is then evaluated at this new point, and this point is added to the sample set and the process of building a new GP model and maximizing the EFF is repeated until the maximum EFF is small. At this stage, the GP model is accurate in the vicinity of the limit state. The GP model is then used to calculate the probability of failure using multimodal importance sampling, which is explained below.

One method to calculate the probability of failure is to directly perform the probability integration numerically by sampling the response function. Sampling methods can be prohibitively expensive because they generally require a large number of response function evaluations. Importance sampling methods reduce this expense by focusing the samples in the important regions of the uncertain space. They do this by centering the sampling density function at the MPP rather than at the mean. This ensures the samples will lie the region of interest, thus increasing the efficiency of the sampling method. Adaptive importance sampling (AIS) further improves the efficiency by adaptively updating the sampling density function. Multimodal adaptive importance sampling [21] is a variation of AIS that allows for the use of multiple sampling densities making it better suited for cases where multiple sections of the limit state are highly probable.

Note that importance sampling methods require that the location of at least one MPP be known because it is used to center the initial sampling density. However, current gradient-based, local search methods used in MPP search may fail to converge or may converge to poor solutions for highly nonlinear problems, possibly making these methods inapplicable. The EGRA algorithm described above does not depend on the availability of accurate gradient information, making convergence more reliable for nonsmooth response functions. Moreover, EGRA has the ability to locate multiple failure points, which can provide multiple starting points and thus a good multimodal sampling density for the initial steps of multimodal AIS. The probability assessment using multimodal AIS thus incorporates probability of failure at multiple points.

6.3.3 Uncertainty Quantification Example using Reliability Analysis

In summary, the user can choose to perform either forward (RIA) or inverse (PMA) mappings when performing a reliability analysis. With either approach, there are a variety of methods from which to choose in terms of limit state approximations (MVFOSM, MVSOSM, x-/u-space AMV, x-/u-space AMV^2, x-/u-space AMV+, x-/u-space AMV^2+, x-/u-space TANA, and FORM/SORM), probability integrations (first-order or second-order), limit state

Hessian selection (analytic, finite difference, BFGS, or SR1), and MPP optimization algorithm (SQP or NIP) selections.

All reliability methods output approximate values of the CDF/CCDF response-probability-reliability levels for prescribed response levels (RIA) or prescribed probability or reliability levels (PMA). In addition, the MV methods additionally output estimates of the mean and standard deviation of the response functions along with importance factors for each of the uncertain variables in the case of independent random variables.

This example quantifies the uncertainty in the "log ratio" response function:

$$g(x_1, x_2) = \frac{x_1}{x_2} \tag{6.1}$$

by computing approximate response statistics using reliability analysis to determine the response cumulative distribution function:

$$P[g(x_1, x_2) < \bar{z}] \tag{6.2}$$

where X_1 and X_2 are identically distributed lognormal random variables with means of `1`, standard deviations of `0.5`, and correlation coefficient of `0.3`.

A DAKOTA input file showing RIA using FORM (option 7 in limit state approximations combined with first-order integration) is listed in Figure 6.8. This input file is `dakota_uq_reliability.in` in the Dakota/examples/methods directory. The user first specifies the `local_reliability` method, followed by the MPP search approach and integration order. In this example, we specify `mpp_search no_approx` and utilize the default first-order integration to select FORM. Finally, the user specifies response levels or probability/reliability levels to determine if the problem will be solved using an RIA approach or a PMA approach. In the example figure of 6.8, we use RIA by specifying a range of `response_levels` for the problem. The resulting output for this input is shown in Figure 6.9, with probability and reliability levels listed for each response level. Figure 6.10 shows that FORM compares favorably to an exact analytic solution for this problem. Also note that FORM does have some error in the calculation of CDF values for this problem, but it is a very small error (on the order of e-11), much smaller than the error obtained when using a Mean Value method, which will be discussed next.

If the user specifies `local_reliability` as a method with no additional specification on how to do the MPP search, then no MPP search is done: the Mean Value method is used. The MV results are shown in Figure 6.11 and consist of approximate mean and standard deviation of the response, the importance factors for each uncertain variable, and approximate probability/reliability levels for the prescribed response levels that have been inferred from the approximate mean and standard deviation (see Mean Value section in Reliability Methods Chapter of DAKOTA Theory Manual [2]). It is evident that the statistics are considerably different from the fully converged FORM results; however, these rough approximations are also much less expensive to calculate. The importance factors are a measure of the sensitivity of the response function(s) to the uncertain input variables, but in this case, are not separable due to the presence of input correlation coefficients. A comparison of the mean value results with the FORM results is shown in Figure 6.10. The mean value results are not accurate near the tail values of the CDF, and can differ from the exact solution by as much as 0.11 in CDF estimates. A comprehensive comparison of various reliability methods applied to the logratio problem is provided in [25].

Additional reliability analysis and design results are provided in Sections 23.6-23.11.

6.4 Stochastic Expansion Methods

The objective of these techniques is to characterize the response of systems whose governing equations involve stochastic coefficients. The development of these techniques mirrors that of deterministic finite element analysis through the utilization of the concepts of projection, orthogonality, and weak convergence. The polynomial chaos

```
strategy,
        single_method graphics

method,
        local_reliability
          mpp_search no_approx
          response_levels = .4 .5 .55 .6 .65 .7
           .75 .8 .85 .9 1. 1.05 1.15 1.2 1.25 1.3
           1.35 1.4 1.5 1.55 1.6 1.65 1.7 1.75

variables,
        lognormal_uncertain = 2
          means             =  1.  1
          std_deviations    =  0.5 0.5
          descriptors       =  'TF1ln'   'TF2ln'
        uncertain_correlation_matrix =  1   0.3
                                        0.3 1

interface,
        system asynch
          analysis_driver = 'log_ratio'

responses,
        response_functions = 1
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-4
        no_hessians
```

Figure 6.8: DAKOTA input file for Reliability UQ example using FORM.

```
Cumulative Distribution Function (CDF) for response_fn_1:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   4.0000000000e-01   4.7624085962e-02   1.6683404020e+00
   5.0000000000e-01   1.0346525475e-01   1.2620507942e+00
   5.5000000000e-01   1.3818404972e-01   1.0885143628e+00
   6.0000000000e-01   1.7616275822e-01   9.3008801339e-01
   6.5000000000e-01   2.1641741368e-01   7.8434989943e-01
   7.0000000000e-01   2.5803428381e-01   6.4941748143e-01
   7.5000000000e-01   3.0020938124e-01   5.2379840558e-01
   8.0000000000e-01   3.4226491013e-01   4.0628960782e-01
   8.5000000000e-01   3.8365052982e-01   2.9590705956e-01
   9.0000000000e-01   4.2393548232e-01   1.9183562480e-01
   1.0000000000e+00   5.0000000000e-01   6.8682233460e-12
   1.0500000000e+00   5.3539344228e-01  -8.8834907167e-02
   1.1500000000e+00   6.0043460094e-01  -2.5447217462e-01
   1.2000000000e+00   6.3004131827e-01  -3.3196278078e-01
   1.2500000000e+00   6.5773508987e-01  -4.0628960782e-01
   1.3000000000e+00   6.8356844630e-01  -4.7770089473e-01
   1.3500000000e+00   7.0761025532e-01  -5.4641676380e-01
   1.4000000000e+00   7.2994058691e-01  -6.1263331274e-01
   1.5000000000e+00   7.6981945355e-01  -7.3825238860e-01
   1.5500000000e+00   7.8755158269e-01  -7.9795460350e-01
   1.6000000000e+00   8.0393505584e-01  -8.5576118635e-01
   1.6500000000e+00   8.1906005158e-01  -9.1178881995e-01
   1.7000000000e+00   8.3301386860e-01  -9.6614373461e-01
   1.7500000000e+00   8.4588021938e-01  -1.0189229206e+00
```

Figure 6.9: Output from Reliability UQ example using FORM.

Figure 6.10: Comparison of the cumulative distribution function (CDF) computed by FORM, the Mean Value method, and the exact CDF for $g(x_1, x_2) = \frac{x_1}{x_2}$

```
MV Statistics for response_fn_1:
  Approximate Mean Response                  =  1.0000000000e+00
  Approximate Standard Deviation of Response =  5.9160798127e-01
  Importance Factors not available.
Cumulative Distribution Function (CDF) for response_fn_1:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   4.0000000000e-01   1.5524721837e-01   1.0141851006e+00
   5.0000000000e-01   1.9901236093e-01   8.4515425050e-01
   5.5000000000e-01   2.2343641149e-01   7.6063882545e-01
   6.0000000000e-01   2.4948115037e-01   6.7612340040e-01
   6.5000000000e-01   2.7705656603e-01   5.9160797535e-01
   7.0000000000e-01   3.0604494093e-01   5.0709255030e-01
   7.5000000000e-01   3.3630190949e-01   4.2257712525e-01
   8.0000000000e-01   3.6765834596e-01   3.3806170020e-01
   8.5000000000e-01   3.9992305332e-01   2.5354627515e-01
   9.0000000000e-01   4.3288618783e-01   1.6903085010e-01
   1.0000000000e+00   5.0000000000e-01   0.0000000000e+00
   1.0500000000e+00   5.3367668035e-01  -8.4515425050e-02
   1.1500000000e+00   6.0007694668e-01  -2.5354627515e-01
   1.2000000000e+00   6.3234165404e-01  -3.3806170020e-01
   1.2500000000e+00   6.6369809051e-01  -4.2257712525e-01
   1.3000000000e+00   6.9395505907e-01  -5.0709255030e-01
   1.3500000000e+00   7.2294343397e-01  -5.9160797535e-01
   1.4000000000e+00   7.5051884963e-01  -6.7612340040e-01
   1.5000000000e+00   8.0098763907e-01  -8.4515425050e-01
   1.5500000000e+00   8.2372893005e-01  -9.2966967555e-01
   1.6000000000e+00   8.4475278163e-01  -1.0141851006e+00
   1.6500000000e+00   8.6405064339e-01  -1.0987005257e+00
   1.7000000000e+00   8.8163821351e-01  -1.1832159507e+00
   1.7500000000e+00   8.9755305196e-01  -1.2677313758e+00
```

Figure 6.11: Output from Reliability UQ example using MV.

expansion is based on a multidimensional orthogonal polynomial approximation in standardized random variables, and the stochastic collocation approach is based on a multidimensional interpolation polynomial approximation in standardized random variables. A distinguishing feature of these two methodologies is that the final solution is expressed as a random process, and not merely as a set of statistics as is the case for many nondeterministic methodologies. This makes these techniques particularly attractive for use in multi-physics applications which link different analysis packages. The first stochastic expansion method is the polynomial chaos expansion (PCE) described in Ghanem, et al. [43], [44]. For smooth functions (i.e., analytic, infinitely-differentiable) in L^2 (i.e., possessing finite variance), exponential convergence rates can be obtained under order refinement for integrated statistical quantities of interest such as mean, variance, and probability. DAKOTA implements the generalized polynomial chaos approach using the Wiener-Askey scheme [126], in which Hermite, Legendre, Laguerre, Jacobi, and generalized Laguerre orthogonal polynomials are used for modeling the effect of continuous random variables described by normal, uniform, exponential, beta, and gamma probability distributions, respectively[1]. These orthogonal polynomial selections are optimal for these distribution types since the inner product weighting function corresponds[2] to the probability density functions for these continuous distributions. Orthogonal polynomials can be computed for any positive weight function, so these five classical orthogonal polynomials may be augmented with numerically-generated polynomials for other probability distributions (e.g., for lognormal, extreme value, and histogram distributions). When independent standard random variables are used (or computed through transformation), the variable expansions are uncoupled, allowing the polynomial orthogonality properties to be applied on a per-dimension basis. This allows one to mix and match the polynomial basis used for each variable without interference with the spectral projection scheme for the response.

In non-intrusive PCE, simulations are used as black boxes and the calculation of chaos expansion coefficients for response metrics of interest is based on a set of simulation response evaluations. To calculate these response PCE coefficients, two primary classes of approaches have been proposed: spectral projection and linear regression. The spectral projection approach projects the response against each basis function using inner products and employs the polynomial orthogonality properties to extract each coefficient. Each inner product involves a multidimensional integral over the support range of the weighting function, which can be evaluated numerically using sampling, tensor-product quadrature, Smolyak sparse grid [104], or cubature [106] approaches. The linear regression approach uses a single linear least squares solution to solve for the set of PCE coefficients which best match a set of response values obtained from either a design of computer experiments ("point collocation" [121]) or from the subset of tensor Gauss points with highest product weight ("probabilistic collocation" [112]).

Stochastic collocation (SC) is another stochastic expansion technique for UQ that is closely related to PCE. As for PCE, exponential convergence rates can be obtained under order refinement for integrated statistical quantities of interest, provided that the response functions are smooth with finite variance. The primary distinction is that, whereas PCE estimates coefficients for known multivariate orthogonal polynomial basis functions, SC forms multivariate interpolation polynomial bases for known coefficients. The interpolation polynomials may be local or global, value-based or gradient-enhanced, and nodal or hierarchical, with a total of six combinations currently supported: Lagrange (global value-based), Hermite (global gradient-enhanced), piecewise linear spline (local value-based) in nodal and hierarchical formulations, and piecewise cubic spline (local gradient-enhanced) in nodal and hierarchical formulations. Interpolation is performed on structured grids such as tensor-product or sparse grids. Starting from a tensor-product multidimensional interpolation polynomial in the value-based case (Lagrange or piecewise linear spline), we have the feature that the ith interpolation polynomial has a value of 1 at collocation point i and a value of 0 for all other collocation points, leading to the use of expansion coefficients that are just the response values at each of the collocation points. In the gradient-enhanced case (Hermite or piecewise cubic spline), SC includes both type 1 and type 2 interpolation polynomials, where the former interpolate the values while producing zero gradients and the latter interpolate the gradients while producing zero values (refer to [2] for additional details). Sparse interpolants are weighted sums of these tensor interpolants;

[1] Orthogonal polynomial selections also exist for discrete probability distributions, but are not yet supported in DAKOTA.
[2] Identical support range; weight differs by at most a constant factor.

however, they are only interpolatory for sparse grids based on fully nested rules and will exhibit some interpolation error at the collocation points for sparse grids based on non-nested rules. A key to maximizing performance with SC is performing collocation using the Gauss points and weights from the same optimal orthogonal polynomials used in PCE. For use of standard Gauss integration rules (not nested variants such as Gauss-Patterson or Genz-Keister) within tensor-product quadrature, tensor PCE expansions and tensor SC interpolants are equivalent in that identical polynomial approximations are generated [15]. Moreover, this equivalence can be extended to sparse grids based on standard Gauss rules, provided that a sparse PCE is formed based on a weighted sum of tensor expansions [14].

The DAKOTA Theory Manual [2] provides full algorithmic details for the PCE and SC methods.

6.4.1 Uncertainty Quantification Example using Stochastic Collocation

A typical DAKOTA input file for performing an uncertainty quantification using polynomial chaos expansions is shown in the Tutorial Chapter, Section 2.4.3.3. The example in the Tutorial Chapter illustrates PCE built on anisotropic tensor product quadrature. The uncertain variables are uniforms, so the expansion is built using classical Legendre polynomials. This section presents a more sophisticated example, where we use stochastic collocation built on an anisotropic sparse grid defined from numerically-generated orthogonal polynomials. The uncertain variables are lognormal in this example and the orthogonal polynomials are generated from Gauss-Wigert recursion coefficients [102] in combination with the Golub-Welsch procedure [54]. The input file is shown in Figure 6.12. This input file is `dakota_sc.in` in the `Dakota/examples/methods` directory. Note that the dimension preference of $(2, 1)$ is inverted to define a γ weighting vector of $(0.5, 1)$ (and $\underline{\gamma}$ of 0.5) for use in the anisotropic Smolyak index set constraint (see Smolyak sparse grids section in Stochastic Expansion Methods chapter in DAKOTA Theory Manual [2]). In this example, we compute CDF probabilities for six response levels of Rosenbrock's function. This example requires 19 function evaluations to calculate the interpolating polynomials in stochastic collocation and the resulting expansion exactly reproduces Rosenbrock's function. The placement of the points generated by the sparse grid is shown in Figure 6.13.

Once the expansion coefficients have been calculated, some statistics are available analytically and others must be evaluated numerically. For the numerical portion, the input file specifies the use of 10000 samples, which will be evaluated on the expansion to compute the CDF probabilities. In Figure 6.14, excerpts from the results summary are presented. We first see the moment statistics for mean, standard deviation, skewness, and kurtosis computed by numerical integration (see Analytic moments section in Stochastic Expansion Methods chapter in DAKOTA Theory Manual [2]), where the numerical row corresponds to integration using the original response values and the expansion row corresponds to integration using values from the interpolant. The response covariance (collapsing to a single variance value for one response function) and global sensitivity indices (Sobol' indices) are presented next. This example shows that variable x1 has the largest main effect (0.99) as compared with variable x2 (0.0007) or the interaction between x1 and x2 (0.005). Finally, we see the numerical results for the CDF probabilities based on 10000 samples performed on the expansion. For example, the probability that the Rosenbrock function is less than 100 is 0.7233. Note that these results are significantly different than the ones presented in Section 2.4.3.3 because of the different assumptions about the inputs: uniform[-2,2] versus lognormals with means of 1.0 and standard deviations of 0.5.

6.5 Epistemic Nondeterministic Methods

Uncertainty quantification is often used as part of the risk assessment of performance, reliability, and safety of engineered systems. Increasingly, uncertainty is separated into two categories for analysis purposes: aleatory and epistemic uncertainty [87, 66]. Aleatory uncertainty is also referred to as variability, irreducible or inherent

```
strategy,
        single_method #graphics

method,
        stoch_collocation
          sparse_grid_level = 3
          dimension_preference = 2 1
          samples = 10000 seed = 12347 rng rnum2
          response_levels = .1 1. 50. 100. 500. 1000.
          variance_based_decomp #univariate_effects
          output silent

variables,
        lognormal_uncertain = 2
          means             =  1.   1.
          std_deviations    =  0.5  0.5
          descriptors       = 'x1' 'x2'

interface,
        direct
        analysis_driver = 'rosenbrock'

responses,
        response_functions = 1
        no_gradients
        no_hessians
```

Figure 6.12: DAKOTA input file for performing UQ using stochastic collocation.

Figure 6.13: Rosenbrock stochastic collocation example: sparse grid points.

uncertainty, or uncertainty due to chance. Examples of aleatory uncertainty include the height of individuals in a population, or the temperature in a processing environment. Aleatory uncertainty is usually modeled with probability distributions, and sampling methods such as Latin Hypercube sampling in DAKOTA can be used to model aleatory uncertainty. In contrast, epistemic uncertainty refers to lack of knowledge or lack of information about a particular aspect of the simulation model, including the system and environment being modeled. An increase in knowledge or information relating to epistemic uncertainty will lead to a reduction in the predicted uncertainty of the system response or performance. For epistemic uncertain variables, typically one does not know enough to specify a probability distribution on a variable. Epistemic uncertainty is referred to as subjective, reducible, or lack of knowledge uncertainty. Examples of epistemic uncertainty include little or no experimental data for a fixed but unknown physical parameter, incomplete understanding of complex physical phenomena, uncertainty about the correct model form to use, etc.

There are many approaches which have been developed to model epistemic uncertainty, including fuzzy set theory, possibility theory, and evidence theory. It is also possible to use simple interval analysis in an epistemic context. Interval analysis and evidence theory are described in more detail below.

6.5.1 Interval Methods for Epistemic Analysis

In interval analysis, one assumes that nothing is known about an epistemic uncertain variable except that its value lies somewhere within an interval. In this situation, it is NOT assumed that the value has a uniform probability of occuring within the interval. Instead, the interpretation is that any value within the interval is a possible value or a potential realization of that variable. In interval analysis, the uncertainty quantification problem is one of determining the resulting bounds on the output (defining the output interval) given interval bounds on the inputs. Again, any output response that falls within the output interval is a possible output with no frequency information assigned to it.

We have the capability to perform interval analysis using either `global_interval_est` or `local_interval_est`. In the global approach, one uses either a global optimization method or a sampling method to assess the bounds. `global_interval_est` allows the user to specify either `lhs`, which performs Latin Hypercube Sampling and takes the minimum and maximum of the samples as the bounds (no optimization is performed) or `ego`. In the case of `ego`, the efficient global optimization method is used to calculate bounds. The ego method is de-

```
Statistics derived analytically from polynomial expansion:

Moment-based statistics for each response function:
                            Mean           Std Dev          Skewness          Kurtosis
response_fn_1
  expansion:    2.5671972656e+02  2.0484189184e+03  2.7419241630e+02  1.9594567379e+06
  numerical:    2.5671972656e+02  2.0484189184e+03  2.7419241630e+02  1.9594567379e+06

Covariance among response functions:
[[  4.1960200651e+06 ]]

Global sensitivity indices for each response function:
response_fn_1 Sobol indices:
                                  Main             Total
                      9.9391978710e-01  9.9928724777e-01 x1
                      7.1275222945e-04  6.0802128961e-03 x2
                           Interaction
                      5.3674606667e-03 x1 x2

Statistics based on 10000 samples performed on polynomial expansion:

Level mappings for each response function:
Cumulative Distribution Function (CDF) for response_fn_1:
     Response Level  Probability Level  Reliability Index  General Rel Index
     --------------  -----------------  -----------------  -----------------
   1.0000000000e-01   1.8100000000e-02
   1.0000000000e+00   8.7800000000e-02
   5.0000000000e+01   5.8410000000e-01
   1.0000000000e+02   7.2330000000e-01
   5.0000000000e+02   9.2010000000e-01
   1.0000000000e+03   9.5660000000e-01
```

Figure 6.14: Excerpt of UQ output for stochastic collocation example.

scribed in Section 9.4. If the problem is amenable to local optimization methods (e.g. can provide derivatives or use finite difference method to calculate derivatives), then one can use local methods to calculate these bounds. `local_interval_est` allows the user to specify either `sqp` which is sequential quadratic programming, or `nip` which is a nonlinear interior point method.

Note that when performing interval analysis, it is necessary to define interval uncertain variables as described in Section 11.3. For interval analysis, one must define only one interval per input variable, in contrast with Dempster-Shafer evidence theory, where an input can have several possible intervals. Interval analysis can be considered a special case of Dempster-Shafer evidence theory where each input is defined by one input interval with a basic probability assignment of one. In DAKOTA, however, the methods are separate and semantic differences exist in the output presentation. If you are performing a pure interval analysis, we recommend using either `global_interval_est` or `local_interval_est` instead of `global_evidence` or `local_evidence`, for reasons of simplicity. An example of interval estimation is found in the test file `dakota_uq_cantilever_interval.in`, and also in the Tutorial, Section 2.4.3.4. These interval methods can also be used as the outer loop within an interval-valued probability analysis for propagating mixed aleatory and epistemic uncertainty – refer to Section 17.1.1 for additional details.

6.5.2 Dempster-Shafer Theory of Evidence

We have chosen to pursue evidence theory at Sandia as a way to model epistemic uncertainty, in part because evidence theory is a generalization of probability theory. Evidence theory is also referred to as Dempster-Shafer theory or the theory of random sets [87]. This section focuses on the use of Dempster-Shafer evidence theory for propagating epistemic uncertainties. When aleatory uncertainties are also present, we may choose either to discretize the aleatory probability distributions into sets of intervals and treat them as well-characterized epistemic variables, or we may choose to segregate the aleatory uncertainties and treat them within an inner loop. A nested Dempster-Shafer approach for propagating mixed aleatory and epistemic uncertainty is described in Section 17.1.3.

In evidence theory, there are two complementary measures of uncertainty: belief and plausibility. Together, belief and plausibility can be thought of as defining lower and upper bounds, respectively, on probabilities. Belief and plausibility define the lower and upper limits or intervals on probability values. Typical plots of cumulative and complementary cumulative belief and plausibility functions are shown in Figure 6.15 [66]. In evidence theory, it is not possible to specify one probability value. Instead, there is a range of values that is consistent with the evidence. The range of values is defined by belief and plausibility. Note that no statement or claim is made about one value within an interval being more or less likely than any other value.

In Dempster-Shafer evidence theory, the uncertain input variables are modeled as sets of intervals. The user assigns a basic probability assignment (BPA) to each interval, indicating how likely it is that the uncertain input falls within the interval. The BPAs for a particular uncertain input variable must sum to one. The intervals may be overlapping, contiguous, or have gaps. In DAKOTA, an interval uncertain variable is specified as `interval_uncertain`. When one defines an interval type variable in DAKOTA, it is also necessary to specify the number of intervals defined for each variable with `iuv_num_intervals` as well the basic probability assignments per interval, `iuv_interval_probs`, and the associated bounds per each interval, `iuv_interval_bounds`. Figure 6.16 shows the input specification for interval uncertain variables. This input file is `dakota_uq_textbook_dste.in`, found in the `Dakota/examples/methods` directory. The example shown in Figure 6.16 has two epistemic uncertain interval variables. The first uncertain variable has three intervals and the second has two. The basic probability assignments for the first variable are 0.5, 0.1, and 0.4, while the BPAs for the second variable are 0.7 and 0.3. Note that it is possible (and often the case) to define an interval uncertain variable with only ONE interval. This means that you only know that the possible value of that variable falls within the interval, and the BPA for that interval would be 1.0. In the case we have shown, the

Figure 6.15: Example cumulative belief and plausibility distribution functions on left; complementary cumulative belief and plausibility distribution functions on right

interval bounds on the first interval for the first variable are 0.6 and 0.9, and the bounds for the second interval for the first variable are 0.1 to 0.5, etc.

Once the intervals, the BPAs, and the interval bounds are defined, the user can run an epistemic analysis by specifying the method as either `global_evidence` or `local_evidence` in the DAKOTA input file. Both of these methods perform Dempster-Shafer calculations: the difference is that the local method uses a local optimization algorithm to calculate the interval bounds and the global method uses either sampling or a global optimization approach to calculate an interval bound. These differences are discussed in more detail below. The intervals and their associated BPAs are then propagated through the simulation to obtain cumulative distribution functions on belief and plausibility. As mentioned above, belief is the lower bound on a probability estimate that is consistent with the evidence, and plausibility is the upper bound on a probability estimate that is consistent with the evidence.

Figure 6.17 shows results for the first response function obtained when running the example in Figure 6.16. In this example, there are 6 output intervals (as a result of the 2 interval input variables with 3 and 2 intervals, respectively). The output intervals are ordered to obtain cumulative bound functions for both belief and plausibility. The cumulative distribution function is presented for both belief (CBF) and plausibility (CPF). The CBF value is the cumulative belief corresponding to a certain output value. For example, the belief that the output value is less than or equal to 0.2 for response 1 is 0.27, and the plausibility that the output is less than or equal to 0.2 is 1 for response 1. The belief that the output value is less than 0.6217 is 0.75, while the plausbility that the output is less than 0.0806 is 0.75. The CBF and CPF may be plotted on a graph and interpreted as bounding the cumulative distribution function (CDF), which is the probability that the output is less than or equal to a certain value. The interval bounds on probability values show the value of epistemic uncertainty analysis: the intervals are usually much larger than expected, giving one a truer picture of the total output uncertainty caused by lack of knowledge or information about the epistemic input quantities.

As in other nondeterministic methods, with `local_evidence` or `global_evidence`, one can specify probability levels and response levels. If response levels are specified, the belief and plausibility function values corresponding to those response levels are calculated (see Belief Prob Level and Plaus Prob Level in the tables shown in Figure 6.17). Similarly, if probability levels are specified, these are first interpreted to be belief values, and the corresponding response levels are calculated (see Belief Resp Level); then they are interpreted to

```
strategy,
        single_method
          tabular_graphics_data

method,
        global_evidence lhs
          samples = 1000
          seed = 59334 rng rnum2
          response_levels = 0.001 0.03 0.2 0.8 0.001 0.2 0.6 0.8
          probability_levels = 0.25 0.5 0.75 0.25 0.5 0.75
          distribution cumulative
          output verbose

variables,
        interval_uncertain = 2
          num_intervals   = 3 2
          interval_probs  = 0.5 0.1 0.4 0.7 0.3
          interval_bounds = 0.6 0.9 0.1 0.5 0.5 1.0 0.3 0.5 0.6 0.8

interface,
        direct
          analysis_driver = 'text_book'

responses,
        response_functions = 2
        no_gradients
        no_hessians
```

Figure 6.16: DAKOTA input file for UQ example using Evidence Theory.

```
Belief and Plausibility for each response function:
Cumulative Belief/Plausibility Functions (CBF/CPF) for response_fn_1:
     Response Level  Belief Prob Level   Plaus Prob Level
     --------------  -----------------   ----------------
   1.0000000000e-03   0.0000000000e+00   0.0000000000e+00
   3.0000000000e-02   0.0000000000e+00   2.7000000000e-01
   2.0000000000e-01   2.7000000000e-01   1.0000000000e+00
   8.0000000000e-01   9.3000000000e-01   1.0000000000e+00
  Probability Level  Belief Resp Level   Plaus Resp Level
  -----------------  -----------------   ----------------
   2.5000000000e-01   2.6187288772e-01   6.2609206069e-02
   5.0000000000e-01   2.9829775860e-01   6.3736734971e-02
   7.5000000000e-01   6.2173551556e-01   8.0596931719e-02
```

Figure 6.17: Results of an Epistemic Uncertainty Quantification using Evidence Theory.

be plausibility values and the corresponding response levels are calculated (see Plaus Resp Level in the table in Figure 6.17). We have recently added the capability to support generalized reliability mappings in the evidence methods. If the user specifies a generalized reliability level, it will be first converted to a probability, then interpreted as a belief and plausibility and the corresponding response levels will be calculated. Likewise, if response levels are specified, the corresponding belief and plausibility values will be mapped to bounds on the generalized reliability levels.

To elaborate on the differences between `global_evidence` and `local_evidence`: both of these methods take the Dempster-Shafer structures specified on the inputs and calculate a resulting Dempster-Shafer structure on the outputs (e.g. a cumulative belief and plausibility function). To calculate the belief and plausibility measures, it is necessary to calculate the minimum and maximum of the response function in each "interval cell combination." For example, in a two variable problem, if the first variable had three intervals and associated BPAs assigned and the second variable had two intervals and associated BPAs assigned, there would be 6 interval cells in total. In each of these six cells, one needs to identify a minimum and maximum value of the response function. This is easy to do if the function is monotonic in both variables, but in general it is not. We offer the capability to use local optimization methods to calculate these bounds: `local_evidence` allows the user to specify either `sqp` which is sequential quadratic programming, or `nip` which is a nonlinear interior point method. We also offer the capability to use global methods to assess these interval cell bounds. `global_evidence` allows the user to specify either `lhs`, which performs Latin Hypercube Sampling and takes the minimum and maximum of the samples within each cell as the bounds (no optimization is performed) or `ego`. In the case of `ego`, the efficient global optimization method is used to calculate bounds. The ego method is described in Section 9.4. Note that for a situation with many uncertain variables, each with a fairly complicated Dempster-Shafer structure described by many intervals, there will be a huge number of interval calls, and the overall process of performing Dempster-Shafer analysis will be extremely expensive. Reference [110] provides more details about the implementation of the optimization methods to perform Dempster-Shafer calculations, as well as comparisons on test problems.

6.6 Future Nondeterministic Methods

Uncertainty analysis methods under investigation for future inclusion into the DAKOTA framework include extensions to the stochastic expansion methods and sampling capabilities currently supported. Advanced "smart sampling" techniques such as bootstrap sampling (BS) and Markov chain Monte Carlo simulation (McMC) are being considered. We also have an active research focus on adaptive sparse grid methods, to more efficiently construct stochastic expansions. We are currently pursuing Bayesian methods, specifically focusing on Bayesian calibration, where a "prior distribution" on a parameter is updated through a Bayesian framework involving experimental data and a likelihood function.

Chapter 7

Optimization Capabilities

7.1 Overview

DAKOTA's optimization capabilities include a variety of gradient-based and nongradient-based optimization methods. Numerous packages are available, some of which are commercial packages, some of which are developed internally to Sandia, and some of which are free software packages from the open source community. The downloaded version of DAKOTA excludes the commercially developed packages but includes HOPSPACK, SCOLIB (formerly known as COLINY), CONMIN, JEGA, NCSUOpt, and OPT++. Interfaces to DOT, NPSOL, and NLPQL are provided with DAKOTA, but to use these commercial optimizers, the user must obtain a software license and the source code for these packages separately. The commercial software can then be compiled into DAKOTA by following DAKOTA's installation procedures (see notes in `Dakota/INSTALL`).

DAKOTA's input commands permit the user to specify two-sided nonlinear inequality constraints of the form $g_{L_i} \leq g_i(\mathbf{x}) \leq g_{U_i}$, as well as nonlinear equality constraints of the form $h_j(\mathbf{x}) = h_{t_j}$ (see also Section 1.4.1). Some optimizers (e.g., NPSOL, OPT++, JEGA) can handle these constraint forms directly, whereas other optimizers (e.g., HOPSPACK, DOT, CONMIN) require DAKOTA to perform an internal conversion of all constraints to one-sided inequality constraints of the form $g_i(\mathbf{x}) \leq 0$. In the latter case, the two-sided inequality constraints are treated as $g_i(\mathbf{x}) - g_{U_i} \leq 0$ and $g_{L_i} - g_i(\mathbf{x}) \leq 0$ and the equality constraints are treated as $h_j(\mathbf{x}) - h_{t_j} \leq 0$ and $h_{t_j} - h_j(\mathbf{x}) \leq 0$. The situation is similar for linear constraints: HOPSPACK, NPSOL, OPT++, and JEGA support them directly, whereas DOT and CONMIN do not. For linear inequalities of the form $a_{L_i} \leq \mathbf{a}_i^T\mathbf{x} \leq a_{U_i}$ and linear equalities of the form $\mathbf{a}_i^T\mathbf{x} = a_{t_j}$, the nonlinear constraint arrays in DOT and CONMIN are further augmented to include $\mathbf{a}_i^T\mathbf{x} - a_{U_i} \leq 0$ and $a_{L_i} - \mathbf{a}_i^T\mathbf{x} \leq 0$ in the inequality case and $\mathbf{a}_i^T\mathbf{x} - a_{t_j} \leq 0$ and $a_{t_j} - \mathbf{a}_i^T\mathbf{x} \leq 0$ in the equality case. Awareness of these constraint augmentation procedures can be important for understanding the diagnostic data returned from the DOT and CONMIN algorithms. Other optimizers fall somewhere in between. NLPQL supports nonlinear equality constraints $h_j(\mathbf{x}) = 0$ and nonlinear one-sided inequalities $g_i(\mathbf{x}) \geq 0$, but does not natively support linear constraints. Constraint mappings are used with NLPQL for both linear and nonlinear cases. Most SCOLIB methods now support two-sided nonlinear inequality constraints and nonlinear constraints with targets, but do not natively support linear constraints. Constraint augmentation is not currently used with SCOLIB since linear constraints will soon be supported natively.

When gradient and Hessian information is used in the optimization, derivative components are most commonly computed with respect to the active continuous variables, which in this case are the *continuous design variables*. This differs from parameter study methods (for which all continuous variables are active) and from nondeterministic analysis methods (for which the uncertain variables are active). Refer to Section 13.3 for additional

information on derivative components and active continuous variables.

7.2 Optimization Software Packages

7.2.1 HOPSPACK Library

The HOPSPACK library [91] provides access to Asynchronous Parallel Pattern Search (APPS) [56], which is available as method `asynch_pattern_search`. It is an asynchronous implementation of generating set search. APPS can handle unconstrained problems as well as those with bound constraints, linear constraints [58], and general nonlinear constraints [57]. APPS is best suited for simulation-based problems with less than 100 variables. It has significant advantages over gradient-based methods when the function is noisy and/or discontinuous. APPS and HOPSPACK leverage the cddlib software [40] to handle degeneracy in linear constraints.

An example specification for APPS with nonlinear constraints is:

```
method,
      asynch_pattern_search
         initial_delta = .5
         contraction_factor = 0.25
         threshold_delta = 1.e-4
         merit_function merit_max
         smoothing_factor = 10.0
```

See the DAKOTA Reference Manual [3] for additional detail on the APPS commands and sample input specifications in `Dakota/test/dakota_apps.in`.

7.2.2 SCOLIB Library

The SCOLIB library [63] contains a variety of nongradient-based optimization algorithms. The suite of SCOLIB optimizers available in DAKOTA currently include the following:

- **Global Optimization Methods**
  - Several evolutionary algorithms, including genetic algorithms (`coliny_ea`)
  - DIRECT [90] (`coliny_direct`)
- **Local Optimization Methods**
  - Solis-Wets (`coliny_solis_wets`)
  - Pattern Search (`coliny_pattern_search`)
- **Interfaces to Third-Party Local Optimization Methods**
  - COBYLA2 (`coliny_cobyla`)

For expensive optimization problems, SCOLIB's global optimizers are best suited for identifying promising regions in the global design space. In multimodal design spaces, the combination of global identification (from SCOLIB) with efficient local convergence (from CONMIN, DOT, NLPQL, NPSOL, or OPT++) can be highly

effective. None of the SCOLIB methods are gradient-based, which makes them appropriate for problems for which gradient information is unavailable or is of questionable accuracy due to numerical noise. The SCOLIB methods support bound constraints and nonlinear constraints, but not linear constraints. The nonlinear constraints in SCOLIB are currently satisfied using penalty function formulations [92]. Support for methods which manage constraints internally is currently being developed and will be incorporated into future versions of DAKOTA. *Note that one observed drawback to* `coliny_solis_wets` *is that it does a poor job solving problems with nonlinear constraints*. Refer to Table 17.1 for additional method classification information.

An example specification for a simplex-based pattern search algorithm from SCOLIB is:

```
method,
      coliny_pattern_search
        max_function_evaluations = 2000
        solution_accuracy = 1.0e-4
        initial_delta = 0.05
        threshold_delta = 1.0e-8
        pattern_basis simplex
        exploratory_moves adaptive_pattern
        contraction_factor = 0.75
```

The DAKOTA Reference Manual [3] contains additional information on the SCOLIB options and settings.

7.2.3 Constrained Minimization (CONMIN) Library

The CONMIN library [116] contains two methods for gradient-based nonlinear optimization. For constrained optimization, the Method of Feasible Directions (DAKOTA's `conmin_mfd` method selection) is available, while for unconstrained optimization, the Fletcher-Reeves conjugate gradient method (DAKOTA's `conmin_frcg` method selection) is available. Both of these methods are most efficient at finding a local minimum in the vicinity of the starting point. The methods in CONMIN can be applied to global optimization problems, but there is no guarantee that they will find the globally optimal design point.

One observed drawback to CONMIN's Method of Feasible Directions is that it does a poor job handling equality constraints. This is the case even if the equality constraint is formulated as two inequality constraints. This problem is what motivates the modifications to MFD that are present in DOT's MMFD algorithm. For problems with equality constraints, it is better to use the OPT++ nonlinear interior point methods, NPSOL, NLPQL, or one of DOT's constrained optimization methods (see below).

An example specification for CONMIN's Method of Feasible Directions algorithm is:

```
method,
      conmin_mfd
        convergence_tolerance = 1.0e-4
        max_iterations = 100
        output quiet
```

Refer to the DAKOTA Reference Manual [3] for more information on the settings that can be used with CONMIN methods.

7.2.4 Design Optimization Tools (DOT) Library

The DOT library [118] contains nonlinear programming optimizers, specifically the Broyden-Fletcher-Goldfarb-Shanno (DAKOTA's `dot_bfgs` method selection) and Fletcher-Reeves conjugate gradient (DAKOTA's `dot_frcg` method selection) methods for unconstrained optimization, and the modified method of feasible directions (DAKOTA's `dot_mmfd` method selection), sequential linear programming (DAKOTA's `dot_slp` method selection), and sequential quadratic programming (DAKOTA's `dot_sqp` method selection) methods for constrained optimization.

All DOT methods are local gradient-based optimizers which are best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the NPSOL, NLPQL, CONMIN, and OPT++ libraries.

Through the `optimization_type` specification, DOT can be used to solve either minimization or maximization problems. For all other optimizer libraries, it is up to the user to reformulate a maximization problem as a minimization problem by negating the objective function (i.e., maximize $f(x)$ is equivalent to minimize $-f(x)$). An example specification for DOT's BFGS quasi-Newton algorithm is:

```
method,
      dot_bfgs
        optimization_type maximize
        convergence_tolerance = 1.0e-4
        max_iterations = 100
        output quiet
```

We here provide a caution regarding `dot_frcg`. In DOT Version 4.20, we have noticed inconsistent behavior of this algorithm across different versions of Linux. Our best assessment is that it is due to different treatments of uninitialized variables. As we do not know the intention of the code authors and maintaining DOT source code is outside of the DAKOTA project scope, we have not made nor are we recommending any code changes to address this. However, all users who use `dot_frcg` in DOT Version 4.20 should be aware that results may not be reliable.

See the DAKOTA Reference Manual [3] for additional detail on the DOT commands. More information on DOT can be obtained by contacting Vanderplaats Research and Development at http://www.vrand.com.

7.2.5 dl_solver — Solvers via Shared Libraries

On computer systems that permit use of shared libraries (most modern systems), DAKOTA can avail itself of optimization solvers contained in shared libraries. This is a first step toward allowing optional parts of DAKOTA, such as proprietary solvers, to be accessed from shared libraries. For example, the DAKOTA source distributions illustrate making a sample shared-library interface to SNOPT [47], whose use would be specified by

```
method,
      dl_solver = 'dl_snopt.dll'
```

The quoted string contains the name of the shared library, optionally followed by keyword assignments known to the library, such as

```
method,
      dl_solver = 'dl_snopt.dll outlev = 1'
```

which would turn on some diagnostic printing in the SNOPT example.

7.2.6 JEGA

The JEGA (John Eddy's Genetic Algorithms) library contains two global optimization methods. The first is a Multi-objective Genetic Algorithm (MOGA) which performs Pareto optimization. The second is a Single-objective Genetic Algorithm (SOGA) which performs optimization on a single objective function. These functions are accessed as (`moga` and `soga`) within DAKOTA.

The `moga` algorithm directly creates a population of Pareto optimal solutions. Over time, the selection operators of a genetic algorithm act to efficiently select non-dominated solutions along the Pareto front. Because a GA involves a population of solutions, many points along the Pareto front can be computed in a single study. Thus, although GAs are computationally expensive when compared to gradient-based methods, the advantage in the multiobjective setting is that one can obtain an entire Pareto set at the end of one genetic algorithm run, as compared with having to run the "weighted sum" single objective problem multiple times with different weights.

The DAKOTA Reference Manual [3] contains additional information on the JEGA options and settings. Section 7.3 discusses additional multiobjective optimization capabilities, and there are MOGA examples in Chapters 2 and 23.

7.2.7 NCSU DIRECT

We have an implementation of the global optimization method called DIRECT (DIviding RECTangles algorithm) that is detailed in [41]. DIRECT is a derivative free global optimization method that balances local search in promising regions of the design space with global search in unexplored regions. DIRECT adaptively subdivides the space of feasible design points to guarantee that iterates are generated in the neighborhood of a global minimum in finitely many iterations. In practice, DIRECT has proven an effective heuristic for many applications.

NCSU DIRECT is specified with `ncsu_direct`. One of the controls is `volume_boxsize_limit`, which terminates the optimization when the volume of the particular rectangle which contains the minimum function value found thus far is less than a certain percentage (given by the volume boxsize limit) of the whole volume of the hyperrectangle defined by the variable bounds. An example specification is given below:

```
method,
      ncsu_direct
      volume_boxsize_limit = 1.e-8
```

The DAKOTA Reference Manual [3] contains additional information on the NCSU DIRECT options and settings.

7.2.8 NLPQL Library

The NLPQL library contains a sequential quadratic programming (SQP) implementation (DAKOTA's `nlpql_sqp` method selection). The particular implementation used is NLPQLP [99], a variant with distributed and non-monotone line search. SQP is a nonlinear programming approach for constrained minimization which solves a series of quadratic programming (QP) subproblems, where each QP minimizes a quadratic approximation to the Lagrangian subject to linearized constraints. It uses an augmented Lagrangian merit function and a BFGS approximation to the Hessian of the Lagrangian. It is an infeasible method in that constraints will be satisfied at the final solution, but not necessarily during the solution process. The non-monotone line search used in NLPQLP is designed to be more robust in the presence of inaccurate or noisy gradients common in many engineering applications.

NLPQL's gradient-based approach is best suited for efficient navigation to a local minimum in the vicinity of

the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the DOT, CONMIN, NPSOL, and OPT++ libraries.

See the DAKOTA Reference Manual [3] for additional detail on the NLPQL commands. More information on NLPQL can be obtained from Prof. Klaus Schittkowski at http://www.uni-bayreuth.de/departments/math/~kschittkowski/nlpqlp20.htm.

7.2.9 NPSOL Library

The NPSOL library [45] contains a sequential quadratic programming (SQP) implementation (DAKOTA's npsol_sqp method selection). Like NLPQL, it solves a series of QP subproblems, uses an augmented Lagrangian merit function and a BFGS approximation to the Hessian of the Lagrangian, and will not necessarily satisfy the constraints until the final solution. It uses a sufficient-decrease line search approach, which is a gradient-based line search for analytic, mixed, or DAKOTA-supplied numerical gradients and is a value-based line search in the vendor numerical case.

NPSOL's gradient-based approach is best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the DOT, CONMIN, NLPQL, and OPT++ libraries. For least squares methods based on NPSOL, refer to Section 8.2.2.

An example of an NPSOL specification is:

```
method,
      npsol_sqp
        convergence_tolerance = 1.0e-6
        max_iterations = 100
        output quiet
```

See the DAKOTA Reference Manual [3] for additional detail on the NPSOL commands. More information on NPSOL can be obtained by contacting Stanford Business Software at http://www.sbsi-sol-optimize.com.

The NPSOL library generates diagnostics in addition to those appearing in the DAKOTA output stream. These diagnostics are written to the default FORTRAN device 9 file (e.g., ftn09 or fort.9, depending on the architecture) in the working directory.

7.2.10 OPT++ Library

The OPT++ library [81] contains primarily nonlinear programming optimizers for unconstrained, bound constrained, and nonlinearly constrained minimization: Polak-Ribiere conjugate gradient (DAKOTA's optpp_cg method selection), quasi-Newton (DAKOTA's optpp_q_newton method selection), finite difference Newton (DAKOTA's optpp_fd_newton method selection), and full Newton (DAKOTA's optpp_newton method selection). The library also contains the parallel direct search nongradient-based method [19] (specified as DAKOTA's optpp_pds method selection).

OPT++'s gradient-based optimizers are best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. OPT++'s PDS method does not use gradients and has some limited global identification abilities; it is best suited for problems for which gradient information is unavailable or is of questionable accuracy due to numerical noise. Some OPT++ methods are

strictly unconstrained (optpp_cg) and some support bound constraints (optpp_pds), whereas the Newton-based methods (optpp_q_newton, optpp_fd_newton, and optpp_newton) all support general linear and nonlinear constraints (refer to Table 20.1). Other gradient-based optimizers include the DOT, CONMIN, NLPQL, and NPSOL libraries. For least squares methods based on OPT++, refer to Section 8.2.1.

An example specification for the OPT++ quasi-Newton algorithm is:

```
method,
      optpp_q_newton
        max_iterations = 50
        convergence_tolerance = 1e-4
        output debug
```

See the DAKOTA Reference Manual [3] for additional detail on the OPT++ commands.

The OPT++ library generates diagnostics in addition to those appearing in the DAKOTA output stream. These diagnostics are written to the file OPT_DEFAULT.out in the working directory.

7.3 Additional Optimization Capabilities

DAKOTA provides several capabilities which extend the services provided by the optimization software packages described in Section 7.2. First, any of the optimization algorithms can be used for multiobjective optimization problems through the use of multiobjective transformation techniques (e.g., weighted sums). Finally, with any optimizer (or least squares solver described in Section 8.2), user-specified (and in some cases automatic or logarithmic) scaling may be applied to continuous design variables, objective functions (or least squares terms), and constraints.

7.3.1 Multiobjective Optimization

Multiobjective optimization means that there are two or more objective functions that you wish to optimize simultaneously. Often these are conflicting objectives, such as cost and performance. The answer to a multi-objective problem is usually not a single point. Rather, it is a set of points called the Pareto front. Each point on the Pareto front satisfies the Pareto optimality criterion, which is stated as follows: a feasible vector X^* is Pareto optimal if there exists no other feasible vector X which would improve some objective without causing a simultaneous worsening in at least one other objective. Thus, if a feasible point X' exists that CAN be improved on one or more objectives simultaneously, it is not Pareto optimal: it is said to be "dominated" and the points along the Pareto front are said to be "non-dominated."

There are three capabilities for multiobjective optimization in DAKOTA. First, there is the MOGA capability described previously in Section 7.2.6. This is a specialized algorithm capability. The second capability involves the use of response data transformations to recast a multiobjective problem as a single-objective problem. Currently, DAKOTA supports the simple weighted sum approach for this transformation, in which a composite objective function is constructed from a set of individual objective functions using a user-specified set of weighting factors. This approach is optimization algorithm independent, in that it works with any of the optimization methods listed previously in this chapter. The third capability is the Pareto-set optimization strategy described in Section 16.4. This capability also utilizes the multiobjective response data transformations to allow optimization algorithm independence; however, it builds upon the basic approach by computing sets of optima in order to generate a Pareto trade-off surface.

In the multiobjective transformation approach in which multiple objectives are combined into one, an appropriate single-objective optimization technique is used to solve the problem. The advantage of this approach is that one can use any number of optimization methods that are especially suited for the particular problem class. One disadvantage of the weighted sum transformation approach is that a linear weighted sum objective cannot locate all optimal solutions in the Pareto set if the Pareto front is nonconvex. Also, if one wants to understand the effects of changing weights, this method can become computationally expensive. Since each optimization of a single weighted objective will find only one point near or on the Pareto front, many optimizations need to be performed to get a good parametric understanding of the influence of the weights.

The selection of a multiobjective optimization problem is made through the specification of multiple objective functions in the responses keyword block (i.e., the `num_objective_functions` specification is greater than `1`). The weighting factors on these objective functions can be optionally specified using the `multi_objective_weights` keyword (the default is equal weightings). The composite objective function for this optimization problem, F, is formed using these weights as follows: $F = \sum_{k=1}^{R} w_k f_k$, where the f_k terms are the individual objective function values, the w_k terms are the weights, and R is the number of objective functions. The weighting factors stipulate the relative importance of the design concerns represented by the individual objective functions; the higher the weighting factor, the more dominant a particular objective function will be in the optimization process. Constraints are not affected by the weighting factor mapping; therefore, both constrained and unconstrained multiobjective optimization problems can be formulated and solved with DAKOTA, assuming selection of an appropriate constrained or unconstrained single-objective optimization algorithm. Future multiobjective response data transformations for goal programming, normal boundary intersection, etc. are planned.

Figure 7.1 shows a DAKOTA input file for a multiobjective optimization problem based on the "textbook" test problem. This input file is named `dakota_multiobj1.in` in the `Dakota/test` directory. In the standard textbook formulation, there is one objective function and two constraints. In the multiobjective textbook formulation, all three of these functions are treated as objective functions (`num_objective_functions = 3`), with weights given by the `multi_objective_weights` keyword. Note that it is not required that the weights sum to a value of one. The multiobjective optimization capability also allows any number of constraints, although none are included in this example.

Figure 7.2 shows an excerpt of the results for this multiobjective optimization problem, with output in verbose mode. The data for function evaluation 9 show that the simulator is returning the values and gradients of the three objective functions and that this data is being combined by DAKOTA into the value and gradient of the composite objective function, as identified by the header "`Multiobjective transformation:`". This combination of value and gradient data from the individual objective functions employs the user-specified weightings of `.7`, `.2`, and `.1`. Convergence to the optimum of the multiobjective problem is indicated in this case by the gradient of the composite objective function going to zero (no constraints are active).

By performing multiple optimizations for different sets of weights, a family of optimal solutions can be generated which define the trade-offs that result when managing competing design concerns. This set of solutions is referred to as the Pareto set. Section 16.4 describes a solution strategy used for directly generating the Pareto set in order to investigate the trade-offs in multiobjective optimization problems.

7.3.2 Optimization with User-specified or Automatic Scaling

Some optimization problems involving design variables, objective functions, or constraints on vastly different scales may be solved more efficiently if these quantities are adjusted to a common scale (typically on the order of unity). With any optimizer (or least squares solver described in Section 8.2), user-specified characteristic value scaling may be applied to any of continuous design variables, functions/residuals, nonlinear inequality and equality constraints, and linear inequality and equality constraints. Automatic scaling is available for variables or responses with one- or two-sided bounds or equalities and may be combined with user-specified scaling values.

```
strategy,
        single_method
        tabular_graphics_data

method,
        npsol_sqp
          output verbose
          convergence_tolerance = 1.e-8

variables,
        continuous_design = 2
          initial_point    0.9    1.1
          upper_bounds     5.8    2.9
          lower_bounds     0.5   -2.9
          descriptors      'x1'   'x2'

interface,
        system asynchronous
          analysis_driver=  'text_book'

responses,
        objective_functions = 3
        multi_objective_weights = .7 .2 .1
        analytic_gradients
        no_hessians
```

Figure 7.1: Example DAKOTA input file for multiobjective optimization.

```
------------------------------
Begin Function Evaluation    9
------------------------------
Parameters for function evaluation 9:
                      5.9388064483e-01 x1
                      7.4158741198e-01 x2

(text_book /tmp/fileFNNH3v /tmp/fileRktLe9)
Removing /tmp/fileFNNH3v and /tmp/fileRktLe9

Active response data for function evaluation 9:
Active set vector = { 3 3 3 } Deriv vars vector = { 1 2 }
                      3.1662048106e-02 obj_fn_1
                     -1.8099485683e-02 obj_fn_2
                      2.5301156719e-01 obj_fn_3
 [ -2.6792982175e-01 -6.9024137415e-02 ] obj_fn_1 gradient
 [  1.1877612897e+00 -5.0000000000e-01 ] obj_fn_2 gradient
 [ -5.0000000000e-01  1.4831748240e+00 ] obj_fn_3 gradient



-----------------------------------
Post-processing Function Evaluation
-----------------------------------
Multiobjective transformation:
                      4.3844693257e-02 obj_fn
 [  1.3827084219e-06  5.8620632776e-07  ] obj_fn gradient

    7    1 1.0E+00    9  4.38446933E-02 1.5E-06    2 T TT

 Exit NPSOL - Optimal solution found.

 Final nonlinear objective value =   0.4384469E-01
```

Figure 7.2: DAKOTA results for the multiobjective optimization example.

Logarithmic ($\log_{10}$) scaling is available and may also be combined with characteristic values. Log scaling is not available for linear constraints. Moreover, when continuous design variables are log scaled, linear constraints are not permitted in the problem formulation. Discrete variable scaling is not supported.

Scaling is enabled on a per-method basis for optimizers and least squares minimizers by including the `scaling` keyword in the relevant `method` specification in the DAKOTA input deck. When scaling is enabled, variables, functions, gradients, Hessians, etc., are transformed such that the optimizer iterates in scaled variable space, whereas evaluations of the computational model as specified in the interface are performed on the original problem scale. Therefore using scaling does not require rewriting the interface to the simulation code. When the `scaling` keyword is omitted, all `*_scale_types` and `*_scales` specifications described below are ignored in the corresponding method, variables, and responses sections. When the method `output_level` is set above normal, scaling initialization and diagnostic information will be printed.

Scaling for a particular variable or response type is enabled through the `*_scale_types` specification (see the Reference Manual method section and references contained therein for a complete keyword list). Valid options for this string specification include `'none'` (default), `'value'`, `'auto'`, or `'log'`, for no, characteristic value, automatic, or logarithmic scaling, respectively (although not all types are valid for scaling all entities). If a single string is specified with any of these keywords it will apply to each component of the relevant vector, e.g., `cdv_scale_types = 'value'` will enable characteristic value scaling for each continuous design variable.

The user may additionally specify no, one, or a vector of characteristic scale values through the `*_scales` specification. These characteristic values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'` and `'log'`. If a single value is specified with any of these keywords it will apply to each component of the relevant vector, e.g., `cdv_scales = 3.0` will apply a characteristic scaling value of 3.0 to each continuous design variable.

When scaling is enabled, the following procedures determine the transformations used to scale each component of a variables or response vector. A warning is issued if scaling would result in division by a value smaller in magnitude than `1.0e10*DBL_MIN`. User-provided values violating this lower bound are accepted unaltered, whereas for automatically calculated scaling, the lower bound is enforced.

- None (`'none'`): no scaling performed (`*_scales` ignored) on this component.
- Characteristic value (`'value'`): the corresponding quantity is scaled (divided) by the required characteristic value provided in the `*_scales` specification, and bounds are adjusted as necessary. If the value is negative, the sense of inequalities are changed accordingly.
- Automatic (`'auto'`): First, any characteristic values from the optional `*_scales` specification are applied. Then, automatic scaling will be attempted according to the following scheme:
  - two-sided bounds scaled into the interval [0,1];
  - one-sided bounds or targets are scaled by a characteristic value to move the bound or target to 1, and the sense of inequalities are changed if necessary;
  - no bounds or targets: no automatic scaling possible for this component

  Automatic scaling is not available for objective functions nor least squares terms since they lack bound constraints. Further, when automatically scaled, linear constraints are scaled by characteristic values only, not affinely scaled into [0,1].
- Logarithmic (`'log'`): First, any characteristic values from the optional `*_scales` specification are applied. Then, $\log_{10}$ scaling is applied. Logarithmic scaling is not available for linear constraints. Further, when continuous design variables are log scaled, linear constraints are not allowed.

Scaling for linear constraints specified through `linear_inequality_scales` or `linear_equality_scales` is applied *after* any (user-specified or automatic) continuous variable scaling. For example, for scaling mapping unscaled continuous design variables x to scaled variables $\tilde{x}$:

$$\tilde{x}^j = \frac{x^j - x_O^j}{x_M^j},$$

where x_M^j is the final component multiplier and x_O^j the offset, we have the following matrix system for linear inequality constraints

$$\begin{gathered} a_L \leq A_i x \leq a_U \\ a_L \leq A_i \left(\mathrm{diag}(x_M)\tilde{x} + x_O\right) \leq a_U \\ a_L - A_i x_O \leq A_i \mathrm{diag}(x_M)\tilde{x} \leq a_U - A_i x_O \\ \tilde{a}_L \leq \tilde{A}_i \tilde{x} \leq \tilde{a}_U, \end{gathered}$$

and user-specified or automatically computed scaling multipliers are applied to this final transformed system, which accounts for any continuous design variable scaling. When automatic scaling is in use for linear constraints they are linearly scaled by characteristic values only, not affinely scaled into the interval $[0, 1]$.

Figure 7.3 demonstrates the use of several scaling keywords for the textbook optimization problem.This input file is `dakota_rosenbrock_scaled.in` in the `Dakota/examples/methods` directory. The continuous design variable `x1` is scaled by a characteristic value of 4.0, whereas `x2` is scaled automatically into $[0, 1]$ based on its bounds. The objective function will be scaled by a factor of 50.0, then logarithmically, the first nonlinear constraint by a factor of 15.0, and the second nonlinear constraint is not scaled.

```
strategy,
        single_method
          graphics,tabular_graphics_data

method,
        conmin_frcg
          scaling
          output verbose

model,
        single

variables,
        continuous_design = 2
          initial_point   -1.2  1.0
          lower_bounds    -2.0      0.001
          upper_bounds     2.0      2.0
          descriptors      'x1'     "x2"
          scale_types = 'value' 'log'
          scales = 4.0 0.1

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
          objective_function_scale_types = 'value'
          objective_function_scales = 50.0
        analytic_gradients
        no_hessians
```

Figure 7.3: Sample usage of scaling keywords in DAKOTA input specification.

Chapter 8

Nonlinear Least Squares Capabilities

8.1 Overview

Nonlinear least-squares methods are optimization algorithms that exploit the special structure of a sum of the squares objective function [46]. These problems commonly arise in parameter estimation, system identification, and test/analysis reconciliation. To exploit the problem structure, more granularity is needed in the response data than is required for a typical optimization problem. That is, rather than using the sum-of-squares objective function and its gradient, least-squares iterators require each term used in the sum-of-squares formulation along with its gradient. This means that the m functions in the DAKOTA response data set consist of the individual least-squares terms along with any nonlinear inequality and equality constraints. These individual terms are often called *residuals* when they denote differences of observed quantities from values computed by the model whose parameters are being estimated.

The enhanced granularity needed for nonlinear least-squares algorithms allows for simplified computation of an approximate Hessian matrix. In Gauss-Newton-based methods for example, the true Hessian matrix is approximated by neglecting terms in which residuals multiply Hessians (matrices of second partial derivatives) of residuals, under the assumption that the residuals tend towards zero at the solution. As a result, residual function value and gradient information (first-order information) is sufficient to define the value, gradient, and approximate Hessian of the sum-of-squares objective function (second-order information). See Section 1.4.2 for additional details on this approximation.

In practice, least-squares solvers will tend to be significantly more efficient than general-purpose optimization algorithms when the Hessian approximation is a good one, e.g., when the residuals tend towards zero at the solution. Specifically, they can exhibit the quadratic convergence rates of full Newton methods, even though only first-order information is used. Gauss-Newton-based least-squares solvers may experience difficulty when the residuals at the solution are significant.

In order to specify a least-squares problem, the responses section of the DAKOTA input should be configured using `calibration_terms` (as opposed to `num_objective_functions` in the case of optimization). The calibration terms refer to the residuals (e.g. typically the differences between the simulation model and the data). Note that DAKOTA expects the residuals and not the square of the residuals. Any linear or nonlinear constraints are handled in an identical way to that of optimization (see Section 7.1; note that neither Gauss-Newton nor NLSSOL require any constraint augmentation and NL2SOL supports neither linear nor nonlinear constraints). Gradients of the least-squares terms and nonlinear constraints are required and should be specified using either `numerical_gradients`, `analytic_gradients`, or `mixed_gradients`. Since ex-

plicit second derivatives are not used by the least-squares methods, the `no_hessians` specification should be used. DAKOTA's scaling options, described in Section 7.3.2 can be used on least-squares problems, using the `calibration_term_scales` keyword to scale least-squares residuals, if desired.

8.2 Solution Techniques

Nonlinear least-squares problems can be solved using the Gauss-Newton algorithm, which leverages the full Newton method from OPT++, the NLSSOL algorithm, which is closely related to NPSOL, or the NL2SOL algorithm, which uses a secant-based algorithm. Details for each are provided below.

8.2.1 Gauss-Newton

DAKOTA's Gauss-Newton algorithm consists of combining an implementation of the Gauss-Newton Hessian approximation (see Section 1.4.2) with full Newton optimization algorithms from the OPT++ package [81] (see Section 7.2.10). This approach can be selected using the `optpp_g_newton` method specification. An example specification follows:

```
method,
      optpp_g_newton
        max_iterations = 50
        convergence_tolerance = 1e-4
        output debug
```

Refer to the DAKOTA Reference Manual [3] for more detail on the input commands for the Gauss-Newton algorithm.

The Gauss-Newton algorithm is gradient-based and is best suited for efficient navigation to a local least-squares solution in the vicinity of the initial point. Global optima in multimodal design spaces may be missed. Gauss-Newton supports bound, linear, and nonlinear constraints. For the nonlinearly-constrained case, constraint Hessians (required for full-Newton nonlinear interior point optimization algorithms) are approximated using quasi-Newton secant updates. Thus, both the objective and constraint Hessians are approximated using first-order information.

8.2.2 NLSSOL

The NLSSOL algorithm is a commercial software product of Stanford University that is bundled with current versions of the NPSOL library (see Section 7.2.9). It uses an SQP-based approach to solve generally-constrained nonlinear least-squares problems. It periodically employs the Gauss-Newton Hessian approximation to accelerate the search. Like the Gauss-Newton algorithm of Section 8.2.1, its derivative order is balanced in that it requires only first-order information for the least-squares terms and nonlinear constraints. This approach can be selected using the `nlssol_sqp` method specification. An example specification follows:

```
method,
      nlssol_sqp
        convergence_tolerance = 1e-8
```

Refer to the DAKOTA Reference Manual [3] for more detail on the input commands for NLSSOL.

8.2.3 NL2SOL

The NL2SOL algorithm [17] is a secant-based least-squares algorithm that is q-superlinearly convergent. It adaptively chooses between the Gauss-Newton Hessian approximation and this approximation augmented by a correction term from a secant update. NL2SOL is appropriate for "large residual" problems, i.e., least-squares problems for which the residuals do not tend towards zero at the solution.

8.2.4 Additional Features and Future plans

DAKOTA can calculate confidence intervals on estimated parameters. These are determined for individual parameters; they are not joint confidence intervals. The intervals reported are 95% intervals around the estimated parameters, and are calculated as the optimal value of the estimated parameters $+/-$ a t-test statistic times the standard error (SE) of the estimated parameter vector. The SE is based on a linearization approximation involving the matrix of the derivatives of the model with respect to the derivatives of the estimated parameters. In the case where these gradients are extremely inaccurate or the model is very nonlinear, the confidence intervals reported are likely to be inaccurate as well. Future work on generating confidence intervals on the estimated parameters for nonlinear least-squares methods will involve adding Bonferroni confidence intervals and one or two methods for calculating joint confidence intervals (such as a linear approximation and the F-test method). See [101] and [119] for more details about confidence intervals. Note that confidence intervals are not calculated when scaling is used, when the number of least-squares terms is less than the number of parameters to be estimated, or when using numerical gradients.

DAKOTA also allows a form of weighted least squares. The user can specify a set of weights that are used to weight each residual term using the keyword `calibration_weights`. Note that these weights must be predetermined by the user and entered in the DAKOTA input file: they are not calculated on-the-fly. The user can also specify scaling for the least-squares terms. Scaling is applied before weighting; usually one or the other would be applied but not both. The Responses section in the DAKOTA Reference Manual [3] has more detail about weighting and scaling of the residual terms.

The least-squares branch in DAKOTA is an area of continuing enhancements, particularly through the addition of new least-squares algorithms. One potential future addition is the orthogonal distance regression (ODR) algorithms which estimate values for both independent and dependent parameters.

8.3 Examples

Both the Rosenbrock and textbook example problems can be formulated as nonlinear least-squares problems. Refer to Chapter 23 for more information on these formulations.

Figure 8.1 shows an excerpt from the output obtained when running NL2SOL on a five-dimensional problem. This input file is named `dakota_nl2test.in` found in the `Dakota/test` directory. Note that the optimal parameter estimates are printed, followed by the residual norm and values of the individual residual terms, followed by the confidence intervals on the parameters.

The analysis driver script (the script being driven by DAKOTA) has to perform several tasks in the case of parameter estimation using nonlinear least-squares methods. The analysis driver script must: (1) read in the values of the parameters supplied by DAKOTA; (2) run the computer simulation with these parameter values; (3) retrieve the results from the computer simulation; (4) compute the difference between each computed simulation value and the corresponding experimental or measured value; and (5) write these residuals (differences) to an external file that gets passed back to DAKOTA. Note there will be one line per residual term, specified with

```
<<<<< Iterator nl2sol completed.
<<<<< Function evaluation summary: 27 total (26 new, 1 duplicate)
<<<<< Best parameters          =
                     3.7541004764e-01 x1
                     1.9358463401e+00 x2
                    -1.4646865611e+00 x3
                     1.2867533504e-02 x4
                     2.2122702030e-02 x5
<<<<< Best residual norm =  7.3924926090e-03; 0.5 * norm^2 =  2.7324473487e-05
<<<<< Best residual terms      =
                    -2.5698266189e-03
                     4.4759880011e-03
                     9.9223430643e-04
                    -1.0634409194e-03

...

Confidence Interval for x1 is [  3.7116510206e-01,  3.7965499323e-01 ]
Confidence Interval for x2 is [  1.4845485507e+00,  2.3871441295e+00 ]
Confidence Interval for x3 is [ -1.9189348458e+00, -1.0104382765e+00 ]
Confidence Interval for x4 is [  1.1948590669e-02,  1.3786476338e-02 ]
Confidence Interval for x5 is [  2.0289951664e-02,  2.3955452397e-02 ]
```

Figure 8.1: Example of confidence intervals on optimal parameters

`num_least_squares_terms` in the DAKOTA input file. It is the last two steps which are different from most other DAKOTA applications.

To simplify specifying a least squares problem, a user may specify a data file containing experimental results or other calibration data. This file may be specified with `calibration_data_file`. In this case, DAKOTA will calculate the residuals (that is, the simulation model results minus the experimental results), and the user-provided script can omit this step: the script can just return the simulation outputs of interest. An example of this can be found in the file named `dakota_nls_datafile.in` in the `Dakota/examples/methods` directory. In this example, there are 3 residual terms. The data file of experimental results associated with this example is `least_squares_test.dat`. These three values are subtracted from the least-squares terms to produce residuals for the nonlinear least-squares problem. Note that the file may be annotated (specified by `annotated`) or freeform (specified by `freeform`). The number of experiments in the calibration data file may be specified with `num_experiments`, with one row of data per experiment. Finally, this data file may contain additional information than just the observed experimental responses. If the observed data has measurement error associated with it, this can be specified in columns of such error data after the response data. The number of calibration terms which have associated error in the data set is given by `num_std_deviations`. Additionally, there is sometimes the need to specify configuration variables. These are often used in Bayesian calibration analysis. These are specified as `num_config_variables`. If the user specifies a positive number of configuration variables, it is expected that they will occur in the text file before the responses.

Chapter 9

Surrogate-Based Minimization

9.1 Overview

Surrogate models approximate an original, high fidelity "truth" model, typically at reduced computational cost. In DAKOTA, several surrogate model selections are possible, which are categorized as data fits, multifidelity models, and reduced-order models, as described in Section 10.4. In the context of minimization (optimization or calibration), surrogate models can speed convergence by reducing function evaluation cost or smoothing noisy response functions.

9.2 Surrogate-Based Local Minimization

In the surrogate-based local minimization method (keyword: `surrogate_based_local`) the minimization algorithm operates on a surrogate model instead of directly operating on the computationally expensive simulation model. The surrogate model can be based on data fits, multifidelity models, or reduced-order models, as described in Section 10.4. Since the surrogate will generally have a limited range of accuracy, the surrogate-based local algorithm periodically checks the accuracy of the surrogate model against the original simulation model and adaptively manages the extent of the approximate optimization cycles using a trust region approach.

Refer to the DAKOTA Theory Manual [2] for algorithmic details on iterate acceptance, merit function formulations, convergence assessment, and constraint relaxation.

9.2.1 SBO with Data Fits

When performing SBO with local, multipoint, and global data fit surrogates, it is necessary to regenerate or update the data fit for each new trust region. In the global data fit case, this can mean performing a new design of experiments on the original high-fidelity model for each trust region, which can effectively limit the approach to use on problems with, at most, tens of variables. Figure 9.1 displays this case. However, an important benefit of the global sampling is that the global data fits can tame poorly-behaved, nonsmooth, discontinuous response variations within the original model into smooth, differentiable, easily navigated surrogates. This allows SBO with global data fits to extract the relevant global design trends from noisy simulation data.

When enforcing local consistency between a global data fit surrogate and a high-fidelity model at a point, care must be taken to balance this local consistency requirement with the global accuracy of the surrogate. In particular, performing a correction on an existing global data fit in order to enforce local consistency can skew the data fit and destroy its global accuracy. One approach for achieving this balance is to include the consistency requirement within the data fit process by constraining the global data fit calculation (e.g., using constrained linear least squares). This allows the data fit to satisfy the consistency requirement while still addressing global accuracy with its remaining degrees of freedom. Embedding the consistency within the data fit also reduces the sampling requirements. For example, a quadratic polynomial normally requires at least $(n+1)(n+2)/2$ samples for n variables to perform the fit. However, with an embedded first-order consistency constraint at a single point, the minimum number of samples is reduced by $n+1$ to $(n^2+n)/2$.

Figure 9.1: SBO iteration progression for global data fits.

In the local and multipoint data fit cases, the iteration progression will appear as in Fig. 9.3. Both cases involve a single new evaluation of the original high-fidelity model per trust region, with the distinction that multipoint approximations reuse information from previous SBO iterates. Like model hierarchy surrogates, these techniques scale to larger numbers of design variables. Unlike model hierarchy surrogates, they generally do not require surrogate corrections, since the matching conditions are embedded in the surrogate form (as discussed for the global Taylor series approach above). The primary disadvantage to these surrogates is that the region of accuracy tends to be smaller than for global data fits and multifidelity surrogates, requiring more SBO cycles with smaller trust regions. More information on the design of experiments methods is available in Chapter 5, and the data fit surrogates are described in Section 10.4.1.

Figure 9.2 shows a DAKOTA input file that implements surrogate-based optimization on Rosenbrock's function. This input file is named `dakota_sbo_rosen.in` in the `Dakota/test` directory. The first method keyword block contains the SBO keyword `surrogate_based_local`, plus the commands for specifying the trust region size and scaling factors. The optimization portion of SBO, using the CONMIN Fletcher-Reeves conjugate gradient method, is specified in the following keyword blocks for `method`, `model`, `variables`, and `responses`. The model used by the optimization method specifies that a global surrogate will be used to map variables into responses (no `interface` specification is used by the surrogate model). The global surrogate is constructed using a DACE method which is identified with the `'SAMPLING'` identifier. This data sampling portion of SBO is specified in the final set of keyword blocks for `method`, `model`, `interface`, and `responses` (the earlier `variables` specification is reused). This example problem uses the Latin hypercube sampling method in the LHS software to select 10 design points in each trust region. A single surrogate model is constructed for the objective function using a quadratic polynomial. The initial trust region is centered at the design point $(x_1, x_2) = (-1.2, 1.0)$, and extends ± 0.4 (10% of the global bounds) from this point in the x_1 and x_2 coordinate directions.

If this input file is executed in DAKOTA, it will converge to the optimal design point at $(x_1, x_2) = (1, 1)$ in approximately 800 function evaluations. While this solution is correct, it is obtained at a much higher cost than a traditional gradient-based optimizer (e.g., see the results obtained from `dakota_rosenbrock.in`). This demonstrates that the SBO method with global data fits is not really intended for use with smooth continuous optimization problems; direct gradient-based optimization can be more efficient for such applications. Rather, SBO with global data fits is best-suited for the types of problems that occur in engineering design where the response quantities may be discontinuous, nonsmooth, or may have multiple local optima [49]. In these types of engineering design problems, traditional gradient-based optimizers often are ineffective, whereas global data fits can extract the global trends of interest despite the presence of local nonsmoothness (for an example problem with multiple local optima, look in `Dakota/test` for the file `dakota_sbo_sine_fcn.in` [50]).

```
strategy,
        single_method
        tabular_graphics_data
        method_pointer = 'SBLO'

method,
        id_method = 'SBLO'
        surrogate_based_local
        model_pointer = 'SURROGATE'
        approx_method_pointer = 'NLP'
        max_iterations = 500
        trust_region
          initial_size = 0.10
          minimum_size = 1.0e-6
          contract_threshold = 0.25
          expand_threshold   = 0.75
          contraction_factor = 0.50
          expansion_factor   = 1.50

method,
        id_method = 'NLP'
        conmin_frcg,
          max_iterations = 50,
          convergence_tolerance = 1e-8

model,
        id_model = 'SURROGATE'
        surrogate global
          responses_pointer = 'SURROGATE_RESP'
          dace_method_pointer = 'SAMPLING'
          correction additive zeroth_order
          polynomial quadratic

variables,
        continuous_design = 2
          initial_point   -1.2  1.0
          lower_bounds    -2.0     -2.0
          upper_bounds     2.0  2.0
          descriptors      'x1'     'x2'

responses,
        id_responses = 'SURROGATE_RESP'
        objective_functions = 1
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-6
        no_hessians

method,
        id_method = 'SAMPLING'
        model_pointer = 'TRUTH'
        sampling
          samples = 10
          seed = 531 rng rnum2
          sample_type lhs
          all_variables

model,
        id_model = 'TRUTH'
        single
          interface_pointer = 'TRUE_FN'
          responses_pointer = 'TRUE_RESP'

interface,
        direct
        id_interface = 'TRUE_FN'
          analysis_driver = 'rosenbrock'
          deactivate evaluation_cache restart_file

responses,
        id_responses = 'TRUE_RESP'
        objective_functions = 1
        no_gradients
        no_hessians
```

Figure 9.2: DAKOTA input file for the surrogate-based local optimization example.

The surrogate-based local minimizer is only mathematically guaranteed to find a local minimum. However, in practice, SBO can often find the global minimum. Due to the random sampling method used within the SBO algorithm, the SBO method will solve a given problem a little differently each time it is run (unless the user specifies a particular random number seed in the dakota input file as is shown in Figure 9.2). Our experience on the quasi-sine function mentioned above is that if you run this problem 10 times with the same starting conditions but different seeds, then you will find the global minimum in about 70-80% of the trials. This is good performance for what is mathematically only a local optimization method.

9.2.2 SBO with Multifidelity Models

When performing SBO with model hierarchies, the low-fidelity model is normally fixed, requiring only a single high-fidelity evaluation to compute a new correction for each new trust region. Figure 9.3 displays this case. This renders the multifidelity SBO technique more scalable to larger numbers of design variables since the number of high-fidelity evaluations per iteration (assuming no finite differencing for derivatives) is independent of the scale of the design problem. However, the ability to smooth poorly-behaved response variations in the high-fidelity model is lost, and the technique becomes dependent on having a well-behaved low-fidelity model[1]. In addition, the parameterizations for the low and high-fidelity models may differ, requiring the use of a mapping between these parameterizations. Space mapping, corrected space mapping, POD mapping, and hybrid POD space mapping are being explored for this purpose [93, 94].

When applying corrections to the low-fidelity model, there is no concern for balancing global accuracy with the local consistency requirements. However, with only a single high-fidelity model evaluation at the center of each trust region, it is critical to use the best correction possible on the low-fidelity model in order to achieve rapid convergence rates to the optimum of the high-fidelity model [26].

Figure 9.3: SBO iteration progression for model hierarchies.

A multifidelity test problem named `dakota_sbo_hierarchical.in` is available in `Dakota/test` to demonstrate this SBO approach. This test problem uses the Rosenbrock function as the high fidelity model and a function named "lf_rosenbrock" as the low fidelity model. Here, lf_rosenbrock is a variant of the Rosenbrock function (see `Dakota/test/lf_rosenbrock.C` for formulation) with the minimum point at $(x_1, x_2) = (0.80, 0.44)$, whereas the minimum of the original Rosenbrock function is $(x_1, x_2) = (1, 1)$. Multifidelity SBO locates the high-fidelity minimum in 11 high fidelity evaluations for additive second-order corrections and in 208 high fidelity evaluations for additive first-order corrections, but fails for zeroth-order additive corrections by converging to the low-fidelity minimum.

9.2.3 SBO with Reduced Order Models

When performing SBO with reduced-order models (ROMs), the ROM is mathematically generated from the high-fidelity model. A critical issue in this ROM generation is the ability to capture the effect of parametric changes within the ROM. Two approaches to parametric ROM are extended ROM (E-ROM) and spanning ROM (S-ROM) techniques [123]. Closely related techniques include tensor singular value decomposition (SVD) methods [75]. In

[1] It is also possible to use a hybrid data fit/multifidelity approach in which a smooth data fit of a noisy low fidelity model is used in combination with a high fidelity model

the single-point and multipoint E-ROM cases, the SBO iteration can appear as in Fig. 9.3, whereas in the S-ROM, global E-ROM, and tensor SVD cases, the SBO iteration will appear as in Fig. 9.1. In addition to the high-fidelity model analysis requirements, procedures for updating the system matrices and basis vectors are also required.

Relative to data fits and multifidelity models, ROMs have some attractive advantages. Compared to data fits such as regression-based polynomial models, they are more physics-based and would be expected to be more predictive (e.g., in extrapolating away from the immediate data). Compared to multifidelity models, ROMS may be more practical in that they do not require multiple computational models or meshes which are not always available. The primary disadvantage is potential invasiveness to the simulation code for projecting the system using the reduced basis.

9.3 Surrogate-Based Global Minimization

Surrogate-based global minimization differs from the surrogate-based local minimization approach discussed in the previous section in several ways: it is not a trust-region approach; initially there is one surrogate constructed over a set of sample points and the optimizer operates on that surrogate (as opposed to adaptively selecting points and re-building a surrogate in each trust region); and there is no guarantee of convergence.

The `surrogate_based_global` method was developed to address two needs. The first is the case where a user wishes to use existing function evaluations or a fixed sample size (perhaps based on computational cost and allocation of resources) to build a surrogate once and optimize on it. In this case (a single global optimization on a surrogate model), the set of surrogate building points is determined in advance as opposed to the trust-region local surrogate optimization in which the number of "true" function evaluations depends on the location and size of the trust region, the goodness of the surrogate within the trust-region, and problem characteristics.

In the second `surrogate_based_global` use case, we want to update the surrogate, but globally. That is, we add points to the sample set used to create the surrogate, rebuild the surrogate, and then perform another global optimization on the new surrogate. Thus, surrogate-based global optimization can be used in an iterative scheme. In one iteration, minimizers of the surrogate model are found, and a selected subset of these are passed to the next iteration. In the next iteration, these surrogate points are evaluated with the "truth" model, and then added to the set of points upon which the next surrogate is constructed. This presents a more accurate surrogate to the minimizer at each subsequent iteration, presumably driving to optimality quickly. Note that a global surrogate is constructed using the same bounds in each iteration. This approach has no guarantee of convergence.

The surrogate-based global method was originally designed for MOGA (a multi-objective genetic algorithm). Since genetic algorithms often need thousands or tens of thousands of points to produce optimal or near-optimal solutions, surrogates can help by reducing the necessary truth model evaluations. Instead of creating one set of surrogates for the individual objectives and running the optimization algorithm on the surrogate once, the idea is to select points along the (surrogate) Pareto frontier, which can be used to supplement the existing points. In this way, one does not need to use many points initially to get a very accurate surrogate. The surrogate becomes more accurate as the iterations progress.

Most single objective optimization methods will return only a single optimal point. In that case, only one point from the surrogate model will be evaluated with the "true" function and added to the pointset upon which the surrogate is based. In this case, it will take many iterations of the surrogate-based global optimization for the approach to converge, and its utility may not be as great as for the multi-objective case when multiple optimal solutions are passed from one iteration to the next to supplement the surrogate. Note that the user has the option of appending the optimal points from the surrogate model to the current set of truth points or using the optimal points from the surrogate model to replace the optimal set of points from the previous iteration. Although appending to the set is the default behavior, at this time we strongly recommend using the option `replace_points` because

it appears to be more accurate and robust.

When using the surrogate-based global method, we first recommend running one optimization on a single surrogate model. That is, set `max_iterations` to 1. This will allow one to get a sense of where the optima are located and also what surrogate types are the most accurate to use for the problem. Note that by fixing the seed of the sample on which the surrogate is built, one can take a DAKOTA input file, change the surrogate type, and re-run the problem without any additional function evaluations by specifying the use of the dakota restart file which will pick up the existing function evaluations, create the new surrogate type, and run the optimization on that new surrogate. Also note that one can specify that surrogates be built for all primary functions and constraints or for only a subset of these functions and constraints. This allows one to use a "truth" model directly for some of the response functions, perhaps due to them being much less expensive than other functions. Finally, a diagnostic threshold can be used to stop the method if the surrogate is so poor that it is unlikely to provide useful points. If the goodness-of-fit has an R-squared value less than 0.5, meaning that less than half the variance of the output can be explained or accounted for by the surrogate model, the surrogate-based global optimization stops and outputs an error message. This is an arbitrary threshold, but generally one would want to have an R-squared value as close to 1.0 as possible, and an R-squared value below 0.5 indicates a very poor fit.

For the surrogate-based global method, we initially recommend a small number of maximum iterations, such as 3–5, to get a sense of how the optimization is evolving as the surrogate gets updated globally. If it appears to be changing significantly, then a larger number (used in combination with restart) may be needed.

Figure 9.4 shows a DAKOTA input file that implements surrogate-based global optimization on a multi-objective test function. This input file is named `dakota_su_mogatest1.in` in the `Dakota/test` directory. The first method keyword block contains the keyword `surrogate_based_global`, plus the commands for specifying five as the maximum iterations and the option to replace points in the global surrogate construction. The method block identified as MOGA specifies a multi-objective genetic algorithm optimizer and its controls. The model keyword block specifies a surrogate model. In this case, a `gaussian_process` model is used as a surrogate. The `dace_method_pointer` specifies that the surrogate will be build on 100 Latin Hypercube samples with a seed = 531. The remainder of the input specification deals with the interface to the actual analysis driver and the 2 responses being returned as objective functions from that driver.

9.4 Efficient Global Minimization

Efficient Global Optimization (EGO) is a global optimization technique that employs response surface surrogates [70, 68]. In each EGO iteration, a Gaussian process (GP) approximation for the objective function is constructed based on sample points of the true simulation. The GP allows one to specify the prediction at a new input location as well as the uncertainty associated with that prediction. The key idea in EGO is to maximize an Expected Improvement Function (EIF), defined as the expectation that any point in the search space will provide a better solution than the current best solution, based on the expected values and variances predicted by the GP model. It is important to understand how the use of this EIF leads to optimal solutions. The EIF indicates how much the objective function value at a new potential location is expected to be less than the predicted value at the current best solution. Because the GP model provides a Gaussian distribution at each predicted point, expectations can be calculated. Points with good expected values and even a small variance will have a significant expectation of producing a better solution (exploitation), but so will points that have relatively poor expected values and greater variance (exploration). The EIF incorporates both the idea of choosing points which minimize the objective and choosing points about which there is large prediction uncertainty (e.g., there are few or no samples in that area of the space, and thus the probability may be high that a sample value is potentially lower than other values). Because the uncertainty is higher in regions of the design space with few observations, this provides a balance between exploiting areas of the design space that predict good solutions, and exploring areas where more

```
strategy,
    single_method
    tabular_graphics_data
    method_pointer = 'SBGO'

method,
    id_method = 'SBGO'
    surrogate_based_global
    model_pointer = 'SURROGATE'
    approx_method_pointer = 'MOGA'
    max_iterations = 5
    replace_points
    output verbose

method,
    id_method = 'MOGA'
    moga
    output silent
        seed = 10983
    population_size = 300
    max_function_evaluations = 5000
    initialization_type unique_random
    crossover_type shuffle_random
        num_offspring = 2 num_parents = 2
        crossover_rate = 0.8
    mutation_type replace_uniform
        mutation_rate = 0.1
    fitness_type domination_count
    replacement_type below_limit = 6
        shrinkage_percentage = 0.9
    niching_type distance 0.05 0.05
    postprocessor_type
        orthogonal_distance 0.05 0.05
    convergence_type metric_tracker
        percent_change = 0.05 num_generations = 10

model,
    id_model = 'SURROGATE'
    surrogate global
        responses_pointer = 'SURROGATE_RESP'
        dace_method_pointer = 'SAMPLING'
        correction additive zeroth_order
      #  polynomial quadratic
        gaussian_process dakota

variables,
    continuous_design = 3
        initial_point      0     0     0
        upper_bounds       4     4     4
        lower_bounds      -4    -4    -4
        descriptors      'x1'  'x2'  'x3'

responses,
    id_responses = 'SURROGATE_RESP'
    objective_functions = 2
    no_gradients
    no_hessians

method,
    id_method = 'SAMPLING'
    model_pointer = 'TRUTH'
    sampling
      samples = 100
      seed = 531
      sample_type lhs
      all_variables

model,
    id_model = 'TRUTH'
    single
      interface_pointer = 'TRUE_FN'
      responses_pointer = 'TRUE_RESP'

interface,
    id_interface = 'TRUE_FN'
    system
      analysis_driver = 'mogatest1'

responses,
    id_responses = 'TRUE_RESP'
    objective_functions = 2
    no_gradients
    no_hessians
```

Figure 9.4: DAKOTA input file for the surrogate-based global optimization example.

```
strategy,
        single_method
          tabular_graphics_data

method,
        efficient_global
          seed = 123456

variables,
        continuous_design = 2
          lower_bounds   -2.0 -2.0
          upper_bounds    2.0  2.0
          descriptors     'x1' 'x2'

interface,
        system
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

Figure 9.5: DAKOTA input file for the efficient global optimization example.

information is needed.

There are two major differences between our implementation and that of [70]: we do not use a branch and bound method to find points which maximize the EIF. Rather, we use the DIRECT algorithm. Second, we allow for multiobjective optimization and nonlinear least squares including general nonlinear constraints. Constraints are handled through an augmented Lagrangian merit function approach (see Surrogate-Based Minimization chapter in DAKOTA Theory Manual [2]).

The method is specified as `efficient_global`. Currently we do not expose any specification controls for the underlying Gaussian process model used or for the optimization of the expected improvement function, which is currently performed by the NCSU DIRECT algorithm. The only item the user can specify is a seed which is used in the Latin Hypercube Sampling to generate the initial set of points which is used to construct the initial Gaussian process. An example specification for the EGO algorithm is shown in Figure 9.5. This can be found in the `dakota_rosenbrock_ego.in` file in the `Dakota/test` directory.

Chapter 10

Models

10.1 Overview

Chapters 4 through 8 have presented the different "iterators" available in DAKOTA. An iterator iterates on a model in order to map a set of variables into a set of responses. This model may involve a simple mapping involving a single interface, or it may involve recursions using sub-iterator and sub-models. These recursion capabilities were developed in order to provide mechanisms for "nesting," "layering," and "recasting" of software components, which allows the use of these components as building blocks to accomplish more sophisticated studies, such as surrogate-based optimization or optimization under uncertainty. In a nested relationship, a sub-iterator is executed using its sub-model for every evaluation of the nested model. In a layered relationship, on the other hand, sub-iterators and sub-models are used only for periodic updates and verifications. And in a recast relationship, the input variable and output response definitions in a sub-model are reformulated in order to support new problem definitions. In each of these cases, the sub-model is of arbitrary type, such that model recursions can be chained together in as long of a sequence as needed (e.g., layered containing nested contained layered containing single in Section 17.2.2). Figure 10.1 displays the model class hierarchy from the DAKOTA Developers Manual [4], with derived classes for single models, nested models, recast models, and two types of surrogate models: data fit and hierarchical/multifidelity. A third type of derived surrogate model supporting reduced-order models (ROM) is planned for future releases.

Section 10.2 describes single models; Section 10.3 describes recast models; Section 10.4 describes surrogate models of the data fit, multifidelity, and ROM type; and Section 10.5 describes nested models. Finally, Chapter 17 presents a number of advanced examples demonstrating these model recursions.

Figure 10.1: The DAKOTA model class hierarchy.

10.2 Single Models

The single model is the simplest model type. It uses a single interface instance (see Chapter 12) to map variables (see Chapter 11) into responses (see Chapter 13). There is no recursion in this case. Refer to the Models chapter in the DAKOTA Reference Manual [3] for additional information on the single model specification.

10.3 Recast Models

The recast model is not directly visible to the user within the input specification. Rather, it is used "behind the scenes" to recast the inputs and outputs of a sub-model for the purposes of reformulating the problem posed to an iterator. Examples include variable and response scaling (see Section 7.3.2), transformations of uncertain variables and associated response derivatives to employ standardized random variables (see Sections 6.3 and 6.4), multiobjective optimization (see Section 7.3.1), merit functions (see Section 9.2), and expected improvement/feasibility (see Sections 9.4 and 6.3.2). Refer to the DAKOTA Developers Manual [4] for additional details on the mechanics of recasting problem formulations.

10.4 Surrogate Models

Surrogate models provide an approximation to an original, high fidelity "truth" model. A number of surrogate model selections are possible, which are categorized as data fits, multifidelity models, and reduced-order models.

Each of the surrogate model types supports the use of correction factors that improve the local accuracy of the surrogate models. The correction factors force the surrogate models to match the true function values and possibly true function derivatives at the center point of each trust region. Currently, DAKOTA supports either zeroth-, first-, or second-order accurate correction methods, each of which can be applied using either an additive, multiplicative, or combined correction function. For each of these correction approaches, the correction is applied to the surrogate model and the corrected model is then interfaced with whatever algorithm is being employed. The default behavior is that no correction factor is applied.

The simplest correction approaches are those that enforce consistency in function values between the surrogate and original models at a single point in parameter space through use of a simple scalar offset or scaling applied to the surrogate model. First-order corrections such as the first-order multiplicative correction (also known as beta correction [12]) and the first-order additive correction [77] also enforce consistency in the gradients and provide a much more substantial correction capability that is sufficient for ensuring provable convergence in SBO algorithms (see Section 9.2). SBO convergence rates can be further accelerated through the use of second-order corrections which also enforce consistency in the Hessians [26], where the second-order information may involve analytic, finite-difference, or quasi-Newton Hessians.

Correcting surrogate models with additive corrections involves

$$\hat{f_{hi_\alpha}}(\mathbf{x}) = f_{lo}(\mathbf{x}) + \alpha(\mathbf{x}) \tag{10.1}$$

where multifidelity notation has been adopted for clarity. For multiplicative approaches, corrections take the form

$$\hat{f_{hi_\beta}}(\mathbf{x}) = f_{lo}(\mathbf{x})\beta(\mathbf{x}) \tag{10.2}$$

where, for local corrections, $\alpha(\mathbf{x})$ and $\beta(\mathbf{x})$ are first or second-order Taylor series approximations to the exact

correction functions:

$$\alpha(\mathbf{x}) = A(\mathbf{x_c}) + \nabla A(\mathbf{x_c})^T(\mathbf{x}-\mathbf{x_c}) + \frac{1}{2}(\mathbf{x}-\mathbf{x_c})^T\nabla^2 A(\mathbf{x_c})(\mathbf{x}-\mathbf{x_c}) \tag{10.3}$$

$$\beta(\mathbf{x}) = B(\mathbf{x_c}) + \nabla B(\mathbf{x_c})^T(\mathbf{x}-\mathbf{x_c}) + \frac{1}{2}(\mathbf{x}-\mathbf{x_c})^T\nabla^2 B(\mathbf{x_c})(\mathbf{x}-\mathbf{x_c}) \tag{10.4}$$

where the exact correction functions are

$$A(\mathbf{x}) = f_{hi}(\mathbf{x}) - f_{lo}(\mathbf{x}) \tag{10.5}$$

$$B(\mathbf{x}) = \frac{f_{hi}(\mathbf{x})}{f_{lo}(\mathbf{x})} \tag{10.6}$$

Refer to [26] for additional details on the derivations.

A combination of additive and multiplicative corrections can provide for additional flexibility in minimizing the impact of the correction away from the trust region center. In other words, both additive and multiplicative corrections can satisfy local consistency, but through the combination, global accuracy can be addressed as well. This involves a convex combination of the additive and multiplicative corrections:

$$\hat{f_{hi_\gamma}}(\mathbf{x}) = \gamma \hat{f_{hi_\alpha}}(\mathbf{x}) + (1-\gamma)\hat{f_{hi_\beta}}(\mathbf{x}) \tag{10.7}$$

where γ is calculated to satisfy an additional matching condition, such as matching values at the previous design iterate.

10.4.1 Data Fit Surrogate Models

A surrogate of the *data fit* type is a non-physics-based approximation typically involving interpolation or regression of a set of data generated from the original model. Data fit surrogates can be further characterized by the number of data points used in the fit, where a local approximation (e.g., first or second-order Taylor series) uses data from a single point, a multipoint approximation (e.g., two-point exponential approximations (TPEA) or two-point adaptive nonlinearity approximations (TANA)) uses a small number of data points often drawn from the previous iterates of a particular algorithm, and a global approximation (e.g., polynomial response surfaces, kriging/gaussian_process, neural networks, radial basis functions, splines) uses a set of data points distributed over the domain of interest, often generated using a design of computer experiments.

DAKOTA contains several types of surface fitting methods that can be used with optimization and uncertainty quantification methods and strategies such as surrogate-based optimization and optimization under uncertainty. These are: polynomial models (linear, quadratic, and cubic), first-order Taylor series expansion, kriging spatial interpolation, artificial neural networks, multivariate adaptive regression splines, radial basis functions, and moving least squares. With the exception of Taylor series methods, all of the above methods listed in the previous sentence are accessed in DAKOTA through the Surfpack library. All of these surface fitting methods can be applied to problems having an arbitrary number of design parameters. However, surface fitting methods usually are practical only for problems where there are a small number of parameters (e.g., a maximum of somewhere in the range of 30-50 design parameters). The mathematical models created by surface fitting methods have a variety of names in the engineering community. These include surrogate models, meta-models, approximation models, and response surfaces. For this manual, the terms surface fit model and surrogate model are used.

The data fitting methods in DAKOTA include software developed by Sandia researchers and by various researchers in the academic community.

10.4.1.1 Procedures for Surface Fitting

The surface fitting process consists of three steps: (1) selection of a set of design points, (2) evaluation of the true response quantities (e.g., from a user-supplied simulation code) at these design points, and (3) using the response data to solve for the unknown coefficients (e.g., polynomial coefficients, neural network weights, kriging correlation factors) in the surface fit model. In cases where there is more than one response quantity (e.g., an objective function plus one or more constraints), then a separate surface is built for each response quantity. Currently, the surface fit models are built using only 0th-order information (function values only), although extensions to using higher-order information (gradients and Hessians) are possible. Each surface fitting method employs a different numerical method for computing its internal coefficients. For example, the polynomial surface uses a least-squares approach that employs a singular value decomposition to compute the polynomial coefficients, whereas the kriging surface uses Maximum Likelihood Estimation to compute its correlation coefficients. More information on the numerical methods used in the surface fitting codes is provided in the DAKOTA Developers Manual [4].

The set of design points that is used to construct a surface fit model is generated using either the DDACE software package [114] or the LHS software package [69]. These packages provide a variety of sampling methods including Monte Carlo (random) sampling, Latin hypercube sampling, orthogonal array sampling, central composite design sampling, and Box-Behnken sampling. More information on these software packages is provided in Chapter 5.

10.4.1.2 Taylor Series

The Taylor series model is purely a local approximation method. That is, it provides local trends in the vicinity of a single point in parameter space. The first-order Taylor series expansion is:

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla_{\mathbf{x}} f(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) \tag{10.8}$$

and the second-order expansion is:

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla_{\mathbf{x}} f(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \nabla^2_{\mathbf{x}} f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) \tag{10.9}$$

where $\mathbf{x}_0$ is the expansion point in n-dimensional parameter space and $f(\mathbf{x}_0)$, $\nabla_{\mathbf{x}} f(\mathbf{x}_0)$, and $\nabla^2_{\mathbf{x}} f(\mathbf{x}_0)$ are the computed response value, gradient, and Hessian at the expansion point, respectively. As dictated by the responses specification used in building the local surrogate, the gradient may be analytic or numerical and the Hessian may be analytic, numerical, or based on quasi-Newton secant updates.

In general, the Taylor series model is accurate only in the region of parameter space that is close to $\mathbf{x}_0$. While the accuracy is limited, the first-order Taylor series model reproduces the correct value and gradient at the point $\mathbf{x}_0$, and the second-order Taylor series model reproduces the correct value, gradient, and Hessian. This consistency is useful in provably-convergent surrogate-based optimization. The other surface fitting methods do not use gradient information directly in their models, and these methods rely on an external correction procedure in order to satisfy the consistency requirements of provably-convergent SBO.

10.4.1.3 Two Point Adaptive Nonlinearity Approximation

The TANA-3 method [127] is a multipoint approximation method based on the two point exponential approximation [35]. This approach involves a Taylor series approximation in intermediate variables where the powers used for the intermediate variables are selected to match information at the current and previous expansion points. The

form of the TANA model is:

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x}_2) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\mathbf{x}_2) \frac{x_{i,2}^{1-p_i}}{p_i}(x_i^{p_i} - x_{i,2}^{p_i}) + \frac{1}{2}\epsilon(\mathbf{x}) \sum_{i=1}^{n}(x_i^{p_i} - x_{i,2}^{p_i})^2 \tag{10.10}$$

where n is the number of variables and:

$$p_i = 1 + \ln\left[\frac{\frac{\partial f}{\partial x_i}(\mathbf{x}_1)}{\frac{\partial f}{\partial x_i}(\mathbf{x}_2)}\right] \Big/ \ln\left[\frac{x_{i,1}}{x_{i,2}}\right] \tag{10.11}$$

$$\epsilon(\mathbf{x}) = \frac{H}{\sum_{i=1}^{n}(x_i^{p_i} - x_{i,1}^{p_i})^2 + \sum_{i=1}^{n}(x_i^{p_i} - x_{i,2}^{p_i})^2} \tag{10.12}$$

$$H = 2\left[f(\mathbf{x}_1) - f(\mathbf{x}_2) - \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\mathbf{x}_2) \frac{x_{i,2}^{1-p_i}}{p_i}(x_{i,1}^{p_i} - x_{i,2}^{p_i})\right] \tag{10.13}$$

and $\mathbf{x}_2$ and $\mathbf{x}_1$ are the current and previous expansion points. Prior to the availability of two expansion points, a first-order Taylor series is used.

10.4.1.4 Linear, Quadratic, and Cubic Polynomial Models

Linear, quadratic, and cubic polynomial models are available in DAKOTA. The form of the linear polynomial model is

$$\hat{f}(\mathbf{x}) \approx c_0 + \sum_{i=1}^{n} c_i x_i \tag{10.14}$$

the form of the quadratic polynomial model is:

$$\hat{f}(\mathbf{x}) \approx c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n}\sum_{j \geq i}^{n} c_{ij} x_i x_j \tag{10.15}$$

and the form of the cubic polynomial model is:

$$\hat{f}(\mathbf{x}) \approx c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n}\sum_{j \geq i}^{n} c_{ij} x_i x_j + \sum_{i=1}^{n}\sum_{j \geq i}^{n}\sum_{k \geq j}^{n} c_{ijk} x_i x_j x_k \tag{10.16}$$

In all of the polynomial models, $\hat{f}(\mathbf{x})$ is the response of the polynomial model; the x_i, x_j, x_k terms are the components of the n-dimensional design parameter values; the c_0 , c_i , c_{ij} , c_{ijk} terms are the polynomial coefficients, and n is the number of design parameters. The number of coefficients, n_c, depends on the order of polynomial model and the number of design parameters. For the linear polynomial:

$$n_{c_{linear}} = n + 1 \tag{10.17}$$

for the quadratic polynomial:

$$n_{c_{quad}} = \frac{(n+1)(n+2)}{2} \tag{10.18}$$

and for the cubic polynomial:

$$n_{c_{cubic}} = \frac{(n^3 + 6n^2 + 11n + 6)}{6} \tag{10.19}$$

There must be at least n_c data samples in order to form a fully determined linear system and solve for the polynomial coefficients. In DAKOTA, a least-squares approach involving a singular value decomposition numerical method is applied to solve the linear system.

The utility of the polynomial models stems from two sources: (1) over a small portion of the parameter space, a low-order polynomial model is often an accurate approximation to the true data trends, and (2) the least-squares procedure provides a surface fit that smooths out noise in the data. For this reason, the surrogate-based optimization strategy often is successful when using polynomial models, particularly quadratic models. However, a polynomial surface fit may not be the best choice for modeling data trends over the entire parameter space, unless it is known a priori that the true data trends are close to linear, quadratic, or cubic. See [83] for more information on polynomial models.

10.4.1.5 Kriging/Gaussian-Process Spatial Interpolation Models

In DAKOTA 5.2, we have 2 versions of spatial interpolation models. One is located in DAKOTA itself and the other in the Surfpack subpackage of DAKOTA which can be compiled in a stand alone mode. These models are denoted as `kriging dakota` and `kriging surfpack` or as `gaussian_process dakota` and `gaussian_process surfpack`. In prior DAKOTA releases, the `dakota` version was referred to as the `gaussian_process` model while the `surfpack` version was referred to as the `kriging` model. As of DAKTOA 5.2, specifying only `gaussian_process` or `kriging` will default to the `surfpack` version in all contexts except Bayesian calibration. For now, both versions are supported but the `dakota` version is deprecated and intended to be removed in a future release. The two `kriging` or `gaussian_process` models are very similar: the differences between them are explained in more detail below.

The Kriging, also known as Gaussian process (GP), method uses techniques developed in the geostatistics and spatial statistics communities ([16], [73]) to produce smooth surface fit models of the response values from a set of data points. The number of times the fitted surface is differentiable will depend on the correlation function that is used. Currently, the Gaussian correlation function is the only option for either version included in DAKOTA; this makes the GP model C^∞-continuous. The form of the GP model is

$$\hat{f}(\underline{x}) \approx \underline{g}(\underline{x})^T \underline{\beta} + \underline{r}(\underline{x})^T \underline{\underline{R}}^{-1} (\underline{f} - \underline{\underline{G}}\,\underline{\beta}) \tag{10.20}$$

where $\underline{x}$ is the current point in n-dimensional parameter space; $\underline{g}(\underline{x})$ is the vector of trend basis functions evaluated at $\underline{x}$; $\underline{\beta}$ is a vector containing the generalized least squares estimates of the trend basis function coefficients; $\underline{r}(\underline{x})$ is the correlation vector of terms between $\underline{x}$ and the data points; $\underline{\underline{R}}$ is the correlation matrix for all of the data points; $\underline{f}$ is the vector of response values; and $\underline{\underline{G}}$ is the matrix containing the trend basis functions evaluated at all data points. The terms in the correlation vector and matrix are computed using a Gaussian correlation function and are dependent on an n-dimensional vector of correlation parameters, $\underline{\theta} = \{\theta_1, \ldots, \theta_n\}^T$. By default, DAKOTA determines the value of $\underline{\theta}$ using a Maximum Likelihood Estimation (MLE) procedure. However, the user can also opt to manually set them in the `gaussian_process surfpack` model by specifying a vector of correlation

lengths, $\underline{l} = \{l_1, \ldots, l_n\}^T$ where $\theta_i = 1/(2l_i^2)$. This definition of correlation lengths makes their effect on the GP model's behavior directly analogous to the role played by the standard deviation in a normal (a.k.a. Gaussian) distribution. In the `gaussian_process surpack` model, we used this analogy to define a small feasible region in which to search for correlation lengths. This region should (almost) always contain some correlation matrices that are well conditioned and some that are optimal, or at least near optimal. More details on Kriging/GP models may be found in [52].

Since a GP has a hyper-parametric error model, it can be used to model surfaces with slope discontinuities along with multiple local minima and maxima. GP interpolation is useful for both SBO and OUU, as well as for studying the global response value trends in the parameter space. This surface fitting method needs a minimum number of design points equal to the sum of the number of basis functions and the number of dimensions, n, but it is recommended to use at least double this amount.

The GP model is guaranteed to pass through all of the response data values that are used to construct the model. Generally, this is a desirable feature. However, if there is considerable numerical noise in the response data, then a surface fitting method that provides some data smoothing (e.g., quadratic polynomial, MARS) may be a better choice for SBO and OUU applications. Another feature of the GP model is that the predicted response values, $\hat{f}(\underline{x})$, decay to the trend function, $\underline{g}(\underline{x})^T\underline{\beta}$, when $\underline{x}$ is far from any of the data points from which the GP model was constructed (i.e., when the model is used for extrapolation).

As mentioned above, there are two `gaussian_process` models in DAKOTA 5.2, the `surfpack` version and the `dakota` version. More details on the `gaussian_process dakota` model can be found in [79]. The differences between these models are as follows:

- Trend Function: The GP models incorporate a parametric trend function whose purpose is to capture large-scale variations. In both models, the trend function can be a constant, linear,or reduced quadratic (main effects only, no interaction terms) polynomial. This is specified by the keyword `trend` followed by one of `constant`, `linear`, or `reduced_quadratic` (in DAKOTA 5.0 and earlier, the reduced quadratic option for the `dakota` version was selected using the keyword, `quadratic`). The `gaussian_process surfpack` model has the additional option of a full (i.e. it includes interaction terms) quadratic polynomial; this is accessed by following the `trend` keyword with `quadratic`.
- Correlation Parameter Determination: Both of the `gaussian_process` models use a Maximum Likelihood Estimation (MLE) approach to find the optimal values of the hyper-parameters governing the mean and correlation functions. By default both models use the global optimization method called DIRECT, although they search regions with different extents. For the `gaussian_process dakota` model, DIRECT is the only option. The `gaussian_process surfpack` model has several options for the optimization method used. These are specified by the `optimization_method` keyword followed by one of these strings:
  - `'global'` which uses the default DIRECT optimizer,
  - `'local'` which uses the CONMIN optimizer,
  - `'sampling'` which generates several random guesses and picks the candidate with greatest likelihood, and
  - `'none'`

  The `'none'` option, and the starting location of the `'local'` optimization, default to the center, in log(correlation length) scale, of the of small feasible region. However, these can also be user specified with the `correlation_lengths` keyword followed by a list of n real numbers. The total number of evaluations of the `gaussian_process surfpack` model's likelihood function can be controlled using the `max_trials` keyword followed by a positive integer. Note that we have found the `'global'` optimization method to be the most robust.

- Ill-conditioning. One of the major problems in determining the governing values for a Gaussian process or Kriging model is the fact that the correlation matrix can easily become ill-conditioned when there are too many input points close together. Since the predictions from the Gaussian process model involve inverting the correlation matrix, ill-conditioning can lead to poor predictive capability and should be avoided. The `gaussian_process surfpack` model defines a small feasible search region for correlation lengths, which should (almost) always contain some well conditioned correlation matrices. In DAKOTA 5.1, the `kriging` (now `gaussian_process surfpack` or `kriging surfpack`) model avoided ill-conditioning by explicitly excluding poorly conditioned $\underline{\underline{R}}$ from consideration on the basis of their having a large (estimate of) condition number; this constraint acted to decrease the size of admissible correlation lengths. Note that a sufficiently bad sample design could require correlation lengths to be so short that any interpolatory Kriging/GP model would become inept at extrapolation and interpolation.

  The `gaussian_process dakota` model has two features to overcome ill-conditioning. The first is that the algorithm will add a small amount of noise to the diagonal elements of the matrix (this is often referred to as a "nugget") and sometimes this is enough to improve the conditioning. The second is that the user can specify to build the GP based only on a subset of points. The algorithm chooses an "optimal" subset of points (with respect to predictive capability on the remaining unchosen points) using a greedy heuristic. This option is specified with the keyword `point_selection` in the input file.

  As of DAKOTA 5.2, the `gaussian_process surfpack` model has a similar capability. Points are **not** discarded prior to the construction of the model. Instead, within the maximum likelihood optimization loop, when the correlation matrix violates the explicit (estimate of) condition number constraint, the `gaussian_process surfpack` model will perform a pivoted Cholesky factorization of the correlation matrix. A bisection search is then used to efficiently find the last point for which the reordered correlation matrix is not too ill-conditioned. Subsequent reordered points are excluded from the GP/Kriging model for the current set of correlation lengths, i.e. they are not used to construct this GP model or compute its likelihood. When necessary, the `gaussian_process surfpack` model will automatically decrease the order of the polynomial trend function. Once the maximum likelihood optimization has been completed, the subset of points that is retained will be the one associated with the most likely set of correlation lengths. Note that a matrix being ill-conditioned means that its rows or columns contain a significant amount of duplicate information. Since the points that were discarded were the ones that contained the least unique information, they should be the ones that are the easiest to predict and provide maximum improvement of the condition number. However, the `gaussian_process surfpack` model is not guaranteed to exactly interpolate the discarded points. Warning: when two very nearby points are on opposite sides of a discontinuity, it is possible for one of them to be discarded by this approach.

  Note that a pivoted Cholesky factorization can be significantly slower than the highly optimized implementation of non-pivoted Cholesky factorization in typical LAPACK distributions. A consequence of this is that the `gaussian_process surfpack` model can take significantly more time to build than the `gaussian_process dakota` version. However, tests indicate that the `gaussian_process surfpack` version will often be more accurate and/or require fewer evaluations of the true function than the `gaussian_process dakota`. For this reason, the `gaussian_process surfpack` version is the default option as of DAKOTA 5.2.

- Gradient Enhanced Kriging (GEK). As of Dakota 5.2, the `use_derivatives` keyword will cause the `gaussian_process surfpack` model to be built from a combination of function value and gradient information. The `gaussian_process dakota` model does not have this capability. Incorporating gradient information will only be beneficial if accurate and inexpensive derivative information is available, and the derivatives are not infinite or nearly so. Here "inexpensive" means that the cost of evaluating a function value plus gradient is comparable to the cost of evaluating only the function value, for example gradients

computed by analytical, automatic differentiation, or continuous adjoint techniques. It is not cost effective to use derivatives computed by finite differences. In tests, GEK models built from finite difference derivatives were also significantly less accurate than those built from analytical derivatives. Note that GEK's correlation matrix tends to have a significantly worse condition number than Kriging for the same sample design.

This issue was addressed by using a pivoted Cholesky factorization of Kriging's correlation matrix (which is a small sub-matrix within GEK's correlation matrix) to rank points by how much unique information they contain. This reordering is then applied to whole points (the function value at a point immediately followed by gradient information at the same point) in GEK's correlation matrix. A standard non-pivoted Cholesky is then applied to the reordered GEK correlation matrix and a bisection search is used to find the last equation that meets the constraint on the (estimate of) condition number. The cost of performing pivoted Cholesky on Kriging's correlation matrix is usually negligible compared to the cost of the non-pivoted Cholesky factorization of GEK's correlation matrix. In tests, it also resulted in more accurate GEK models than when pivoted Cholesky or whole-point-block pivoted Cholesky was performed on GEK's correlation matrix.

10.4.1.6 Artificial Neural Network (ANN) Models

The ANN surface fitting method in DAKOTA employs a stochastic layered perceptron (SLP) artificial neural network based on the direct training approach of Zimmerman [128]. The SLP ANN method is designed to have a lower training cost than traditional ANNs. This is a useful feature for SBO and OUU where new ANNs are constructed many times during the optimization process (i.e., one ANN for each response function, and new ANNs for each optimization iteration). The form of the SLP ANN model is

$$\hat{f}(\mathbf{x}) \approx \tanh(\tanh((\mathbf{x}\mathbf{A}_0 + \theta_0)\mathbf{A}_1 + \theta_1)) \tag{10.21}$$

where $\mathbf{x}$ is the current point in n-dimensional parameter space, and the terms $\mathbf{A}_0, \theta_0, \mathbf{A}_1, \theta_1$ are the matrices and vectors that correspond to the neuron weights and offset values in the ANN model. These terms are computed during the ANN training process, and are analogous to the polynomial coefficients in a quadratic surface fit. A singular value decomposition method is used in the numerical methods that are employed to solve for the weights and offsets.

The SLP ANN is a non parametric surface fitting method. Thus, along with kriging and MARS, it can be used to model data trends that have slope discontinuities as well as multiple maxima and minima. However, unlike kriging, the ANN surface is not guaranteed to exactly match the response values of the data points from which it was constructed. This ANN can be used with SBO and OUU strategies. As with kriging, this ANN can be constructed from fewer than $n_{c_{quad}}$ data points, however, it is a good rule of thumb to use at least $n_{c_{quad}}$ data points when possible.

10.4.1.7 Multivariate Adaptive Regression Spline (MARS) Models

This surface fitting method uses multivariate adaptive regression splines from the MARS3.5 package [39] developed at Stanford University.

The form of the MARS model is based on the following expression:

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} a_m B_m(\mathbf{x}) \tag{10.22}$$

where the a_m are the coefficients of the truncated power basis functions B_m, and M is the number of basis functions. The MARS software partitions the parameter space into subregions, and then applies forward and backward regression methods to create a local surface model in each subregion. The result is that each subregion contains its own basis functions and coefficients, and the subregions are joined together to produce a smooth, C^2-continuous surface model.

MARS is a nonparametric surface fitting method and can represent complex multimodal data trends. The regression component of MARS generates a surface model that is not guaranteed to pass through all of the response data values. Thus, like the quadratic polynomial model, it provides some smoothing of the data. The MARS reference material does not indicate the minimum number of data points that are needed to create a MARS surface model. However, in practice it has been found that at least $n_{c_{quad}}$, and sometimes as many as 2 to 4 times $n_{c_{quad}}$, data points are needed to keep the MARS software from terminating. Provided that sufficient data samples can be obtained, MARS surface models can be useful in SBO and OUU applications, as well as in the prediction of global trends throughout the parameter space.

10.4.1.8 Radial Basis Functions

Radial basis functions are functions whose value typically depends on the distance from a center point, called the centroid, **c**. The surrogate model approximation is then built up as the sum of K weighted radial basis functions:

$$\hat{f}(\mathbf{x}) = \sum_{k=1}^{K} w_k \phi(\| \mathbf{x} - \mathbf{c_k} \|) \tag{10.23}$$

where the ϕ are the individual radial basis functions. These functions can be of any form, but often a Gaussian bell-shaped function or splines are used. Our implementation uses a Gaussian radial basis function. The weights are determined via a linear least squares solution approach. See [88] for more details.

10.4.1.9 Moving Least Squares

Moving Least Squares can be considered a more specialized version of linear regression models. In linear regression, one usually attempts to minimize the sum of the squared residuals, where the residual is defined as the difference between the surrogate model and the true model at a fixed number of points. In weighted least squares, the residual terms are weighted so the determination of the optimal coefficients governing the polynomial regression function, denoted by $\hat{f}(\mathbf{x})$, are obtained by minimizing the weighted sum of squares at N data points:

$$\sum_{n=1}^{N} w_n(\| \hat{f}(\mathbf{x_n}) - f(\mathbf{x_n}) \|) \tag{10.24}$$

Moving least squares is a further generalization of weighted least squares where the weighting is "moved" or recalculated for every new point where a prediction is desired. [85] The implementation of moving least squares is still under development. We have found that it works well in trust region methods where the surrogate model is constructed in a constrained region over a few points. It does not appear to be working as well globally, at least at this point in time.

10.4.2 Multifidelity Surrogate Models

A second type of surrogate is the *model hierarchy* type (also called multifidelity, variable fidelity, variable complexity, etc.). In this case, a model that is still physics-based but is of lower fidelity (e.g., coarser discretization, reduced element order, looser convergence tolerances, omitted physics) is used as the surrogate in place of the high-fidelity model. For example, an inviscid, incompressible Euler CFD model on a coarse discretization could be used as a low-fidelity surrogate for a high-fidelity Navier-Stokes model on a fine discretization.

10.4.3 Reduced Order Models

A third type of surrogate model involves *reduced-order modeling* techniques such as proper orthogonal decomposition (POD) in computational fluid dynamics (also known as principal components analysis or Karhunen-Loeve in other fields) or spectral decomposition (also known as modal analysis) in structural dynamics. These surrogate models are generated directly from a high-fidelity model through the use of a reduced basis (e.g., eigenmodes for modal analysis or left singular vectors for POD) and projection of the original high-dimensional system down to a small number of generalized coordinates. These surrogates are still physics-based (and may therefore have better predictive qualities than data fits), but do not require multiple system models of varying fidelity (as required for model hierarchy surrogates).

10.5 Nested Models

Nested models utilize a sub-iterator and a sub-model to perform a complete iterative study as part of every evaluation of the model. This sub-iteration accepts variables from the outer level, performs the sub-level analysis, and computes a set of sub-level responses which are passed back up to the outer level. As described in the Models chapter of the Reference Manual [3], mappings are employed for both the variable inputs to the sub-model and the response outputs from the sub-model.

In the variable mapping case, primary and secondary variable mapping specifications are used to map from the top-level variables into the sub-model variables. These mappings support three possibilities in any combination: (1) insertion of an active top-level variable value into an identified sub-model distribution parameter for an identified active sub-model variable, (2) insertion of an active top-level variable value into an identified active sub-model variable value, and (3) addition of an active top-level variable value as an inactive sub-model variable, augmenting the active sub-model variables.

In the response mapping case, primary and secondary response mapping specifications are used to map from the sub-model responses back to the top-level responses. These specifications provide real-valued multipliers that are applied to the sub-iterator response results to define the outer level response set. These nested data results may be combined with non-nested data through use of the "optional interface" component within nested models.

Several examples of nested model usage are provided in Chapter 17.

Chapter 11

Variables

11.1 Overview

The `variables` specification in a DAKOTA input file specifies the parameter set to be iterated by a particular method. In the case of an optimization study, these variables are adjusted in order to locate an optimal design; in the case of parameter studies/sensitivity analysis/design of experiments, these parameters are perturbed to explore the parameter space; and in the case of uncertainty analysis, the variables are associated with distribution/interval characterizations which are used to compute corresponding distribution/interval characterizations for response functions. To accommodate these and other types of studies, DAKOTA supports design, uncertain, and state variable types for continuous and discrete variable domains, where uncertain types can be further categorized as either aleatory or epistemic and discrete domains can be further categorized as discrete range, discrete integer set, or discrete real set.

This chapter will present a brief overview of the types of variables and their uses, as well as cover some user issues relating to file formats and the active set vector. For a detailed description of variables section syntax and example specifications, refer to the Variables Commands chapter in the DAKOTA Reference Manual [3].

11.2 Design Variables

Design variables are those variables which are modified for the purposes of computing an optimal design. These variables may be continuous (real-valued between bounds), discrete range (integer-valued between bounds), discrete set of integers (integer-valued from finite set), and discrete set of reals (real-valued from finite set).

11.2.1 Continuous Design Variables

The most common type of design variables encountered in engineering applications are of the continuous type. These variables may assume any real value (e.g., `12.34`, `-1.735e+07`) within their bounds. All but a handful of the optimization algorithms in DAKOTA support continuous design variables exclusively.

11.2.2 Discrete Design Variables

Engineering design problems may contain discrete variables such as material types, feature counts, stock gauge selections, etc. These variables may assume only a fixed number of values, as compared to a continuous variable which has an uncountable number of possible values within its range. Discrete variables may involve a range of consecutive integers (x can be any integer between `1` and `10`), a set of integer values (x can be `101`, `212`, or `355`), or a set of real values (e.g., x can be `4.2`, `6.4`, or `8.5`).

Discrete variables may be classified as either "categorical" or "noncategorical." In the latter noncategorical case, the discrete requirement can be relaxed during the solution process since the model can still compute meaningful response functions for values outside the allowed discrete range or set. For example, a discrete variable representing the thickness of a structure is generally a noncategorical variable since it can assume a continuous range of values during the algorithm iterations, even if it is desired to have a stock gauge thickness in the end. In the former categorical case, the discrete requirement cannot be relaxed since the model cannot obtain a solution for values outside the range or set. For example, feature counts are generally categorical discrete variables, since most computational models will not support a non-integer value for the number of instances of some feature (e.g., number of support brackets).

Gradient-based optimization methods cannot be directly applied to problems with discrete variables since derivatives only exist for a variable continuum. For problems with noncategorical variables, branch and bound techniques can be used to relax the discrete requirements and apply gradient-based methods to a series of generated subproblems. For problems with categorical variables, nongradient-based methods (e.g., `coliny_ea`) are commonly used.

In addition to engineering applications, many non-engineering applications in the fields of scheduling, logistics, and resource allocation contain discrete design parameters. Within the Department of Energy, solution techniques for these problems impact programs in stockpile evaluation and management, production planning, nonproliferation, transportation (routing, packing, logistics), infrastructure analysis and design, energy production, environmental remediation, and tools for massively parallel computing such as domain decomposition and meshing.

11.2.2.1 Discrete Design Integer Variables

There are two types of discrete design integer variables supported by DAKOTA.

- The `discrete_design_range` specification supports a range of consecutive integers between specified `lower_bounds` and `upper_bounds`.

- The `discrete_design_set_integer` specification supports a set of enumerated integer values through the `set_values` specification. The set of values specified is stored internally as an STL set container, which enforces an ordered, unique representation of the integer data. Underlying this set of ordered, unique integers is a set of indices that run from 0 to one less than the number of set values. These indices are used by some iterative algorithms (e.g., parameter studies, SCOLIB iterators) for simplicity in discrete value enumeration when the actual corresponding set values are immaterial. In the case of parameter studies, this index representation is exposed through certain step and partition control specifications (see Chapter 4).

11.2.2.2 Discrete Design Real Variables

There is one type of discrete design real variable supported by DAKOTA.

- The `discrete_design_set_real` specification specification supports a set of enumerated real values through the `set_values` specification. As for the discrete integer set variables described in Section 11.2.2.1, internal storage of the set values is ordered and unique and an underlying index representation is exposed for the specification of some iterative algorithms.

11.3 Uncertain Variables

Deterministic variables (i.e., those with a single known value) do not capture the behavior of the input variables in all situations. In many cases, the exact value of a model parameter is not precisely known. An example of such an input variable is the thickness of a heat treatment coating on a structural steel I-beam used in building construction. Due to variabilities and tolerances in the coating process, the thickness of the layer is known to follow a normal distribution with a certain mean and standard deviation as determined from experimental data. The inclusion of the uncertainty in the coating thickness is essential to accurately represent the resulting uncertainty in the response of the building.

11.3.1 Aleatory Uncertain Variables

Aleatory uncertainties are irreducible variabilities inherent in nature. They are characterized by having a sufficiently rich set of data as to allow modeling using probability distributions, and probabilistic methods are commonly used for propagating nput aleatory uncertainties described by probability distribution specifications. The two following sections describe the continuous and discrete aleatory uncertain variables supported by DAKOTA.

For aleatory random variables, DAKOTA supports a user-supplied correlation matrix to provide correlations among the input variables. By default, the correlation matrix is set to the identity matrix, i.e., no correlation among the uncertain variables.

For additional information on random variable probability distributions, refer to [62] and [109]. Refer to the DAKOTA Reference Manual [3] for more detail on the uncertain variable specifications and to Chapter 6 for a description of methods available to quantify the uncertainty in the response.

11.3.1.1 Continuous Aleatory Uncertain Variables

- Normal: a probability distribution characterized by a mean and standard deviation. Also referred to as Gaussian. Bounded normal is also supported by some methods with an additional specification of lower and upper bounds.
- Lognormal: a probability distribution characterized by a mean and either a standard deviation or an error factor. The natural logarithm of a lognormal variable has a normal distribution. Bounded lognormal is also supported by some methods with an additional specification of lower and upper bounds.
- Uniform: a probability distribution characterized by a lower bound and an upper bound. Probability is constant between the bounds.
- Loguniform: a probability distribution characterized by a lower bound and an upper bound. The natural logarithm of a loguniform variable has a uniform distribution.
- Triangular: a probability distribution characterized by a mode, a lower bound, and an upper bound.
- Exponential: a probability distribution characterized by a beta parameter.

- Beta: a flexible probability distribution characterized by a lower bound and an upper bound and alpha and beta parameters. The uniform distribution is a special case.

- Gamma: a flexible probability distribution characterized by alpha and beta parameters. The exponential distribution is a special case.

- Gumbel: the Type I Largest Extreme Value probability distribution. Characterized by alpha and beta parameters.

- Frechet: the Type II Largest Extreme Value probability distribution. Characterized by alpha and beta parameters.

- Weibull: the Type III Smallest Extreme Value probability distribution. Characterized by alpha and beta parameters.

- Histogram Bin: an empirically-based probability distribution characterized by a set of (x, y) pairs that map out histogram bins (a continuous interval with associated bin count).

11.3.1.2 Discrete Aleatory Uncertain Variables

The following types of discrete aleatory uncertain variables are available:

- Poisson: integer-valued distribution used to predict the number of discrete events that happen in a given time interval.

- Binomial: integer-valued distribution used to predict the number of failures in a number of independent tests or trials.

- Negative Binomial: integer-valued distribution used to predict the number of times to perform a test to have a target number of successes.

- Geometric: integer-valued distribution used to model the number of successful trials that might occur before a failure is observed.

- Hypergeometric: integer-valued distribution used to model the number of failures observed in a set of tests that has a known proportion of failures.

- Histogram Point: an empirically-based probability distribution characterized by a set of real-valued (x, y) pairs that map out histogram points (a discrete point value with associated count).

11.3.2 Epistemic Uncertain Variables

Epistemic uncertainties are reducible uncertainties resulting from a lack of knowledge. For epistemic uncertainties, data is generally sparse, making the use of probability theory questionable and leading to nonprobabilistic methods based on interval specifications DAKOTA currently supports one epistemic uncertain variable type.

11.3.2.1 Continuous Epistemic Uncertain Variables

- Interval: an interval-based specification characterized by sets of lower and upper bounds and Basic Probability Assignments (BPAs) associated with each interval. The intervals may be overlapping, contiguous, or disjoint, and a single interval (with probability = 1) per variable is an important special case. The interval distribution is not a probability distribution, as the exact structure of the probabilities within each interval is not known. It is commonly used with epistemic uncertainty methods.

11.4 State Variables

State variables consist of "other" variables which are to be mapped through the simulation interface, in that they are not to be used for design and they are not modeled as being uncertain. State variables provide a convenient mechanism for parameterizing additional model inputs which, in the case of a numerical simulator, might include solver convergence tolerances, time step controls, or mesh fidelity parameters. For additional model parameterizations involving strings (e.g., "mesh1.exo"), refer to the analysis components specification described in Section 11.6.1 and in the Interface Commands chapter of the DAKOTA Reference Manual [3]. Similar to the design variables discussed in Section 11.2, state variables can be a continuous range (real-valued between bounds), a discrete range (integer-valued between bounds), a discrete integer-valued set, or a discrete real-valued set.

State variables, as with other types of variables, are viewed differently depending on the method in use. Since these variables are neither design nor uncertain variables, algorithms for optimization, least squares, and uncertainty quantification do not iterate on these variables; i.e., they are not active and are hidden from the algorithm. However, DAKOTA still maps these variables through the user's interface where they affect the computational model in use. This allows optimization, least squares, and uncertainty quantification studies to be executed under different simulation conditions (which will result, in general, in different results). Parameter studies and design of experiments methods, on the other hand, are general-purpose iterative techniques which do not draw a distinction between variable types. They include state variables in the set of variables to be iterated, which allows these studies to explore the effect of state variable values on the response data of interest.

In the future, state variables might be used in direct coordination with an optimization, least squares, or uncertainty quantification algorithm. For example, state variables could be used to enact model adaptivity through the use of a coarse mesh or loose solver tolerances in the initial stages of an optimization with continuous model refinement as the algorithm nears the optimal solution.

11.5 Mixed Variables

As alluded to in the previous section, the iterative method selected for use in DAKOTA determines what subset, or view, of the variables data is active in the iteration. The general case of having a mixture of various different types of variables is supported within all of the DAKOTA methods even though certain methods will only modify certain types of variables (e.g., optimizers and least squares methods only modify design variables, and uncertainty quantification methods, with the exception of `all_variables` mode, only utilize uncertain variables). This implies that variables which are not under the direct control of a particular iterator will be mapped through the interface in an unmodified state. This allows for a variety of parameterizations within the model in addition to those which are being used by a particular iterator, which can provide the convenience of consolidating the control over various modeling parameters in a single file (the DAKOTA input file). An important related point is that the variable set that is active with a particular iterator is the same variable set for which derivatives are typically computed (see Section 13.3).

```
<int>    variables
<double> <label_cdv_i>          (i = 1 to n_cdv)
<int>    <label_ddiv_i>         (i = 1 to n_ddiv)
<double> <label_ddrv_i>         (i = 1 to n_ddrv)
<double> <label_cauv_i>         (i = 1 to n_cauv)
<int>    <label_dauiv_i>        (i = 1 to n_dauiv)
<double> <label_daurv_i>        (i = 1 to n_daurv)
<double> <label_ceuv_i>         (i = 1 to n_ceuv)
<int>    <label_deuiv_i>        (i = 1 to n_deuiv)
<double> <label_deurv_i>        (i = 1 to n_deurv)
<double> <label_csv_i>          (i = 1 to n_csv)
<int>    <label_dsiv_i>         (i = 1 to n_dsiv)
<double> <label_dsrv_i>         (i = 1 to n_dsrv)
<int>    functions
<int>    ASV_i:label_response_i       (i = 1 to m)
<int>    derivative_variables
<int>    DVV_i:label_cdv_i            (i = 1 to p)
<int>    analysis_components
<string> AC_i:analysis_driver_name_i  (i = 1 to q)
```

Figure 11.1: Parameters file data format - standard option.

11.6 DAKOTA Parameters File Data Format

Simulation interfaces which employ system calls and forks to create separate simulation processes must communicate with the simulation code through the file system. This is accomplished through the reading and writing of parameters and results files. DAKOTA uses a particular format for this data input/output. Depending on the user's interface specification, DAKOTA will write the parameters file in either standard or APREPRO format (future XML formats are planned). The former option uses a simple "`value tag`" format, whereas the latter option uses a "`{ tag = value }`" format for compatibility with the APREPRO utility [103] (as well as DPrePro, BPREPRO, and JPrePost variants).

11.6.1 Parameters file format (standard)

Prior to invoking a simulation, DAKOTA creates a parameters file which contains the current parameter values and a set of function requests. The standard format for this parameters file is shown in Figure 11.1.

where "`<int>`" denotes an integer value, "`<double>`" denotes a double precision value, and "`<string>`" denotes a string value. Each of the colored blocks (black for variables, blue for active set vector, red for derivative variables vector, and green for analysis components) denotes an array which begins with an array length and a descriptive tag. These array lengths are useful for dynamic memory allocation within a simulator or filter program.

The first array for variables begins with the total number of variables (`n`) with its identifier string "`variables`." The next `n` lines specify the current values and descriptors of all of the variables within the parameter set *in the following order*: continuous design, discrete integer design (integer range, integer set), discrete real design (real set), continuous aleatory uncertain (normal, lognormal, uniform, loguniform, triangular, exponential, beta,

gamma, gumbel, frechet, weibull, histogram bin), discrete integer aleatory uncertain (poisson, binomial, negative binomial, geometric, hypergeometric), discrete real aleatory uncertain (histogram point), continuous epistemic uncertain (interval), discrete integer epistemic uncertain (none at this time), discrete real epistemic uncertain (none at this time), continuous state, discrete integer state (integer range, integer set), and discrete real state (real set) variables. This ordering is consistent with the lists in Sections 11.2.2.1, 11.3.1.1 and 11.3.1.2 and the specification order in dakota.input.txt. The lengths of these vectors add to a total of n (that is, $n = n_{cdv} + n_{ddiv} + n_{ddrv} + n_{cauv} + n_{dauiv} + n_{daurv} + n_{ceuv} + n_{deuiv} + n_{deurv} + n_{csv} + n_{dsiv} + n_{dsrv}$). If any of the variable types are not present in the problem, then its block is omitted entirely from the parameters file. The tags are the variable descriptors specified in the user's DAKOTA input file, or if no descriptors have been specified, default descriptors are used.

The second array for the active set vector (ASV) begins with the total number of functions (`m`) and its identifier string "`functions`." The next `m` lines specify the request vector for each of the `m` functions in the response data set followed by the tags "`ASV_i:label_response`", where the label is either a user-provided response descriptor or a default-generated one. These integer codes indicate what data is required on the current function evaluation and are described further in Section 11.7.

The third array for the derivative variables vector (DVV) begins with the number of derivative variables (`p`) and its identifier string "`derivative_variables`." The next `p` lines specify integer variable identifiers followed by the tags "`DVV_i:label_cdv`". These integer identifiers are used to identify the subset of variables that are active for the calculation of derivatives (gradient vectors and Hessian matrices), and correspond to the list of variables in the first array (e.g., an identifier of 2 indicates that the second variable in the list is active for derivatives). The labels are again taken from user-provided or default variable descriptors.

The final array for the analysis components (AC) begins with the number of analysis components (`q`) and its identifier string "`analysis_components`." The next `q` lines provide additional strings for use in specializing a simulation interface followed by the tags "`AC_i:analysis_driver_name`", where `analysis_driver_name` indicates the driver associated with this component. These strings are specified in a user's input file for a set of `analysis_drivers` using the `analysis_components` specification. The subset of the analysis components used for a particular analysis driver is the set passed in a particular parameters file.

Several standard-format parameters file examples are shown in Section 12.6.

11.6.2 Parameters file format (APREPRO)

For the APREPRO format option, the same data is present and the same ordering is used as in the standard format. The only difference is that values are associated with their tags within "`{ tag = value }`" constructs as shown in Figure 11.2. An APREPRO-format parameters file example is shown in Section 12.6.

The use of the APREPRO format option allows direct usage of these parameters files by the APREPRO utility, which is a file pre-processor that can significantly simplify model parameterization. Similar pre-processors include DPrePro, BPREPRO, and JPrePost. *[Note: APREPRO is a Sandia-developed pre-processor that is not currently distributed with DAKOTA. DPrePro is a Perl script distributed with DAKOTA that performs many of the same functions as APREPRO, and is optimized for use with DAKOTA parameters files in either format. BPREPRO and JPrePost are additional Perl and JAVA tools, respectively, in use at other sites.]* When a parameters file in APREPRO format is included within a template file (using an include directive), the APREPRO utility recognizes these constructs as variable definitions which can then be used to populate targets throughout the template file [103]. DPrePro, conversely, does not require the use of includes since it processes the DAKOTA parameters file and template simulation file separately to create a simulation input file populated with the variables data.

```
{ DAKOTA_VARS = <int> }
{ <label_cdv_i> = <double> }          (i = 1 to n_cdv)
{ <label_ddiv_i> = <int> }            (i = 1 to n_ddiv)
{ <label_ddrv_i> = <double> }         (i = 1 to n_ddrv)
{ <label_cauv_i> = <double> }         (i = 1 to n_cauv)
{ <label_dauiv_i> = <int> }           (i = 1 to n_dauiv)
{ <label_daurv_i> = <double> }        (i = 1 to n_daurv)
{ <label_ceuv_i> = <double> }         (i = 1 to n_ceuv)
{ <label_deuiv_i> = <int> }           (i = 1 to n_deuiv)
{ <label_deurv_i> = <double> }        (i = 1 to n_deurv)
{ <label_csv_i> = <double> }          (i = 1 to n_csv)
{ <label_dsiv_i> = <int> }            (i = 1 to n_dsiv)
{ <label_dsrv_i> = <double> }         (i = 1 to n_dsrv)
{ DAKOTA_FNS = <int> }
{ ASV_i:label_response_i = <int> }              (i = 1 to m)
{ DAKOTA_DER_VARS = <int> }
{ DVV_i:label_cdv_i = <int> }                   (i = 1 to p)
{ DAKOTA_AN_COMPS = <int> }
{ AC_i:analysis_driver_name_i = <string> }      (i = 1 to q)
```

Figure 11.2: Parameters file data format - APREPRO option.

11.7 The Active Set Vector

The active set vector contains a set of integer codes, one per response function, which describe the data needed on a particular execution of an interface. Integer values of 0 through 7 denote a 3-bit binary representation of all possible combinations of value, gradient, and Hessian requests for a particular function, with the most significant bit denoting the Hessian, the middle bit denoting the gradient, and the least significant bit denoting the value. The specific translations are shown in Table 11.1.

The active set vector in DAKOTA gets its name from managing the active set, i.e., the set of functions that are active on a particular function evaluation. However, it also manages the type of data that is needed for functions that are active, and in that sense, has an extended meaning beyond that typically used in the optimization literature.

Table 11.1: Active set vector integer codes.

Integer Code	Binary representation	Meaning
7	111	Get Hessian, gradient, and value
6	110	Get Hessian and gradient
5	101	Get Hessian and value
4	100	Get Hessian
3	011	Get gradient and value
2	010	Get gradient
1	001	Get value
0	000	No data required, function is inactive

11.7.1 Active set vector control

Active set vector control may be turned off to allow the user to simplify the supplied interface by removing the need to check the content of the active set vector on each evaluation. The Interface Commands chapter in the DAKOTA Reference Manual [3] provides additional information on this option (`deactivate active_set_vector`). Of course, this option trades some efficiency for simplicity and is most appropriate for those cases in which only a relatively small penalty occurs when computing and returning more data than may be needed on a particular function evaluation.

Chapter 12

Interfaces

12.1 Overview

The `interface` specification in a DAKOTA input file controls details of function evaluations. The mechanisms currently in place for function evaluations involve interfacing with one or more computational simulation codes, computing algebraic mappings, or a combination of the two.

This chapter will describe algebraic mappings in Section 12.2, followed by discussion of a variety of mechanisms for simulation code invocation in Section 12.3. This chapter also provides an overview of simulation interface components, covers issues relating to file management and presents a number of example data mappings.

For a detailed description of interface specification syntax, refer to the interface commands chapter in the DAKOTA Reference Manual [3].

12.2 Algebraic Mappings

If desired, one can define algebraic input-output mappings using the AMPL code [38] and save these mappings in 3 files: `stub.nl`, `stub.col`, and `stub.row`, where `stub` is a particular root name describing a particular problem. These files names can be communicated to DAKOTA using the `algebraic_mappings` input.

DAKOTA will use `stub.col` and `stub.row` to obtain input and output identifier strings, respectively, and will use the AMPL solver library [42] to evaluate expressions conveyed in `stub.nl`, and, if needed, their first and second derivatives.

As a simple example (from `Dakota/test/ampl/fma`), consider algebraic mappings based on Newton's law $F = ma$. The following is an AMPL input file of variable and expression declarations and output commands:

```
var mass;
var a;
var v;
minimize force: mass*a;
minimize energy: 0.5 * mass * v^2;
option auxfiles rc;     # request stub.row and stub.col
write gfma;             # write stub.nl, stub.row, stub.col
```

When processed by an AMPL processor, three files are created (as requested by the "option auxfiles" command). The first is the `fma.nl` file containing problem statistics, expression graphs, bounds, etc.:

```
g3 0 1 0        # problem fma
 3 0 2 0 0      # vars, constraints, objectives, ranges, eqns
 0 2    # nonlinear constraints, objectives
 0 0    # network constraints: nonlinear, linear
 0 3 0  # nonlinear vars in constraints, objectives, both
 0 0 0 1        # linear network variables; functions; arith, flags
 0 0 0 0 0      # discrete variables: binary, integer, nonlinear (b,c,o)
 0 4    # nonzeros in Jacobian, gradients
 6 4    # max name lengths: constraints, variables
 0 0 0 0 0      # common exprs: b,c,o,c1,o1
O0 0
o2
v0
v1
O1 0
o2
o2
n0.5
v0
o5
v2
n2
b
3
3
3
k2
0
0
G0 2
0 0
1 0
G1 2
0 0
2 0
```

Next, the `fma.col` file contains the set of variable descriptor strings:

```
mass
a
v
```

and the `fma.row` file contains the set of response descriptor strings:

```
force
energy
```

The variable and objective function names declared within AMPL should be a subset of the variable descriptors and response descriptors used by DAKOTA (see the DAKOTA Reference Manual [3] for information on DAKOTA variable and response descriptors). Ordering of the inputs and outputs within the AMPL declaration is not important, as DAKOTA will reorder data as needed. The following listing shows an excerpt from `Dakota/test/dakota_ampl.in`, which demonstrates a combined algebraic/simulation-based mapping in which algebraic mappings from the `fma` definition are overlaid with simulation-based mappings from `text_book`:

```
variables,
        continuous_design = 5
          descriptor    'x1' 'mass' 'a' 'x4' 'v'
          initial_point  0.0  2.0  1.0  0.0  3.0
          lower_bounds  -3.0  0.0 -5.0 -3.0 -5.0
          upper_bounds   3.0 10.0  5.0  3.0  5.0

interface,
        algebraic_mappings = 'ampl/fma.nl'
        system
          analysis_driver = 'text_book'
          parameters_file = 'tb.in'
          results_file    = 'tb.out'
          file_tag

responses,
        response_descriptors = 'force' 'ineq1' 'energy'
        num_objective_functions = 1
        num_nonlinear_inequality_constraints = 1
        num_nonlinear_equality_constraints = 1
        nonlinear_equality_targets = 20.0
        analytic_gradients
        no_hessians
```

Note that the algebraic inputs and outputs are a subset of the total inputs and outputs and that DAKOTA will track the algebraic contributions to the total response set using the order of the descriptor strings. In the case where both the algebraic and simulation-based components contribute to the same function, they are added together.

To solve `text_book` algebraically (refer to Section 2.2 for definition), the following AMPL model file could be used

```
#   Problem : Textbook problem used in DAKOTA testing
#             Constrained quartic, 2 continuous variables
#   Solution: x=(0.5, 0.5), obj = .125, c1 = 0, c2 = 0
#

# continuous variables
var x1 >=  0.5 <= 5.8 := 0.9;
var x2 >= -2.9 <= 2.9 := 1.1;

# objective function
minimize obj: (x1 - 1)^4 + (x2 - 1)^4;

# constraints (current required syntax for DAKOTA/AMPL interface)
minimize c1: x1^2 - 0.5*x2;
minimize c2: x2^2 - 0.5*x1;

# required for output of *.row and *.col files
option auxfiles rc;
```

Note that the nonlinear constraints should not currently be declared as constraints within AMPL. Since the DAKOTA variable bounds and constraint bounds/targets currently take precedence over any AMPL specification, the current approach is to declare all AMPL outputs as objective functions and then map them into the appropriate response function type (objectives, least squares terms, nonlinear inequality/equality constraints, or generic response functions) within the DAKOTA input specification.

12.3 Simulation Interfaces

The invocation of a simulation code is performed using either system calls, forks, or direct function invocations. In the system call and fork cases, a separate process is created for the simulation and communication between DAKOTA and the simulation occurs through parameter and response files. For system call and fork interfaces, the interface section must specify the details of this data transfer. In the direct function case, a separate process is not created and communication occurs directly through the function argument list. Sections 12.3.1 through 12.3.5 provide information on the simulation interfacing approaches.

12.3.1 The Direct Function Simulation Interface

The direct function interface may be used to invoke simulations that are linked into the DAKOTA executable. This interface eliminates overhead from process creation and file I/O and can simplify operations on massively parallel computers. These advantages are balanced with the practicality of converting an existing simulation code into a library with a subroutine interface. Sandia codes for structural dynamics (Salinas), computational fluid dynamics (Sage), and circuit simulation (Xyce) and external codes such as Phoenix Integration's ModelCenter framework and The Mathworks' Matlab have been linked in this way, and a direct interface to Sandia's SIERRA multiphysics framework is under development. In the latter case, the additional effort is particularly justified since SIERRA unifies an entire suite of physics codes. [*Note: the "sandwich implementation" of combining a direct interface plug-in with DAKOTA's library mode is discussed in the DAKOTA Developers Manual [4]*].

In addition to direct linking with simulation codes, the direct interface also provides access to internal polynomial test functions that are used for algorithm performance and regression testing. The following test functions

are available: `cantilever`, `cyl_head`, `log_ratio`, `rosenbrock`, `short_column`, and `text_book` (including `text_book1`, `text_book2`, `text_book3`, and `text_book_ouu`). While these functions are also available as external programs in the `Dakota/test` directory, maintaining internally linked versions allows more rapid testing. See Chapter 23 for additional information on several of these test problems. An example input specification for a direct interface follows:

```
interface,
        direct
          analysis_driver = 'rosenbrock'
```

Additional specification examples are provided in Section 2.4 and additional information on asynchronous usage of the direct function interface is provided in Section 19.2.1.1. Guidance for usage of some particular direct simulation interfaces is in Section 18.3 and the details of adding a simulation code to the direct interface are provided in Section 18.2.

12.3.2 The System Call Simulation Interface

Users are strongly encouraged to use the fork simulation interface if possible, though the system interface is still supported for portability and backward compatibility. The system call approach invokes a simulation code or simulation driver by using the `system` function from the standard C library [72]. In this approach, the system call creates a new process that communicates with DAKOTA through parameter and response files. The system call approach allows the simulation to be initiated via its standard invocation procedure (as a "black box") and then coordinated with a variety of tools for pre- and post-processing. This approach has been widely used in previous studies [29, 31, 23]. The system call approach involves more process creation and file I/O overhead than the direct function approach, but this extra overhead is usually insignificant compared with the cost of a simulation. An example of a system call interface specification follows:

```
interface,
        system
          analysis_driver = 'text_book'
          parameters_file = 'text_book.in'
          results_file    = 'text_book.out'
          file_tag file_save
```

More detailed examples of using the system call interface are provided in Section 2.4.4.1 and in Section 18.1, and information on asynchronous usage of the system call interface is provided in Section 19.2.1.2.

12.3.3 The Fork Simulation Interface

The fork simulation interface uses the `fork`, `exec`, and `wait` families of functions to manage simulation codes or simulation drivers. (In a native MS Windows version of DAKOTA, similar Win32 functions, such as `_spawnvp()`, are used instead.) Calls to `fork` or `vfork` create a copy of the DAKOTA process, `execvp` replaces this copy with the simulation code or driver process, and then DAKOTA uses the `wait` or `waitpid` functions to wait for completion of the new process. Transfer of variables and response data between DAKOTA and the simulator code or driver occurs through the file system in exactly the same manner as for the system call interface. An example of a fork interface specification follows:

```
interface,
        fork
          input_filter    = 'test_3pc_if'
```

```
output_filter   = 'test_3pc_of'
analysis_driver = 'test_3pc_ac'
parameters_file = 'tb.in'
results_file    = 'tb.out'
file_tag
```

Information on asynchronous usage of the fork interface is provided in Section 19.2.1.3.

12.3.4 Syntax for Filter and Driver Strings

With the fork interface, and on most systems, with the system interface as well, the string values supplied for `input_filter`, `output_filter`, and `analysis_driver` can involve simple Bourne-shell syntax for specifying environment values that the filter or driver will see. For example,

```
analysis_driver = 'opfile=myspec outlev=2 mydriver'
```

would cause `mydriver` to be invoked with environment variables `opfile` and `outlev` having the values "myspec" and "2", respectively. If the driver is a shell script, it can access these values as `$opfile` and `$outlev`; a compiled driver can obtain these values from a function; drivers written in C or C++ can use the standard `getenv` function (e.g., invoking `getenv("opfile")`).

Both the values assigned to environment variables and name of the file invoked as filter or driver can contain spaces, provided that the values in question are quoted. Within strings delimited by single quotes, you can use double quotes for quoting, and vice versa. For instance,

```
analysis_driver = 'opfile="my spec" "my driver"'
```

and

```
analysis_driver = "opfile='my spec' 'my driver'"
```

both specify a driver named "`my driver`" and value "`my spec`" for `$opfile`.

12.3.5 Fork or System Call: Which to Use?

The primary operational difference between the fork and system call simulation interfaces is that, in the fork interface, the `fork`/`exec` functions return a process identifier that the `wait`/`waitpid` functions can use to detect the completion of a simulation for either synchronous or asynchronous operations. The system call simulation interface, on the other hand, must use a response file detection scheme for this purpose in the asynchronous case. Thus, an important advantage of the fork interface over the system call interface is that it avoids the potential of a file race condition when employing asynchronous local parallelism (refer to Section 19.2.1). This condition can occur when the responses file has been created but the writing of the response data set to this file has not been completed (see Section 19.2.1.2). While significant care has been taken to manage this file race condition in the system call case, the fork interface still has the potential to be more robust when performing function evaluations asynchronously.

Another advantage of the fork interface is that it has additional asynchronous capabilities when a function evaluation involves multiple analyses. As shown in Table 19.1, the fork interface supports asynchronous local and hybrid parallelism modes for managing concurrent analyses within function evaluations, whereas the system call

interface does not. These additional capabilities again stem from the ability to track child processes by their process identifiers.

The only disadvantage to the fork interface compared with the system interface is that the `fork`/`exec`/`wait` functions are not part of the standard C library, whereas the `system` function is. As a result, support for implementations of the `fork`/`exec`/`wait` functions can vary from platform to platform. At one time, these commands were not available on some of Sandia's massively parallel computers. However, in the more mainstream UNIX environments, availability of `fork`/`exec`/`wait` should not be an issue.

In summary, the system call interface has been a workhorse for many years and is well tested and proven, but the fork interface supports additional capabilities and is recommended when managing asynchronous simulation code executions. Having both interfaces available has proven to be useful on a number of occasions and they will both continue to be supported for the foreseeable future.

12.4 Simulation Interface Components

Figure 12.1 is an extension of Figure 1.1 that adds details of the components that make up each of the simulation interfaces (system call, fork, and direct). These components include an `input_filter` ("IFilter"), one or more `analysis_drivers` ("Analysis Code/Driver"), and an `output_filter` ("OFilter"). The input and output filters provide optional facilities for managing simulation pre- and post-processing, respectively. More specifically, the input filter can be used to insert the DAKOTA parameters into the input files required by the simulator program, and the output filter can be used to recover the raw data from the simulation results and compute the desired response data set. If there is a single analysis code, it is often convenient to combine these pre- and post-processing functions into a single simulation driver script, and the separate input and output filter facilities are rarely used in this case. If there are multiple analysis drivers, however, the input and output filter facilities provide a convenient means for managing *non-repeated* portions of the pre- and post-processing for multiple analyses. That is, pre- and post-processing tasks that must be performed for each analysis can be performed within the individual analysis drivers, and shared pre- and post-processing tasks that are only performed once for the set of analyses can be performed within the input and output filters.

When spawning function evaluations using system calls or forks, DAKOTA must communicate parameter and response data with the analysis drivers and filters through use of the file system. This is accomplished by passing the names of the parameters and results files on the command line when executing an analysis driver or filter. The input filter or analysis driver read data from the parameters file and the output filter or analysis driver write the appropriate data to the responses file. While not essential when the file names are fixed, the file names must be retrieved from the command line when DAKOTA is changing the file names from one function evaluation to the next (i.e., using temporary files or root names tagged with numerical identifiers). In the case of a UNIX C-shell script, the two command line arguments are retrieved using `$argv[1]` and `$argv[2]` (see [5]). Similarly, Bourne shell scripts retrieve the two command line arguments using `$1` and `$2`, and Perl scripts retrieve the two command line arguments using `@ARGV[0]` and `@ARGV[1]`. In the case of a C or C++ program, command line arguments are retrieved using `argc` (argument count) and `argv` (argument vector) [72], and for Fortran 77, the `iargc` function returns the argument count and the `getarg` subroutine returns command line arguments.

12.4.1 Single analysis driver without filters

If a single `analysis_driver` is selected in the interface specification and filters are not needed (as indicated by omission of the `input_filter` and `output_filter` specifications), then only one process will appear in the execution syntax of the simulation interface. An example of this syntax in the system call case is:

Figure 12.1: Components of the simulation interface

```
driver params.in results.out
```

where "`driver`" is the user-specified analysis driver and "`params.in`" and "`results.out`" are the names of the parameters and results files, respectively, passed on the command line. In this case, the user need not retrieve the command line arguments since the same file names will be used each time.

For the same mapping, the fork simulation interface echoes the following syntax:

```
blocking fork: driver params.in results.out
```

for which only a single blocking fork is needed to perform the evaluation.

Executing the same mapping with the direct simulation interface results in an echo of the following syntax:

```
Direct function: invoking driver
```

where this analysis driver must be linked as a function within DAKOTA's direct interface (see Section 18.2). Note that no parameter or response files are involved, since such values are passed directly through the function argument lists.

Both the system call and fork interfaces support asynchronous operations. The asynchronous system call execution syntax involves executing the system call in the background:

```
driver params.in.1 results.out.1 &
```

and the asynchronous fork execution syntax involves use of a nonblocking fork:

```
nonblocking fork: driver params.in.1 results.out.1
```

where file tagging (see Section 12.5.2) has been user-specified in both cases to prevent conflicts between concurrent analysis drivers. In these cases, the user must retrieve the command line arguments since the file names

change on each evaluation. Execution of the direct interface must currently be performed synchronously since multithreading is not yet supported (see Section 19.2.1.1).

12.4.2 Single analysis driver with filters

When filters are used, the syntax of the system call that DAKOTA performs is:

```
ifilter params.in results.out; driver params.in results.out;
     ofilter params.in results.out
```

in which the input filter ("`ifilter`"), analysis driver ("`driver`"), and output filter ("`ofilter`") processes are combined into a single system call through the use of semi-colons (see [5]). All three portions are passed the names of the parameters and results files on the command line.

For the same mapping, the fork simulation interface echoes the following syntax:

```
blocking fork: ifilter params.in results.out;
     driver params.in results.out; ofilter params.in results.out
```

where a series of three blocking forks is used to perform the evaluation.

Executing the same mapping with the direct simulation interface results in an echo of the following syntax:

```
Direct function: invoking { ifilter driver ofilter }
```

where each of the three components must be linked as a function within DAKOTA's direct interface. Since asynchronous operations are not yet supported, execution simply involves invocation of each of the three linked functions in succession. Again, no files are involved since parameter and response data are passed directly through the function argument lists.

Asynchronous executions would appear as follows for the system call interface:

```
(ifilter params.in.1 results.out.1; driver params.in.1 results.out.1;
     ofilter params.in.1 results.out.1) &
```

and, for the fork interface, as:

```
nonblocking fork: ifilter params.in.1 results.out.1;
     driver params.in.1 results.out.1; ofilter params.in.1 results.out.1
```

where file tagging of evaluations has again been user-specified in both cases. For the system call simulation interface, use of parentheses and semi-colons to bind the three processes into a single system call simplifies asynchronous process management compared to an approach using separate system calls. The fork simulation interface, on the other hand, does not rely on parentheses and accomplishes asynchronous operations by first forking an intermediate process. This intermediate process is then reforked for the execution of the input filter, analysis driver, and output filter. The intermediate process can be blocking or nonblocking (nonblocking in this case), and the second level of forks can be blocking or nonblocking (blocking in this case). The fact that forks can be reforked multiple times using either blocking or nonblocking approaches provides the enhanced flexibility to support a variety of local parallelism approaches (see Chapter 19).

12.4.3 Multiple analysis drivers without filters

If a list of `analysis_drivers` is specified and filters are not needed (i.e., neither `input_filter` nor `output_filter` appears), then the system call syntax would appear as:

```
driver1 params.in results.out.1; driver2 params.in results.out.2;
     driver3 params.in results.out.3
```

where "`driver1`", "`driver2`", and "`driver3`" are the user-specified analysis drivers and "`params.in`" and "`results.out`" are the user-selected names of the parameters and results files. Note that the results files for the different analysis drivers have been automatically tagged to prevent overwriting. This automatic tagging of *analyses* (see Section 12.5.4) is a separate operation from user-selected tagging of *evaluations* (see Section 12.5.2).

For the same mapping, the fork simulation interface echoes the following syntax:

```
blocking fork: driver1 params.in results.out.1;
     driver2 params.in results.out.2; driver3 params.in results.out.3
```

for which a series of three blocking forks is needed (no reforking of an intermediate process is required).

Executing the same mapping with the direct simulation interface results in an echo of the following syntax:

```
Direct function: invoking { driver1 driver2 driver3 }
```

where, again, each of these components must be linked within DAKOTA's direct interface and no files are involved for parameter and response data transfer.

Both the system call and fork interfaces support asynchronous function evaluations. The asynchronous system call execution syntax would be reported as

```
(driver1 params.in.1 results.out.1.1; driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3) &
```

and the nonblocking fork execution syntax would be reported as

```
nonblocking fork: driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2; driver3 params.in.1 results.out.1.3
```

where, in both cases, file tagging of evaluations has been user-specified to prevent conflicts between concurrent analysis drivers and file tagging of the results files for multiple analyses is automatically used. In the fork interface case, an intermediate process is forked to allow a non-blocking function evaluation, and this intermediate process is then reforked for the execution of each of the analysis drivers.

12.4.4 Multiple analysis drivers with filters

Finally, when combining filters with multiple `analysis_drivers`, the syntax of the system call that DAKOTA performs is:

```
ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1
```

in which all processes have again been combined into a single system call through the use of semi-colons and parentheses. Note that the secondary file tagging for the results files is only used for the analysis drivers and not for the filters. This is consistent with the filters' defined purpose of managing the non-repeated portions of analysis pre- and post-processing (e.g., overlay of response results from individual analyses; see Section 12.5.4 for additional information).

For the same mapping, the fork simulation interface echoes the following syntax:

```
blocking fork: ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1
```

for which a series of five blocking forks is used (no reforking of an intermediate process is required).

Executing the same mapping with the direct simulation interface results in an echo of the following syntax:

```
Direct function: invoking { ifilter driver1 driver2 driver3 ofilter }
```

where each of these components must be linked as a function within DAKOTA's direct interface. Since asynchronous operations are not supported, execution simply involves invocation of each of the five linked functions in succession. Again, no files are involved for parameter and response data transfer since this data is passed directly through the function argument lists.

Asynchronous executions would appear as follows for the system call interface:

```
(ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1) &
```

and for the fork interface:

```
nonblocking fork: ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1
```

where, again, user-selected file tagging of evaluations is combined with automatic file tagging of analyses. In the fork interface case, an intermediate process is forked to allow a non-blocking function evaluation, and this intermediate process is then reforked for the execution of the input filter, each of the analysis drivers, and the output filter.

12.5 Simulation File Management

This section describes some management features used for files that transfer data between DAKOTA and simulation codes (i.e., when the system call or fork interfaces are used). These features can generate unique filenames when DAKOTA executes programs in parallel and can help one debug the interface between DAKOTA and a simulation code.

12.5.1 File Saving

Before driver execution: In DAKOTA 5.0 and newer, an existing results file will be removed immediately prior to executing the analysis driver. This new behavior addresses a common user problem resulting from starting DAKOTA with stale results files in the run directory. To override this default behavior and preserve any existing results files, specify `allow_existing_results`.

After driver execution: The `file_save` option in the interface specification allows the user to control whether parameters and results files are retained or removed from the working directory after the analysis completes. DAKOTA's default behavior is to remove files once their use is complete to reduce clutter. If the method output setting is verbose, a file remove notification will follow the function evaluation echo, e.g.,

```
driver /usr/tmp/aaaa20305 /usr/tmp/baaa20305
Removing /usr/tmp/aaaa20305 and /usr/tmp/baaa20305
```

However, if `file_save` appears in the interface specification, these files will not be removed. This latter behavior is often useful for debugging communication between DAKOTA and simulator programs. An example of a `file_save` specification is shown in the file tagging example below.

12.5.2 File Tagging for Evaluations

When a user provides `parameters_file` and `results_file` specifications, the `file_tag` option in the interface specification causes DAKOTA to make the names of these files unique by appending the function evaluation number to the root file names. Default behavior is to not tag these files, which has the advantage of allowing the user to ignore command line argument passing and always read to and write from the same file names. However, it has the disadvantage that files may be overwritten from one function evaluation to the next. When `file_tag` appears in the interface specification, the file names are made unique by the appended evaluation number. This uniqueness requires the user's interface to get the names of these files from the command line. The file tagging feature is most often used when concurrent simulations are running in a common disk space, since it can prevent conflicts between the simulations. An example specification of `file_tag` and `file_save` is shown below:

```
interface,
        system
          analysis_driver = 'text_book'
          parameters_file = 'text_book.in'
          results_file    = 'text_book.out'
          file_tag file_save
```

Special case: When a user specifies names for the parameters and results files and `file_save` is used without `file_tag`, untagged files are used in the function evaluation but are then moved to tagged files after the function evaluation is complete, to prevent overwriting files for which a `file_save` request has been given. If the output control is set to verbose, then a notification similar to the following will follow the function evaluation echo:

```
driver params.in results.out
Files with non-unique names will be tagged to enable file_save:
Moving params.in to params.in.1
Moving results.out to results.out.1
```

12.5.3 Temporary Files

If parameters_file and results_file are not specified by the user, temporary files having generated names are used. For example, a system call to a single analysis driver might appear as:

```
driver /usr/tmp/aaaa20305 /usr/tmp/baaa20305
```

and a system call to an analysis driver with filter programs might appear as:

```
ifilter /usr/tmp/aaaa22490 usr/tmp/baaa22490;
     driver /usr/tmp/aaaa22490 usr/tmp/baaa22490;
     ofilter /usr/tmp/aaaa22490 /usr/tmp/baaa22490
```

These files have unique names created by the tmpnam utility from the C standard library [72]. This uniqueness requires the user's interface to get the names of these files from the command line. File tagging with evaluation number is unnecessary with temporary files (since they are already unique); thus, file_tag requests will be ignored. A file_save request will be honored, but it should be used with care since the temporary file directory could easily become cluttered without the user noticing.

12.5.4 File Tagging for Analysis Drivers

When multiple analysis drivers are involved in performing a function evaluation with either the system call or fork simulation interface, a secondary file tagging is *automatically* used to distinguish the results files used for the individual analyses. This applies to both the case of user-specified names for the parameters and results files and the default temporary file case. Examples for the former case were shown previously in Section 12.4.3 and Section 12.4.4. The following examples demonstrate the latter temporary file case. Even though Unix temporary files have unique names for a particular function evaluation, tagging is still needed to manage the individual contributions of the different analysis drivers to the response results, since the same root results filename is used for each component. For the system call interface, the syntax would be similar to the following:

```
ifilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ;
     driver1 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.1;
     driver2 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.2;
     driver3 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.3;
     ofilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ
```

and, for the fork interface, similar to:

```
blocking fork:
     ifilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ;
     driver1 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.1;
     driver2 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.2;
     driver3 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.3;
     ofilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ
```

Tagging of results files with an analysis identifier is needed since each analysis driver must contribute a user-defined subset of the total response results for the evaluation. If an output filter is not supplied, DAKOTA will combine these portions through a simple overlaying of the individual contributions (i.e., summing the results in /var/tmp/baaxkaOKZ.1, /var/tmp/baaxkaOKZ.2, and /var/tmp/baaxkaOKZ.3). If this simple approach is inadequate, then an output filter should be supplied to perform the combination. This is the reason

why the results file for the output filter does not use analysis tagging; it is responsible for the results combination (i.e., combining `/var/tmp/baaxkaOKZ.1`, `/var/tmp/baaxkaOKZ.2`, and `/var/tmp/baaxkaOKZ.3` into `/var/tmp/baaxkaOKZ`). In this case, DAKOTA will read only the results file from the output filter (i.e., `/var/tmp/baaxkaOKZ`) and interpret it as the total response set for the evaluation.

Parameters files are not currently tagged with an analysis identifier. This reflects the fact that DAKOTA does not attempt to subdivide the requests in the active set vector for different analysis portions. Rather, the total active set vector is passed to each analysis driver and the appropriate subdivision of work *must be defined by the user*. This allows the division of labor to be very flexible. In some cases, this division might occur across response functions, with different analysis drivers managing the data requests for different response functions. And in other cases, the subdivision might occur within response functions, with different analysis drivers contributing portions to each of the response functions. The only restriction is that each of the analysis drivers must follow the response format dictated by the total active set vector. For response data for which an analysis driver has no contribution, 0's must be used as placeholders.

12.5.5 Work Directories

Sometimes it is convenient for simulators and filters to run in a directory different from the one where DAKOTA is invoked. For instance, when performing concurrent evaluations and/or analyses, it is often necessary to cloister input and output files in separate directories to avoid conflicts. A simulator script used as an `analysis_driver` can of course include commands to change to a different directory if desired (while still arranging to write a results file in the original directory), but DAKOTA has facilities that may simplify the creation of simulator scripts. When the `work_directory` feature is enabled, DAKOTA will create a directory for each evaluation/analysis (with optional tagging and saving as with files). To enable the `work_directory` feature an interface specification includes the keyword

```
work_directory
```

then DAKOTA will arrange for the simulator and any filters to wake up in the work directory, with $PATH adjusted (if necessary) so programs that could be invoked without a relative path to them (i.e., by a name not involving any slashes) from DAKOTA's directory can also be invoked from the simulator's (and filter's) directory. On occasion, it is convenient for the simulator to have various files, e.g., data files, available in the directory where it runs. If, say, `my/special/directory` is such a directory (as seen from DAKOTA's directory), the interface specification

```
work_directory named 'my/special/directory'
```

would cause DAKOTA to start the simulator and any filters in that directory. If the directory did not already exist, DAKOTA would create it and would remove it after the simulator (or output filter, if specifed) finished, unless instructed not to do so by the appearance of `directory_save` or its synonym `dir_save` in the interface specification. If `named '...'` does not appear, then `directory_save` cannot appear either, and DAKOTA creates a temporary directory (using the `tmpnam` function to determine its name) for use by the simulator and any filters. If you specify `directory_tag` (or `dir_tag`), DAKOTA causes each invocation of the simulator and any filters to start in a a subdirectory of the work directory with a name composed of the work directory's name followed by a period and the invocation number (1, 2, ...); this might be useful in debugging.

Sometimes it can be helpful for the simulator and filters to start in a new directory populated with some files. Adding

```
template_directory 'my/template'
```

to the work directory specification would cause the contents of directory `my/template` to be linked recursively into the work directory. Linking makes sense if files are large, but when practical, it is far more reliable to have copies of the files; adding `copy` to the specification would cause the contents of the template directory to be copied recursively to the work directory. The linking or copying does not replace existing files unless `replace` also appears in the specification. Instead of `template_directory` ..., you can specify `template_files`, followed by one or more quoted strings, as in

```
template_files 'zip' 'zap' 'foo/zot'
```

which would cause `zip`, `zap`, and `foo/zot` to be linked (or, with `copy`, copied) recursively to the work directory.

Here is a summary of possibilities for a work directory specification, with `[...]` denoting that ... is optional:

```
work_directory [ named '...' ]
  [ directory_tag ]     # or dir_tag
  [ directory_save ]    # or dir_save
  [ template_directory '...' # or template_files '...' '...' ...
     [ copy ]
     [ replace ]
     ]
```

12.6 Parameter to Response Mappings

In this section, interface mapping examples are presented through the discussion of several parameters files and their corresponding results files. A typical input file for 2 variables ($n = 2$) and 3 functions ($m = 3$) using the standard parameters file format (see Section 11.6.1) is as follows:

```
                     2 variables
1.500000000000000e+00 cdv_1
1.500000000000000e+00 cdv_2
                     3 functions
                     1 ASV_1
                     1 ASV_2
                     1 ASV_3
                     2 derivative_variables
                     1 DVV_1
                     2 DVV_2
                     0 analysis_components
```

where numerical values are associated with their tags within "`value tag`" constructs. The number of design variables (n) and the string "`variables`" are followed by the values of the design variables and their tags, the number of functions (m) and the string "`functions`", the active set vector (ASV) and its tags, the number of derivative variables and the string "`derivative_variables`", the derivative variables vector (DVV) and its tags, the number of analysis components and the string "`analysis_components`", and the analysis components array and its tags. The descriptive tags for the variables are always present and they are either the descriptors in the user's variables specification, if given there, or are default descriptors. The length of the active set vector is equal to the number of functions (m). In the case of an optimization data set with an objective function and two nonlinear constraints (three response functions total), the first ASV value is associated with the objective function and the remaining two are associated with the constraints (in whatever consistent constraint order has been defined

by the user). The DVV defines a subset of the variables used for computing derivatives. Its identifiers are 1-based and correspond to the full set of variables listed in the first array. Finally, the analysis components pass additional strings from the user's `analysis_components` specification in a DAKOTA input file through to the simulator. They allow the development of simulation drivers that are more flexible, by allowing them to be passed additional specifics at run time, e.g., the names of model files such as a particular mesh to use.

For the APREPRO format option (see Section 11.6.2), the same set of data appears as follows:

```
{ DAKOTA_VARS     =                     2 }
{ cdv_1           =  1.500000000000000e+00 }
{ cdv_2           =  1.500000000000000e+00 }
{ DAKOTA_FNS      =                     3 }
{ ASV_1           =                     1 }
{ ASV_2           =                     1 }
{ ASV_3           =                     1 }
{ DAKOTA_DER_VARS =                     2 }
{ DVV_1           =                     1 }
{ DVV_2           =                     2 }
{ DAKOTA_AN_COMPS =                     0 }
```

where the numerical values are associated with their tags within "`{ tag = value }`" constructs.

The user-supplied simulation interface, comprised of a simulator program or driver and (optionally) filter programs, is responsible for reading the parameters file and creating a results file that contains the response data requested in the ASV. This response data is written in the format described in Section 13.2. Since the ASV contains all ones in this case, the response file corresponding to the above input file would contain values for the three functions:

```
1.250000000000000e-01 f
1.500000000000000e+00 c1
1.500000000000000e+00 c2
```

Since function tags are optional, the following would be equally acceptable:

```
1.250000000000000e-01
1.500000000000000e+00
1.500000000000000e+00
```

For the same parameters with different ASV components,

```
                    2 variables
1.500000000000000e+00 cdv_1
1.500000000000000e+00 cdv_2
                    3 functions
                    3 ASV_1
                    3 ASV_2
                    3 ASV_3
                    2 derivative_variables
                    1 DVV_1
                    2 DVV_2
                    0 analysis_components
```

the following response data is required:

```
1.250000000000000e-01 f
1.500000000000000e+00 c1
1.500000000000000e+00 c2
[ 5.000000000000000e-01 5.000000000000000e-01 ]
[ 3.000000000000000e+00 -5.000000000000000e-01 ]
[ -5.000000000000000e-01 3.000000000000000e+00 ]
```

Here, we need not only the function values, but also each of their gradients. The derivatives are computed with respect to `cdv_1` and `cdv_2` as indicated by the DVV values. Another modification to the ASV components yields the following parameters file:

```
                     2 variables
1.500000000000000e+00 cdv_1
1.500000000000000e+00 cdv_2
                     3 functions
                     2 ASV_1
                     0 ASV_2
                     2 ASV_3
                     2 derivative_variables
                     1 DVV_1
                     2 DVV_2
                     0 analysis_components
```

for which the following results file is needed:

```
[ 5.000000000000000e-01 5.000000000000000e-01 ]
[ -5.000000000000000e-01 3.000000000000000e+00 ]
```

Here, we need gradients for functions `f` and `c2`, but not for `c1`, presumably since this constraint is inactive.

A full Newton optimizer might make the following request:

```
                     2 variables
1.500000000000000e+00 cdv_1
1.500000000000000e+00 cdv_2
                     1 functions
                     7 ASV_1
                     2 derivative_variables
                     1 DVV_1
                     2 DVV_2
                     0 analysis_components
```

for which the following results file,

```
1.250000000000000e-01 f
[ 5.000000000000000e-01 5.000000000000000e-01 ]
[[ 3.000000000000000e+00 0.000000000000000e+00
   0.000000000000000e+00 3.000000000000000e+00 ]]
```

containing the objective function, its gradient vector, and its Hessian matrix, is needed. Again, the derivatives (gradient vector and Hessian matrix) are computed with respect to `cdv_1` and `cdv_2` as indicated by the DVV values.

Lastly, a more advanced example could have multiple types of variables present; in this example, 2 continuous design and 3 discrete design range, 2 normal uncertain, and 3 continuous state and 2 discrete state range variables. When a mixture of variable types is present, the content of the DVV (and therefore the required length of gradient vectors and Hessian matrices) depends upon the type of study being performed (see Section 13.3). For a reliability analysis problem, the uncertain variables are the active continuous variables and the following parameters file would be typical:

```
                    12 variables
1.500000000000000e+00 cdv_1
1.500000000000000e+00 cdv_2
                     2 ddriv_1
                     2 ddriv_2
                     2 ddriv_3
5.000000000000000e+00 nuv_1
5.000000000000000e+00 nuv_2
3.500000000000000e+00 csv_1
3.500000000000000e+00 csv_2
3.500000000000000e+00 csv_3
                     4 dsriv_1
                     4 dsriv_2
                     3 functions
                     3 ASV_1
                     3 ASV_2
                     3 ASV_3
                     2 derivative_variables
                     6 DVV_1
                     7 DVV_2
                     2 analysis_components
            mesh1.exo AC_1
              db1.xml AC_2
```

Gradients are requested with respect to variable entries 6 and 7, which correspond to normal uncertain variables `nuv_1` and `nuv_2`. The following response data would be appropriate:

```
7.943125000000000e+02 f
1.500000000000000e+00 c1
1.500000000000000e+00 c2
[ 2.560000000000000e+02 2.560000000000000e+02 ]
[ 0.000000000000000e+00 0.000000000000000e+00 ]
[ 0.000000000000000e+00 0.000000000000000e+00 ]
```

In a parameter study, however, no distinction is drawn between different types of continuous variables, and derivatives would be needed with respect to all continuous variables ($n_{dvv} = 7$ for the continuous design variables `cdv_1` and `cdv_2`, the normal uncertain variables `nuv_1` and `nuv_2`, and the continuous state variables `csv_1`, `csv_2` and `csv_3`). The parameters file would appear as

```
                    12 variables
1.500000000000000e+00 cdv_1
1.500000000000000e+00 cdv_2
                     2 ddriv_1
                     2 ddriv_2
                     2 ddriv_3
5.000000000000000e+00 nuv_1
```

```
5.000000000000000e+00 nuv_2
3.500000000000000e+00 csv_1
3.500000000000000e+00 csv_2
3.500000000000000e+00 csv_3
                    4 dsriv_1
                    4 dsriv_2
                    3 functions
                    3 ASV_1
                    3 ASV_2
                    3 ASV_3
                    7 derivative_variables
                    1 DVV_1
                    2 DVV_2
                    6 DVV_3
                    7 DVV_4
                    8 DVV_5
                    9 DVV_6
                   10 DVV_7
                    2 analysis_components
             mesh1.exo AC_1
               db1.xml AC_2
```

and the corresponding results would appear as

```
7.943125000000000e+02 f
1.500000000000000e+00 c1
1.500000000000000e+00 c2
[  5.000000000000000e-01  5.000000000000000e-01  2.560000000000000e+02
   2.560000000000000e+02  6.250000000000000e+01  6.250000000000000e+01
   6.250000000000000e+01 ]
[  3.000000000000000e+00 -5.000000000000000e-01  0.000000000000000e+00
   0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
   0.000000000000000e+00 ]
[ -5.000000000000000e-01  3.000000000000000e+00  0.000000000000000e+00
   0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
   0.000000000000000e+00 ]
```

Chapter 13

Responses

13.1 Overview

The `responses` specification in a DAKOTA input file controls the types of data that can be returned from an interface during DAKOTA's execution. The specification includes the number and type of response functions (objective functions, nonlinear constraints, calibration terms, etc.) as well as availability of first and second derivatives (gradient vectors and Hessian matrices) for these response functions.

This chapter will present a brief overview of the response data sets and their uses, as well as cover some user issues relating to file formats and derivative vector and matrix sizing. For a detailed description of responses section syntax and example specifications, refer to the Responses Commands chapter in the DAKOTA Reference Manual [3].

13.1.1 Response function types

The types of response functions listed in the responses specification should be consistent with the iterative technique called for in the method specification:

- an optimization data set comprised of `num_objective_functions`, `num_nonlinear_inequality_constraints`, and `num_nonlinear_equality_constraints`. This data set is appropriate for use with optimization methods (e.g., the methods in Section 3.5).

- a calibration data set comprised of `calibration_terms`, `num_nonlinear_inequality_constraints`, and `num_nonlinear_equality_constraints`. This data set is appropriate for use with nonlinear least squares algorithms (e.g., the methods in Section 3.7).

- a generic data set comprised of `num_response_functions`. This data set is appropriate for use with uncertainty quantification methods (e.g., the methods in Section 3.4).

Certain general-purpose iterative techniques, such as parameter studies and design of experiments methods, can be used with any of these data sets.

13.1.2 Gradient availability

Gradient availability for these response functions may be described by:

- `no_gradients`: gradients will not be used.
- `numerical_gradients`: gradients are needed and will be approximated by finite differences.
- `analytic_gradients`: gradients are needed and will be supplied by the simulation code (without any finite differencing by DAKOTA).
- `mixed_gradients`: the simulation will supply some gradient components and DAKOTA will approximate the others by finite differences.

The gradient specification also links back to the iterative method used. Gradients commonly are needed when the iterative study involves gradient-based optimization, reliability analysis for uncertainty quantification, or local sensitivity analysis.

13.1.3 Hessian availability

Hessian availability for the response functions is similar to the gradient availability specifications, with the addition of support for "quasi-Hessians":

- `no_hessians`: Hessians will not be used.
- `numerical_gradients`: Hessians are needed and will be approximated by finite differences. These finite differences may be involve first-order differences of gradients (if analytic gradients are available for the response function of interest) or second-order differences of function values (in all other cases).
- `quasi_hessians`: Hessians are needed and will be approximated by secant updates (BFGS or SR1) from a series of gradient evaluations.
- `analytic_hessians`: Hessians are needed and are available directly from the simulation code.
- `mixed_hessians`: Hessians are needed and will be obtained from a mix of numerical, analytic, and "quasi" sources.

The Hessian specification also links back to the iterative method in use; Hessians commonly would be used in gradient-based optimization by full Newton methods or in reliability analysis with second-order limit state approximations or second-order probability integrations.

13.2 DAKOTA Results File Data Format

Simulation interfaces using system calls and forks to create separate simulation processes must communicate with the simulation through the file system. This is done by reading and writing files of parameters and results. DAKOTA uses its own format for this data input/output. For the results file, only one format is supported (versus the two parameter-file formats described in Section 11.6). Ordering of response functions is as listed in Section 13.1.1 (e.g., objective functions or calibration terms are first, followed by nonlinear inequality constraints, followed by nonlinear equality constraints).

```
<double> <fn_tag1>
<double> <fn_tag2>
...
<double> <fn_tagm>
[ <double> <double> .. <double> ]
[ <double> <double> .. <double> ]
...
[ <double> <double> .. <double> ]
[[ <double> <double> .. <double> ]]
[[ <double> <double> .. <double> ]]
...
[[ <double> <double> .. <double> ]]
```

Figure 13.1: Results file data format.

After a simulation, DAKOTA expects to read a file containing responses reflecting the current parameters and corresponding to the function requests in the active set vector. The response data must be in the format shown in Figure 13.1.

The first block of data (shown in black) conveys the requested function values and is followed by a block of requested gradients (shown in blue), followed by a block of requested Hessians (shown in red). If the amount of data in the file does not match the function request vector, DAKOTA will abort execution with an error message.

Function values have no bracket delimiters and *optionally* one character-string tag per function can be supplied. These tags are not used by DAKOTA and are only included as an optional field for consistency with the parameters file format and for backwards compatibility. If tags are used, they must be separated from numeric function values by white space (one or more blanks, tabs, or newline characters) and there must not be any white space embedded within a character-string tag (e.g., use "`variable1`" or "`variable_1`," but not "`variable 1`").

Gradient vectors are surrounded by single brackets [...n_{dvv}-vector of doubles...]. Tags are not used and must not be present. White space separating the brackets from the data is optional.

Hessian matrices are surrounded by double brackets [[...$n_{dvv} \times n_{dvv}$ matrix of doubles...]]. Hessian components (numeric values for second partial derivatives) are listed by rows and separated by white space; in particular, they can be spread across multiple lines for readability. Tags are not used and must not be present. White space after the initial double bracket and before the final one is optional, but none can appear within the double brackets.

The format of the numeric fields may be floating point or scientific notation. In the latter case, acceptable exponent characters are "`E`" or "`e`." A common problem when dealing with Fortran programs is that a C++ read of a numeric field using "`D`" or "`d`" as the exponent (i.e., a double precision value from Fortran) may fail or be truncated. In this case, the "`D`" exponent characters must be replaced either through modifications to the Fortran source or compiler flags or through a separate post-processing step (e.g., using the UNIX `sed` utility).

13.3 Active Variables for Derivatives

An important question for proper management of both gradient and Hessian data is: if several different types of variables are used, *for which variables are response function derivatives needed?* That is, how is n_{dvv} determined? The short answer is that the derivative variables vector (DVV) specifies the set of variables to be used for

Table 13.1: Variable views for different iterators.

Method	Active view	Derivative variables
branch and bound	Merged Design	$n_{cdv} + n_{ddiv} + n_{ddrv}$
optimization, nonlinear least squares	Mixed Design	n_{cdv}
sampling (standard mode)	Mixed Uncertain	$n_{cauv} + n_{ceuv}$
local reliability, global reliability (standard mode), stochastic expansion (standard mode)	Mixed Aleatory Uncertain	n_{cauv}
interval estimation, evidence	Mixed Epistemic Uncertain	n_{ceuv}
parameter studies, design of experiments, uncertainty quantification (all_variables mode)	Mixed All	$n_{cdv} + n_{cauv} + n_{ceuv} + n_{csv}$

computing derivatives, and n_{dvv} is the length of this vector. The long answer is that, in most cases, the DVV is defined directly from the set of active continuous variables for the iterative method in use.

Since methods determine what subset, or view, of the variables is active in the iteration, it is this same set of variables for which derivatives are most commonly computed (see also Section 11.5). Derivatives are never needed with respect to any discrete variables (since these derivatives do not in general exist) and the active continuous variables depend on the type of study being performed. For optimization and calibration problems, the active continuous variables are the *continuous design variables* ($n_{dvv} = n_{cdv}$), since they are the variables the minimizer manipulates. Similarly, for uncertainty quantification methods that use gradient and/or Hessian information, the active continuous variables are the *continuous uncertain variables* ($n_{dvv} = n_{cauv}$ for aleatory methods, $n_{dvv} = n_{ceuv}$ for epistemic methods, $n_{dvv} = n_{cauv} + n_{ceuv}$ for methods that handle both), with the exception of `all_variables` mode. And lastly, parameter study methods that are cataloging gradient and/or Hessian information do not draw a distinction among continuous variables; therefore, the active continuous variables are defined from *all continuous variables* that are specified ($n_{dvv} = n_{cdv} + n_{cauv} + n_{ceuv} + n_{csv}$). Additional detail on these variables views is provided in Table 13.1.

In a few cases, derivatives are needed with respect to the *inactive* continuous variables. This occurs for nested iteration where a top-level iterator sets derivative requirements (with respect to its active continuous variables) on the final solution of the lower-level iterator (for which the top-level active variables are inactive). For example, in an uncertainty analysis within a nested design under uncertainty algorithm, derivatives of the lower-level response functions may be needed with respect to the design variables, which are active continuous variables at the top level but are inactive within the uncertainty quantification. These instances are the reason for the creation and inclusion of the DVV vector — to clearly indicate the variables whose partial derivatives are needed.

In all cases, if the DVV is honored, then the correct derivative components are returned. In simple cases, such as optimization and calibration studies that only specify design variables and for nondeterministic analyses that only specify uncertain variables, derivative component subsets are not an issue and the exact content of the DVV may be safely ignored.

Chapter 14

Inputs to DAKOTA

14.1 Overview of Inputs

DAKOTA supports a number of command-line arguments, as described in Section 2.1.2. Among these are specifications for the DAKOTA input file and, optionally, a restart file. The syntax of the DAKOTA input file is described in detail in the DAKOTA Reference Manual [3], and the restart file is described in Chapter 21.

A DAKOTA input file may be prepared manually with a text editor such as Emacs, Vi, or WordPad, or it may be defined with the JAGUAR DAKOTA graphical user interface. The JAGUAR DAKOTA GUI is built on the Java-based Eclipse Framework [1] and presents the DAKOTA input specification options in synchronized text-editing and graphical views. JAGUAR includes templates and wizards for helping create DAKOTA studies and can invoke DAKOTA to run an analysis. The DAKOTA GUI for Linux, Windows, and Mac, is available for download from the DAKOTA website `http://dakota.sandia.gov/`, along with licensing information, separate GUI documentation, and installation tips.

14.1.1 Tabular Data Formats

The DAKOTA input file and/or command line may identify additional files for data import as described in Section 14.2. Some of these files are in tabular data format.

DAKOTA versions 5.1+ (October 2011) and newer use two formats for tabular data file input and output. Tabular data refer to numeric data in text form related to, e.g., tabular graphics data, least squares and Bayesian calibration data, samples/points files for constructing surrogates, pre-run output, and post-run input. Both formats are written/read with C++ stream operators/conversions, so most integer and floating point formats are acceptable for numeric data. The formats are:

- **Annotated Matrix** (default for all I/O; specified via `annotated`): text file with one leading row of comments/column labels and one leading column of evaluation/row IDs surrounding num_rows x num_cols whitespace-separated numeric data, (newlines separating rows are not currently required, but may be in the future). The numeric data in a row may correspond to variables, variables followed by responses, data point for calibration, etc., depending on context.
- **Free-form Matrix** (optional; previously default for samples files and least squares data; specified via `freeform`): text file with no leading row and no leading column. The num_rows x num_cols total numeric

data entries may appear separated with any whitespace including arbitrary spaces, tabs, and newlines. In this format, vectors may therefore appear as a single row or single column (or mixture; entries will populate the vector in order).

Attention: Prior to October 2011, calibration and surrogate data files were free-form format. They now default to annotated format, though `freeform` remains an option. For both formats, a warning will be generated if a specific number of data are expected, but extra is found and an error generated when there is insufficient data. Some TPLs like SCOLIB and JEGA manage their own file I/O and only support the free-form option.

14.2 Data Imports

The DAKOTA input file and/or command line may identify additional files used to import data into DAKOTA.

14.2.1 AMPL algebraic mappings: stub.nl, stub.row, and stub.col

As described in Section 12.2, an AMPL specification of algebraic input-to-output relationships may be imported into DAKOTA and used to define or augment the mappings of a particular interface.

14.2.2 Genetic algorithm population import

Genetic algorithms (GAs) from the JEGA and SCOLIB packages support a population import feature using the keywords `initialization_type flat_file = STRING`. This is useful for warm starting GAs from available data or previous runs. Refer to the Method Specification chapter in the DAKOTA Reference Manual [3] for additional information on this specification. The flat file must be in free-form format as described in Section 14.1.1.

14.2.3 Calibration (least squares) data import

Deterministic least squares and Bayesian Calibration methods allow specification of experimental observations to difference with responses in calculating a model misfit metric (such as a sum of squared residuals). The default file format is `annotated`, but the `freeform` option is supported. The data file should contain one row per experiment. The columns of the data file contain, in sequence

- **configuration variables (optional):** state variable values indicating the configuration at which this experiment was conducted; length must agree with thee number of state variables active in the study.
- **experimental observations (required):** experimental data values to difference with model responses; length number of responses.
- **experimental standard deviations (optional):** measurement errors (standard deviations) associated with the data; length 1 (same value for each model response) or num_responses.

See 8.3 and the DAKOTA Reference Manual Responses section for further details.

14.2.4 PCE coefficient import

Polynomial chaos expansion (PCE) methods compute coefficients for response expansions which employ a basis of multivariate orthogonal polynomials. Normally, the `polynomial_chaos` method calculates these coefficients based either on a spectral projection or a linear regression (see Section 6.4). However, DAKOTA also supports the option of importing a set of response PCE coefficients based on the specification `expansion_import_file = STRING`. This is useful for evaluating moments analytically or computing probabilities numerically from a known response expansion. Refer to the Method Specification chapter in the DAKOTA Reference Manual [3] for additional information on this specification.

14.2.5 Surrogate construction from data files

Global data fit surrogates may be constructed from a variety of data sources. One of these sources is an auxiliary data file, as specified by the keywords `reuse_samples points_file = STRING`. The file may be in annotated (default) or free-form format with columns corresponding to variables and responses. Refer to the Model Specification chapter in the DAKOTA Reference Manual [3] for additional information on this specification.

14.2.6 Variables/responses import to post-run

The post-run mode (supported only for sampling, parameter study, and DACE methods) requires specification of a file containing parameter and response data in annotated tabular format (see Section 14.1.1; free-form is not supported). An evaluation ID column is followed by columns for variables, then those for responses, with an ignored header row of labels and then one row per evaluation. Typically this file would be generated by executing `dakota -i dakota.in -pre_run ::variables.dat` and then adding columns of response data to variables.dat to make varsresponses.dat. The file is specified at the command line with:

```
dakota -i dakota.in -post_run varsresponses.dat::
```

Chapter 15

Output from DAKOTA

15.1 Overview of Output Formats

Given an emphasis on complex numerical simulation codes that run on massively parallel supercomputers, DAKOTA's output has been designed to provide a succinct, text-based reporting of the progress of the iterations and function evaluations performed by an algorithm. In addition, DAKOTA provides a tabular output format that is useful for data visualization with external tools and a basic graphical output capability that is useful as a monitoring tool. The JAGUAR DAKOTA GUI is an emerging capability that will provide more advanced visualization facilities in time.

15.2 Standard Output

DAKOTA outputs basic information to "standard out" (i.e., the screen) for each function evaluation, consisting of an evaluation number, parameter values, execution syntax, the active set vector, and the response data set. To describe the standard output of DAKOTA, optimization of the "container" problem (see Chapter 23 for problem formulation) is used as an example. The input file for this example is shown in Figure 15.1. In this example, there is one equality constraint, and DAKOTA's finite difference algorithm is used to provide central difference numerical gradients to the NPSOL optimizer.

```
strategy,
        single_method
        graphics
        tabular_graphics_data

method,
        npsol_sqp

variables,
        continuous_design = 2
          descriptors'H' 'D'
          initial_point 4.5 4.5
          lower_bounds  0.0 0.0

interface,
        system
          analysis_driver = 'container'
          parameters_file = 'container.in'
          results_file    = 'container.out'
          file_tag

responses,
        objective_functions = 1
        nonlinear_equality_constraints = 1
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 0.001
        no_hessians
```

Figure 15.1: DAKOTA input file for the "container" example problem.

A partial listing of the DAKOTA output for the container optimization example follows:

```
Running MPI executable in serial mode.
DAKOTA version 5.0 released 12/21/2009.
Subversion revision 5635M built Dec 18 2009 17:19:56.
Constructing Single Method Strategy...
Writing new restart file dakota.rst
methodName = npsol_sqp
gradientType = numerical
Numerical gradients using central differences
to be calculated by the dakota finite difference routine.
hessianType = none

>>>>> Running Single Method Strategy.

>>>>> Running npsol_sqp iterator.




                     NPSOL  ---  Version 5.0-2      Sept 1995
                     ========================================

-----------------------------------------
Begin Dakota derivative estimation routine
-----------------------------------------

>>>>> Initial map for analytic portion of response:

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      4.5000000000e+00 H
                      4.5000000000e+00 D

container container.in.1 container.out.1

Active response data for function evaluation 1:
Active set vector = { 1 1 }
                      1.0713145108e+02 obj_fn
                      8.0444076396e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[1] + h:

------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                      4.5045000000e+00 H
                      4.5000000000e+00 D
```

```
container container.in.2 container.out.2

Active response data for function evaluation 2:
Active set vector = { 1 1 }
                     1.0719761302e+02 obj_fn
                     8.1159770472e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[1] - h:

------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                     4.4955000000e+00 H
                     4.5000000000e+00 D

container container.in.3 container.out.3

Active response data for function evaluation 3:
Active set vector = { 1 1 }
                     1.0706528914e+02 obj_fn
                     7.9728382320e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[2] + h:

------------------------------
Begin Function Evaluation    4
------------------------------
Parameters for function evaluation 4:
                     4.5000000000e+00 H
                     4.5045000000e+00 D

container container.in.4 container.out.4

Active response data for function evaluation 4:
Active set vector = { 1 1 }
                     1.0727959301e+02 obj_fn
                     8.1876180243e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[2] - h:

------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                     4.5000000000e+00 H
                     4.4955000000e+00 D

container container.in.5 container.out.5

Active response data for function evaluation 5:
```

```
Active set vector = { 1 1 }
                      1.0698339109e+02 obj_fn
                      7.9013403937e+00 nln_eq_con_1


>>>>> Total response returned to iterator:

Active set vector = { 3 3 } Deriv vars vector = { 1 2 }
                      1.0713145108e+02 obj_fn
                      8.0444076396e+00 nln_eq_con_1
 [  1.4702653619e+01  3.2911324639e+01 ] obj_fn gradient
 [  1.5904312809e+01  3.1808625618e+01 ] nln_eq_con_1 gradient




 Majr Minr    Step  Fun  Merit function Norm gZ  Violtn   nZ Penalty Conv
    0    1 0.0E+00    1  9.90366719E+01 1.6E+00 8.0E+00    1 0.0E+00 F FF

<SNIP>

>>>>> Dakota finite difference gradient evaluation for x[2] - h:

------------------------------
Begin Function Evaluation   40
------------------------------
Parameters for function evaluation 40:
                      4.9873894231e+00 H
                      4.0230575428e+00 D

container container.in.40 container.out.40

Active response data for function evaluation 40:
Active set vector = { 1 1 }
                      9.8301287596e+01 obj_fn
                     -1.2698647501e-01 nln_eq_con_1


>>>>> Total response returned to iterator:

Active set vector = { 3 3 } Deriv vars vector = { 1 2 }
                      9.8432498116e+01 obj_fn
                     -9.6918029158e-12 nln_eq_con_1
 [  1.3157517860e+01  3.2590159623e+01 ] obj_fn gradient
 [  1.2737124497e+01  3.1548877601e+01 ] nln_eq_con_1 gradient


    7    1 1.0E+00    8  9.84324981E+01 4.8E-11 9.7E-12    1 1.7E+02 T TT

 Exit NPSOL - Optimal solution found.

 Final nonlinear objective value =    98.43250

NPSOL exits with INFORM code = 0 (see "Interpretation of output" section in NPSOL manual)
```

```
NOTE: see Fortran device 9 file (fort.9 or ftn09)
      for complete NPSOL iteration history.
<<<<< Function evaluation summary: 40 total (40 new, 0 duplicate)
<<<<< Best parameters          =
                     4.9873894231e+00 H
                     4.0270846274e+00 D
<<<<< Best objective function  =
                     9.8432498116e+01
<<<<< Best constraint values   =
                    -9.6918029158e-12
<<<<< Best data captured at function evaluation 36

<<<<< Iterator npsol_sqp completed.
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =       0.09 [parent =   0.082988, child =   0.007012]
  Total wall clock =    0.34364
Exit graphics window to terminate DAKOTA.
```

The first block of lines provide a report on the DAKOTA configuration and settings. The lines that follow, down to the line "`Exit NPSOL - Optimal solution found`", contain information about the function evaluations that have been requested by NPSOL and performed by DAKOTA. Evaluations 6 through 39 have been omitted from the listing for brevity.

Following the line "`Begin Function Evaluation 1`", the initial values of the design variables, the syntax of the function evaluation, and the resulting objective and constraint function values are listed. The values of the design variables are labeled with the tags `H` and `D`, respectively, according to the descriptors to these variables given in the input file, Figure 15.1. The values of the objective function and volume constraint are labeled with the tags `obj_fn` and `nln_eq_con_1`, respectively. Note that the initial design parameters are infeasible since the equality constraint is violated ($\neq 0$). However, by the end of the run, the optimizer finds a design that is both feasible and optimal for this example. Between the design variables and response values, the content of the system call to the simulator is displayed as "`(container container.in.1 container.out.1)`", with `container` being the name of the simulator and `container.in.1` and `container.out.1` being the names of the parameters and results files, respectively.

Just preceding the output of the objective and constraint function values is the line "`Active set vector = {1 1}`". The active set vector indicates the types of data that are required from the simulator for the objective and constraint functions, and values of "`1`" indicate that the simulator must return values for these functions (gradient and Hessian data are not required). For more information on the active set vector, see Section 11.7.

Since finite difference gradients have been specified, DAKOTA computes their values by making additional function evaluation requests to the simulator at perturbed parameter values. Examples of the gradient-related function evaluations have been included in the sample output, beginning with the line that reads "`>>>>> Dakota finite difference evaluation for x[1] + h:`". The resulting finite difference gradients are listed after function evaluation 5 beginning with the line "`>>>>> Total response returned to iterator:`". Here, another active set vector is displayed in the DAKOTA output file. The line "`Active set vector = { 3 3 }`" indicates that the total response resulting from the finite differencing contains function values and gradients.

The final lines of the DAKOTA output, beginning with the line "`<<<<< Function evaluation summary:`", summarize the results of the optimization study. The best values of the optimization parameters, objective function, and volume constraint are presented along with the function evaluation number where they occurred, total

```
%eval_id              H              D         obj_fn   nln_eq_con_1
       1            4.5            4.5    107.1314511     8.04440764
       2    5.801246882    3.596476363    94.33737399    -4.59103645
       3    5.197920019    3.923577479     97.7797214  -0.6780884711
       4    4.932877133    4.044776216    98.28930566  -0.1410680284
       5    4.989328733    4.026133158     98.4270019 -0.005324671422
       6    4.987494493    4.027041977    98.43249058 -7.307058453e-06
       7    4.987391669     4.02708372    98.43249809 -2.032538049e-08
       8    4.987389423    4.027084627    98.43249812 -9.691802916e-12
```

Figure 15.2: DAKOTA's tabular output file showing the iteration history of the "container" optimization problem.

function evaluation counts, and a timing summary. In the end, the objective function has been minimized and the equality constraint has been satisfied (driven to zero within the constraint tolerance).

The DAKOTA results are intermixed with iteration information from the NPSOL library. The lines with the heading "`Majr Minr Step Fun Merit function Norm gZ Violtn nZ Penalty Conv`" come from Fortran write statements within NPSOL. The output is mixed since both DAKOTA and NPSOL are writing to the same standard output stream. The relative locations of these output contributions can vary depending on the specifics of output buffering and flushing on a particular platform and depending on whether or not the standard output is being redirected to a file. In some cases, output from the optimization library may appear on each iteration (as in this example), and in other cases, it may appear at the end of the DAKOTA output. Finally, a more detailed summary of the NPSOL iterations is written to the Fortran device 9 file (e.g., `fort.9` or `ftn09`).

15.3 Tabular Output Data

DAKOTA has the capability to print the iteration history in tabular form to a file. The keyword `tabular_graphics_data` needs to be included in the strategy specification (see Figure 15.1). The primary intent of this capability is to facilitate the transfer of DAKOTA's iteration history data to an external mathematical analysis and/or graphics plotting package (e.g., MATLAB, TECplot, Excel, S-plus, Minitab). Any evaluations from DAKOTA's internal finite differencing are suppressed, which leads to better data visualizations. This suppression of lower level data is consistent with the data that is sent to the graphics windows, as described in Section 15.4. If this data suppression is undesirable, Section 21.2.3 describes an approach where every function evaluation, even the ones from finite differencing, can be saved to a file in tabular format.

The default file name for the tabular output data is "`dakota_tabular.dat`" and the output from the "container" optimization problem is shown in Figure 15.2. This annotated tabular format (see Section 14.1.1) file contains the complete history of data requests from NPSOL (8 requests map into a total of 40 function evaluations when including the central finite differencing). The first column is the data request number, the second and third columns are the design parameter values (labeled in the example as "`H`" and "`D`"), the fourth column is the objective function (labeled "`obj_fn`"), and the fifth column is the nonlinear equality constraint (labeled "`nln_eq_con_1`").

Figure 15.3: DAKOTA 2D graphics for "container" problem showing history of an objective function, an equality constraint, and two variables.

Figure 15.4: Options for DAKOTA 2D graphics.

15.4 Graphics Output

Graphics capabilities are available for monitoring the progress of an iterative study. The graphics option is invoked by adding the `graphics` flag in the strategy specification of the DAKOTA input file (see Figure 15.1). The graphics display the values of each response function (e.g., objective and constraint functions) and each parameter for the function evaluations in the study. As for the tabular output described in Section 15.3, internal finite difference evaluations are suppressed in order to omit this clutter from the graphics. Figure 15.3 shows the optimization iteration history for the container example.

If DAKOTA is executed on a remote machine, the DISPLAY variable in the user's UNIX environment [48] may need to be set to the local machine in order to display the graphics window.

The scroll bars which are located on each graph below and to the right of each plot may be operated by dragging on the bars or pressing the arrows, both of which result in expansion/contraction of the axis scale. Clicking on the "Options" button results in the window shown in Figure 15.4, which allows the user to include min/max markers on the vertical axis, vertical and horizontal axis labels, and a plot legend within the corresponding graphics plot. In addition, the values of either or both axes may be plotted using a logarithmic scale (so long as all plot values are greater than zero) and an encapsulated postscript (EPS) file, named `dakota_graphic_i.eps` where *i* is the plot window number, can be created using the "Print" button.

15.5 Error Messages Output

A variety of error messages are printed by DAKOTA in the event that an error is detected in the input specification. Some of the more common input errors, and the associated error messages, are described below. See also the Common Specification Mistakes section in the DAKOTA Reference Manual [3].

Incorrectly spelled specifications, such as ``numericl_gradients'', will result in error messages of the form:

```
Parser detected syntax error: unrecognized identifier 'numericl_gradients'
  within responses keyword.
Please refer to the dakota.input.txt distributed with this executable.
```

The input parser catches syntax errors, but not logic errors. The fact that certain input combinations are erroneous must be detected after parsing, at object construction time. For example, if a `no_gradients` specification for a response data set is combined with selection of a gradient-based optimization method, then this error must be detected during set-up of the optimizer (see last line of listing):

```
Running MPI executable in serial mode.
DAKOTA version 4.0 released 05/12/2006.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
methodName = npsol_sqp
gradientType = none
hessianType = none

Error: gradient-based optimizers require a gradient specification.
```

Another common mistake involves a mismatch between the amount of data expected on a function evaluation and the data returned by the user's simulation code or driver. The available response data is specified in the responses keyword block, and the subset of this data needed for a particular evaluation is managed by the active set vector. For example, if DAKOTA expects function values and gradients to be returned (as indicated by an active set vector containing 3's), but the user's simulation code only returns function values, then the following error message is generated:

```
At EOF: insufficient data for functionGradient 1
```

Unfortunately, descriptive error messages are not available for all possible failure modes of DAKOTA. If you encounter core dumps, segmentation faults, or other failures, please request help using the support mechanisms described on the DAKOTA website.

15.6 Variables Output from Pre-run

The pre-run mode (supported only for select methods) permits specification of an output file to which DAKOTA will write parameter (variables) data in annotated format (see Section 14.1.1) with data columns corresponding to each variable. This file can be generated with sampling, parameter study, and DACE methods by invoking

```
dakota -i dakota.in -pre_run ::variables.dat
```

for example, to output the variables (samples) in an LHS study.

Chapter 16

Advanced Strategies

16.1 Overview

DAKOTA's strategy capabilities were developed in order to provide a control layer for managing multiple iterators and models. It was driven by the observed need for "meta-optimization" and other high level systems analysis procedures in real-world engineering design problems. This capability allows the use of existing iterative algorithm and computational model software components as building blocks to accomplish more sophisticated studies, such as hybrid minimization, multistart local minimization, or Pareto optimization. Other strategy-like capabilities are enabled by the model recursion capabilities described in Chapter 10. When these model recursion specifications are sufficient to completely describe a multi-iterator, multi-model solution approach, then a separate strategy specification is not used (see Chapter 17 for examples). In addition, some previous strategy capabilities (i.e., the surrogate-based minimization approaches descibed in Chapter 9) have migrated into the method specification to allow their componentization and reuse elsewhere. This trend will continue in future releases so that only the most generic coordination approaches will remain.

16.2 Hybrid Minimization

In the hybrid minimization strategy (keyword: `hybrid`), a sequence of minimization methods are applied to find an optimal design point. The goal of this strategy is to exploit the strengths of different minimization algorithms through different stages of the minimization process. Global/local optimization hybrids (e.g., genetic algorithms combined with nonlinear programming) are a common example in which the desire for a global optimum is balanced with the need for efficient navigation to a local optimum. An important related feature is that the sequence of minimization algorithms can employ models of varying fidelity. In the global/local case, for example, it would often be advantageous to use a low-fidelity model in the global search phase, followed by use of a more refined model in the local search phase.

The specification for hybrid minimization involves a list of method identifier strings, and each of the corresponding method specifications has the responsibility for identifying the model specification (which may in turn identify variables, interface, and responses specifications) that each method will use (see the DAKOTA Reference Manual [3] and the example discussed below). Currently, only the sequential hybrid approach is available. The `embedded` and `collaborative` approaches are not fully functional at this time.

In the `sequential` hybrid minimization approach, a sequence of minimization methods is invoked in the order

specified in the DAKOTA input file. After each method completes execution, the best solution or solutions from that method are used as the starting point(s) for the following method. The number of solutions transferred is defined by how many that method can generate and how many the user specifies with the individual method keyword `final_solutions`. For example, currently only a few of the global optimization methods such as genetic algorithms (e.g. `moga` and `coliny_ea`) and sampling methods return multiple solutions. In this case, the specified number of solutions from the previous method will be used to initialize the subsequent method. If the subsequent method cannot accept multiple input points (currently only a few methods such as the genetic algorithms in JEGA allow multiple input points), then multiple instances of the subsequent method are generated, each one initialized by one of the optimal solutions from the previous method. For example, if LHS sampling were run as the first method and the number of final solutions was 10 and the DOT conjugate gradient was the second method, there would be 10 instances of `dot_frcg` started, each with a separate LHS sample solution as its initial point. Method switching is governed by the separate convergence controls of each method; that is, *each method is allowed to run to its own internal definition of completion without interference*. Individual method completion may be determined by convergence criteria (e.g., `convergence_tolerance`) or iteration limits (e.g., `max_iterations`).

Figure 16.1 shows a DAKOTA input file that specifies a sequential hybrid optimization strategy to solve the "textbook" optimization test problem. This input file is named `dakota_hybrid.in` in the `Dakota/test` directory. The three optimization methods are identified using the `method_list` specification in the strategy section of the input file. The identifier strings listed in the specification are '`GA`' for genetic algorithm, '`PS`' for pattern search, and '`NLP`' for nonlinear programming. Following the strategy keyword block are the three corresponding method keyword blocks. Note that each method has a tag following the `id_method` keyword that corresponds to one of the method names listed in the strategy keyword block. By following the identifier tags from `method` to `model` and from `model` to `variables`, `interface`, and `responses`, it is easy to see the specification linkages for this problem. The GA optimizer runs first and uses model '`M1`' which includes variables '`V1`', interface '`I1`', and responses '`R1`'. Note that in the specification, `final_solutions=1`, so only one (the best) solution is returned from the GA. However, it is possible to change this to `final_solutions=5` and get five solutions passed from the GA to the Pattern Search (for example). Once the GA is complete, the PS optimizer starts from the best GA result and again uses model '`M1`'. Since both GA and PS are nongradient-based optimization methods, there is no need for gradient or Hessian information in the '`R1`' response keyword block. The NLP optimizer runs last, using the best result from the PS method as its starting point. It uses model '`M2`' which includes the same '`V1`' and '`I1`' keyword blocks, but uses the responses keyword block '`R2`' since the full Newton optimizer used in this example (`optpp_newton`) needs analytic gradient and Hessian data to perform its search.

16.3 Multistart Local Minimization

A simple, heuristic, global minimization technique is to use many local minimization runs, each of which is started from a different initial point in the parameter space. This is known as multistart local minimization. This is an attractive strategy in situations where multiple local optima are known or expected to exist in the parameter space. However, there is no theoretical guarantee that the global optimum will be found. This approach combines the efficiency of local minimization methods with a user-specified global stratification (using a specified `starting_points` list, a number of specified `random_starts`, or both; see the DAKOTA Reference Manual [3] for additional specification details). Since solutions for different starting points are independent, parallel computing may be used to concurrently run the local minimizations.

An example input file for multistart local optimization on the "quasi_sine" test function (see `quasi_sine_fcn.C` in `Dakota/test`) is shown in Figure 16.2. The strategy keyword block in the input file contains the keyword `multi_start`, along with the set of starting points (3 random and 5 listed) that will be used for the optimization

```
strategy,
        graphics
        hybrid sequential
          method_list = 'GA' 'PS' 'NLP'

method,
        id_method = 'GA'
        model_pointer = 'M1'
        coliny_ea
          final_solutions = 1
          seed = 1234
          population_size = 5
          verbose output

method,
        id_method = 'PS'
        model_pointer = 'M1'
        coliny_pattern_search stochastic
          seed = 1234
          initial_delta = 0.1
          threshold_delta = 1.e-4
          solution_accuracy = 1.e-10
          exploratory_moves basic_pattern
          verbose output

method,
        id_method = 'PS2'
        model_pointer = 'M1'
        max_function_evaluations = 10
        coliny_pattern_search stochastic
          seed = 1234
          initial_delta = 0.1
          threshold_delta = 1.e-4
          solution_accuracy = 1.e-10
          exploratory_moves basic_pattern
          verbose output

method,
        id_method = 'NLP'
        model_pointer = 'M2'
        optpp_newton
          gradient_tolerance = 1.e-12
          convergence_tolerance = 1.e-15
          verbose output

model,
        id_model = 'M1'
        single
          variables_pointer = 'V1'
          interface_pointer = 'I1'
          responses_pointer = 'R1'

model,
        id_model = 'M2'
        single
          variables_pointer = 'V1'
          interface_pointer = 'I1'
          responses_pointer = 'R2'

variables,
        id_variables = 'V1'
        continuous_design = 2
          initial_point    0.6    0.7
          upper_bounds     5.8    2.9
          lower_bounds     0.5   -2.9
          descriptors      'x1'   'x2'

interface,
        id_interface = 'I1'
        direct
          analysis_driver=  'text_book'

responses,
        id_responses = 'R1'
        objective_functions = 1
        no_gradients
        no_hessians

responses,
        id_responses = 'R2'
        objective_functions = 1
        analytic_gradients
        analytic_hessians
```

Figure 16.1: DAKOTA input file for a hybrid optimization strategy.

```
strategy,
        multi_start graphics
          method_pointer = 'NLP'
          random_starts = 3 seed = 123
          starting_points = -.8  -.8
                            -.8   .8
                             .8  -.8
                             .8   .8
                              0.   0.

method,
        id_method = 'NLP'
        dot_bfgs

variables,
        continuous_design = 2
          lower_bounds    -1.0     -1.0
          upper_bounds     1.0  1.0
          descriptors      'x1'     'x2'

interface,
        system #asynchronous
          analysis_driver = 'quasi_sine_fcn'

responses,
        objective_functions = 1
        analytic_gradients
        no_hessians
```

Figure 16.2: DAKOTA input file for a multistart local optimization strategy.

runs. The other keyword blocks in the input file are similar to what would be used in a single optimization run.

The `quasi_sine` test function has multiple local minima, but there is an overall trend in the function that tends toward the global minimum at $(x1, x2) = (0.177, 0.177)$. See [50] for more information on this test function. Figure 16.3 shows the results summary for the eight local optimizations performed. From the five specified starting points and the 3 random starting points (as identified by the `x1`, `x2` headers), the eight local optima (as identified by the `x1*`, `x2*` headers) are all different and only one of the local optimizations finds the global minimum.

16.4 Pareto Optimization

The Pareto optimization strategy (keyword: `pareto_set`) is one of three multiobjective optimization capabilities discussed in Section 7.3.1. In the Pareto optimization strategy, multiple sets of multiobjective weightings are evaluated. The user can specify these weighting sets in the strategy keyword block using a `multi_objective_weight_sets` list, a number of `random_weight_sets`, or both (see the DAKOTA Reference Manual [3] for additional specification details). Figure 16.4 shows the input commands from the file `dakota_pareto.in` in the `Dakota/test`

```
<<<<< Results summary:
   set_id             x1             x2            x1*            x2*         obj_fn
        1           -0.8           -0.8  -0.8543728666  -0.8543728666   0.5584096919
        2           -0.8            0.8  -0.9998398719    0.177092822    0.291406596
        3            0.8           -0.8    0.177092822  -0.9998398719    0.291406596
        4            0.8            0.8   0.1770928217   0.1770928217   0.0602471946
        5              0              0  0.03572926375  0.03572926375  0.08730499239
        6  -0.7767971993  0.01810943539  -0.7024118387  0.03572951143   0.3165522387
        7  -0.3291571008  -0.7697378755   0.3167607374  -0.4009188363   0.2471403213
        8   0.8704730469   0.7720679005    0.177092899   0.3167611757  0.08256082751
```

Figure 16.3: DAKOTA results summary for a multistart local optimization strategy.

directory.

DAKOTA performs one multiobjective optimization problem for each set of multiobjective weights. The collection of computed optimal solutions form a Pareto set, which can be useful in making trade-off decisions in engineering design. Since solutions for different multiobjective weights are independent, parallel computing may be used to concurrently execute the multiobjective optimization problems.

Figure 16.5 shows the results summary for the Pareto-set optimization strategy. For the four multiobjective weighting sets (as identified by the `w1`, `w2`, `w3` headers), the local optima (as identified by the `x1`, `x2` headers) are all different and correspond to individual objective function values of (f_1, f_2, f_3) = (0.0,0.5,0.5), (13.1,-1.2,8.16), (532.,33.6,-2.9), and (0.125,0.0,0.0) (note: the composite objective function is tabulated under the `obj_fn` header). The first three solutions reflect exclusive optimization of each of the individual objective functions in turn, whereas the final solution reflects a balanced weighting and the lowest sum of the three objectives. Plotting these (f_1, f_2, f_3) triplets on a 3-dimensional plot results in a Pareto surface (not shown), which is useful for visualizing the trade-offs in the competing objectives.

```
strategy,
        pareto_set
          graphics
          opt_method_pointer = 'NLP'
          multi_objective_weight_sets =
                                  1.   0.   0.
                                  0.   1.   0.
                                  0.   0.   1.
                                .333 .333 .333


method,
        id_method = 'NLP'
        dot_bfgs



model,
        single

variables,
        continuous_design = 2
          initial_point     0.9    1.1
          upper_bounds      5.8    2.9
          lower_bounds      0.5   -2.9
          descriptors       'x1'   'x2'

interface,
        system #asynchronous
          analysis_driver = 'text_book'

responses,
        objective_functions = 3
        analytic_gradients
        no_hessians
```

Figure 16.4: DAKOTA input file for the Pareto optimization strategy.

```
<<<<< Results summary:
   set_id            w1            w2            w3            x1            x2        obj_fn
        1             1             0             0   0.9996554048   0.997046351 7.612301561e-11
        2             0             1             0           0.5           2.9          -1.2
        3             0             0             1           5.8 1.12747589e-11          -2.9
        4         0.333         0.333         0.333           0.5   0.5000000041      0.041625
```

Figure 16.5: DAKOTA results summary for the Pareto-set optimization strategy.

Chapter 17

Advanced Model Recursions

The surrogate and nested model constructs admit a wide variety of multi-iterator, multi-model solution approaches. For example, optimization within optimization (for hierarchical multidisciplinary optimization), uncertainty quantification within uncertainty quantification (for interval-valued or second-order probability), uncertainty quantification within optimization (for optimization under uncertainty), and optimization within uncertainty quantification (for uncertainty of optima) are all supported, with and without surrogate model indirection. Three important examples are highlighted: second-order probability, optimization under uncertainty, and surrogate-based uncertainty quantification.

17.1 Mixed Aleatory-Epistemic UQ

Mixed UQ approaches employ nested models to embed one uncertainty quantification (UQ) within another. The outer level UQ is commonly linked to epistemic uncertainties (also known as reducible uncertainties) resulting from a lack of knowledge, and the inner UQ is commonly linked to aleatory uncertainties (also known as irreducible uncertainties) that are inherent in nature. The outer level generates sets of realizations of the epistemic parameters, and each set of these epistemic parameters in used within a separate inner loop probabilistic analysis over the aleatory random variables. In this manner, ensembles of aleatory statistics are generated, one set for each realization of the epistemic parameters.

In DAKOTA, we support interval-valued probability (IVP), second-order probability (SOP), and Dempster-Shafer theory of evidence (DSTE) approaches to mixed uncertainty. These three approaches differ by how they treat the epistemic variables in the outer loop: they are treated as intervals in IVP, as subjective probability distributions in SOP, and as belief structures in DSTE. This set of techniques provides a spectrum of assumed epistemic structure, from strongest assumptions in SOP to weakest in IVP.

17.1.1 Interval-valued probability (IVP)

In IVP (also known as probability bounds analysis [36, 71, 7]), we employ an outer loop of interval estimation in combination with an aleatory inner loop. In interval analysis, it is assumed that nothing is known about the uncertain input variables except that they lie within certain intervals. The problem of uncertainty propagation then becomes an interval analysis problem: given inputs that are defined within intervals, what is the corresponding interval on the outputs?

Starting from a specification of intervals and probability distributions on the inputs, the intervals may augment the probability distributions, insert into the probability distributions, or some combination. We generate an ensemble of cumulative distribution functions (CDF) or Complementary Cumulative Distribution Functions (CCDF), one CDF/CCDF result for each aleatory analysis. Plotting an entire ensemble of CDFs or CCDFs in a "horsetail" plot allows one to visualize the upper and lower bounds on the family of distributions (see Figure 17.1). Given

Figure 17.1: Example CDF ensemble. Commonly referred to as a "horsetail" plot.

that the ensemble stems from multiple realizations of the epistemic uncertainties, the interpretation is that each CDF/CCDF instance has no relative probability of occurrence, only that each instance is possible. For prescribed response levels on the CDF/CCDF, an interval on the probability is computed based on the bounds of the ensemble at that level, and vice versa for prescribed probability levels. This interval on a statistic is interpreted simply as a possible range, where the statistic could take any of the possible values in the range.

A sample input file is shown in Figure 17.2, in which the outer epistemic level variables are defined as intervals. This file is `dakota_uq_cantilever_sop_rel.in` in `Dakota/examples/methods`. Samples will be generated from these intervals to select means for X and Y that are employed in an inner level reliability analysis of the cantilever problem (see Section 23.10). Figure 17.3 shows excerpts from the resulting output. In this particular example, the outer loop generates 50 possible realizations of epistemic variables, which are then sent to the inner loop to calculate statistics such as the mean weight, and cumulative distribution function for the stress and displacement reliability indices. Thus, the outer loop has 50 possible values for the mean weight but there is no distribution structure on these 50 samples. So, only the minimum and maximum value are reported. Similarly, the minimum and maximum values of the CCDF for the stress and displacement reliability indices are reported.

When performing an epistemic analysis, response levels and probability levels should only be defined in the inner loop. For example, if one wants to generate an interval around possible CDFs or CCDFS, we suggest defining a number of probability levels in the inner loop (0.1, 0.2, 0.3, etc). For each epistemic instance, these will be calculated during the inner loop and reported back to the outer loop. In this way, there will be an ensemble of CDF percentiles (for example) and one will have interval bounds for each of these percentile levels defined. Finally, although the epistemic variables are often values defining distribution parameters for the inner loop, they are not required to be: they can just be separate uncertain variables in the problem.

As compared to aleatory quantities of interest (e.g., mean, variance, probability) that must be integrated over a full probability domain, we observe that the desired minima and maxima of the output ranges are local point solutions

```
strategy,
        single_method
          method_pointer = 'EPISTEMIC'

method,
        id_method = 'EPISTEMIC'
        model_pointer = 'EPIST_M'
        sampling
          samples = 50 seed = 12347

model,
        id_model = 'EPIST_M'
        nested
          variables_pointer  = 'EPIST_V'
          sub_method_pointer = 'ALEATORY'
          responses_pointer  = 'EPIST_R'
          primary_variable_mapping   = 'X'    'Y'
          secondary_variable_mapping = 'mean' 'mean'
          primary_response_mapping   = 1. 0. 0. 0. 0. 0. 0. 0.
                                       0. 0. 0. 0. 1. 0. 0. 0.
                                       0. 0. 0. 0. 0. 0. 0. 1.

variables,
        id_variables = 'EPIST_V'
        interval_uncertain =2
          num_intervals = 1 1
          interval_probs = 1.0 1.0
          interval_bounds = 400. 600. 800. 1200.
          descriptors       'X_mean' 'Y_mean'

responses,
        id_responses = 'EPIST_R'
        response_functions = 3
        response_descriptors = 'mean_wt' 'ccdf_beta_s' 'ccdf_beta_d'
        no_gradients
        no_hessians

method,
        id_method = 'ALEATORY'
        model_pointer = 'ALEAT_M'
        local_reliability
          mpp_search no_approx
          num_response_levels = 0 1 1
          response_levels = 0.0 0.0
          compute reliabilities
          complementary distribution

model,
        id_model = 'ALEAT_M'
        single
          variables_pointer = 'ALEAT_V'
          interface_pointer = 'ALEAT_I'
          responses_pointer = 'ALEAT_R'

variables,
        id_variables = 'ALEAT_V'
        continuous_design = 2
          initial_point    2.4522 3.8826
          descriptors 'w' 't'
        normal_uncertain = 4
          means             =  40000. 29.E+6 500. 1000.
          std_deviations    =  2000. 1.45E+6 100. 100.
          descriptors       =  'R' 'E' 'X' 'Y'

interface,
        id_interface = 'ALEAT_I'
        direct
          analysis_driver = 'cantilever'
          deactivate evaluation_cache restart_file

responses,
        id_responses = 'ALEAT_R'
        response_functions = 3
        response_descriptors = 'weight' 'stress' 'displ'
        analytic_gradients
        no_hessians
```

Figure 17.2: DAKOTA input file for the interval-valued probability example.

```
Statistics based on 50 samples:

Min and Max values for each response function:
mean_wt:  Min = 9.5209117200e+00  Max = 9.5209117200e+00
ccdf_beta_s:  Min = 1.7627715524e+00  Max = 4.2949468386e+00
ccdf_beta_d:  Min = 2.0125192955e+00  Max = 3.9385559339e+00
```

Figure 17.3: Interval-valued statistics for cantilever beam reliability indices.

in the epistemic parameter space, such that we may employ directed optimization techniques to compute these extrema and potentially avoid the cost of sampling the full epistemic space.

In `dakota/test`, test input files such as `dakota_uq_cantilever_ivp_exp.in` and `dakota_uq_short_column_ivp_exp.in` replace the outer loop sampling with the local and global interval optimization methods described in Section 6.5.1. In these cases, we no longer generate horse tails and infer intervals, but rather compute the desired intervals directly.

17.1.2 Second-order probability (SOP)

SOP is similar to IVP in its segregation of aleatory and epistemic uncertainties and its use of nested iteration. However, rather than modeling epistemic uncertainty with a single interval per variable and computing interval-valued statistics, we instead employ subjective probability distributions and compute epistemic statistics on the aleatory statistics (for example, probabilities on probabilities – the source of the "second-order" terminology [55]). Now the different hairs of the horsetail shown in Figure 17.1 have a relative probability of occurrence and stronger inferences may be drawn. In particular, mean, 5th percentile, and 95th percentile probability values are a common example. Second-order probability is sometimes referred to as probability of frequency (PoF) analysis, referring to a probabilistic interpretation of the epistemic variables and a frequency interpretation of the aleatory variables. The PoF terminology is used in a recent National Academy of Sciences report on the Quantification of Margins and Uncertainties (QMU) [84].

Rather than employing interval estimation techniques at the outer loop in SOP, we can instead rely on the same probabilistic methods for aleatory propagation employed for the inner loop. In certain special cases, expected values of expected values can be computed in closed form (selected statistics for combined variable stochastic expansions, refer to the DAKOTA Theory Manual [2]), without need for a separate outer loop method.

The previous example in Figure 17.2 can be modified to define the epistemic outer loop using uniform variables instead of interval variables (annotated test `#1` in `dakota/test/dakota_uq_cantilever_sop_rel.in`). The process of generating the epistemic values is essentially the same in both cases; however, the interpretation of results is quite different. In IVP, each "hair" or individual CDF in the horsetail plot in Figure 17.1 would be interpreted as a possible realization of aleatory uncertainty conditional on a particular epistemic sample realization. The ensemble then indicates the influence of the epistemic variables (e.g. by how widespread the ensemble is). However, if the outer loop variables are defined to be uniformly distributed in SOP, then the outer loop results will be reported as statistics (such as mean and standard deviation) and not merely intervals. It is important to emphasize that these outer level output statistics are only meaningful to the extent that the outer level input probability specifications are meaningful (i.e., to the extent that the epistemic variables are known to be uniform).

In `dakota/test`, additional test input files such as `dakota_uq_cantilever_sop_exp.in` and `dakota_uq_short_column_sop_exp.in` explore other outer/inner loop probabilistic analysis combinations, particulary using stochastic expansion methods .

17.1.3 Dempster-Shafer Theory of Evidence

In IVP, we estimate a single epistemic output interval for each aleatory statistic. This same nested analysis procedure may be employed within the cell computations of a DSTE approach. Instead of a single interval, we now compute multiple output intervals, one for each combination of the input basic probability assignments, in order to define epistemic belief and plausibility functions on the aleatory statistics computed in the inner loop. While this can significantly increase the computational requirements, belief and plausibility functions provide a more finely resolved epistemic characterization than a basic output interval.

The single-level DSTE approach for propagating epistemic uncertainties is described in Section 6.5.2 and in the DAKOTA Theory Manual [2]. Examples of nested DSTE for propagating mixed uncertainties can be seen in `dakota/test` in the input file `dakota_uq_ishigami_dste_exp.in`.

17.2 Optimization Under Uncertainty (OUU)

Optimization under uncertainty (OUU) approaches incorporate an uncertainty quantification method within the optimization process. This is often needed in engineering design problems when one must include the effect of input parameter uncertainties on the response functions of interest. A typical engineering example of OUU would minimize the probability of failure of a structure for a set of applied loads, where there is uncertainty in the loads and/or material properties of the structural components.

In OUU, a nondeterministic method is used to evaluate the effect of uncertain variable distributions on response functions of interest (refer to Chapter 6 for additional information on nondeterministic analysis). Statistics on these response functions are then included in the objective and constraint functions of an optimization process. Different UQ methods can have very different features from an optimization perspective, leading to the tailoring of optimization under uncertainty approaches to particular underlying UQ methodologies.

If the UQ method is sampling based, then three approaches are currently supported: nested OUU, surrogate-based OUU, and trust-region surrogate-based OUU. Additional details and computational results are provided in [27].

Another class of OUU algorithms is called reliability-based design optimization (RBDO). RBDO methods are used to perform design optimization accounting for reliability metrics. The reliability analysis capabilities described in Section 6.3 provide a rich foundation for exploring a variety of RBDO formulations. [24] investigated bi-level, fully-analytic bi-level, and first-order sequential RBDO approaches employing underlying first-order reliability assessments. [25] investigated fully-analytic bi-level and second-order sequential RBDO approaches employing underlying second-order reliability assessments.

When using stochastic expansions for UQ, analytic moments and analytic design sensitivities can be exploited as described in [34]. Several approaches for obtaining design sensitivities of statistical metrics are discussed in Section 17.2.5.

Finally, when employing epistemic methods for UQ, the set of statistics available for use within optimization are interval-based. Robustness metrics typically involve the width of the intervals, and reliability metrics typically involve the worst case upper or lower bound of the interval.

Each of these OUU methods is overviewed in the following sections.

17.2.1 Nested OUU

In the case of a nested approach, the optimization loop is the outer loop which seeks to optimize a nondeterministic quantity (e.g., minimize probability of failure). The uncertainty quantification (UQ) inner loop evaluates this

Figure 17.4: Formulation 1: Nested OUU.

nondeterministic quantity (e.g., computes the probability of failure) for each optimization function evaluation. Figure 17.4 depicts the nested OUU iteration where $\mathbf{d}$ are the design variables, $\mathbf{u}$ are the uncertain variables characterized by probability distributions, $\mathbf{r_u}(\mathbf{d}, \mathbf{u})$ are the response functions from the simulation, and $\mathbf{s_u}(\mathbf{d})$ are the statistics generated from the uncertainty quantification on these response functions.

Figure 17.5 shows a DAKOTA input file for a nested OUU example problem that is based on the textbook test problem. This input file is named `dakota_ouu1_tb.in` in the `dakota/test` directory. In this example, the objective function contains two probability of failure estimates, and an inequality constraint contains another probability of failure estimate. For this example, failure is defined to occur when one of the textbook response functions exceeds its threshold value. The strategy keyword block at the top of the input file identifies this as an OUU problem. The strategy keyword block is followed by the optimization specification, consisting of the optimization method, the continuous design variables, and the response quantities that will be used by the optimizer. The mapping matrices used for incorporating UQ statistics into the optimization response data are described in the DAKOTA Reference Manual [3]. The uncertainty quantification specification includes the UQ method, the uncertain variable probability distributions, the interface to the simulation code, and the UQ response attributes. As with other complex DAKOTA input files, the identification tags given in each keyword block can be used to follow the relationships among the different keyword blocks.

Latin hypercube sampling is used as the UQ method in this example problem. Thus, each evaluation of the response functions by the optimizer entails 50 Latin hypercube samples. In general, nested OUU studies can easily generate several thousand function evaluations and gradient-based optimizers may not perform well due to noisy or insensitive statistics resulting from under-resolved sampling. These observations motivate the use of surrogate-based approaches to OUU.

Other nested OUU examples in the `dakota/test` directory include `dakota_ouu1_tbch.in`, which adds an additional interface for including deterministic data in the textbook OUU problem, and `dakota_ouu1_cantilever.in`, which solves the cantilever OUU problem (see Section 23.10) with a nested approach. For each of these files, the "`1`" identifies formulation 1, which is short-hand for the nested approach.

17.2.2 Surrogate-Based OUU (SBOUU)

Surrogate-based optimization under uncertainty strategies can be effective in reducing the expense of OUU studies. Possible formulations include use of a surrogate model at the optimization level, at the uncertainty quantification level, or at both levels. These surrogate models encompass both data fit surrogates (at the optimization or UQ level) and model hierarchy surrogates (at the UQ level only). Figure 17.6 depicts the different surrogate-based formulations where $\hat{\mathbf{r}}_\mathbf{u}$ and $\hat{\mathbf{s}}_\mathbf{u}$ are approximate response functions and approximate response statistics, respectively, generated from the surrogate models.

```
strategy,
        single_method
          method_pointer = 'OPTIM'

method,
        id_method = 'OPTIM'
        model_pointer = 'OPTIM_M'
        npsol_sqp
          convergence_tolerance = 1.e-8

model,
        id_model = 'OPTIM_M'
        nested
          variables_pointer  = 'OPTIM_V'
          sub_method_pointer = 'UQ'
          responses_pointer  = 'OPTIM_R'
          primary_response_mapping   = 0. 0. 1. 0. 0. 1. 0. 0. 0.
          secondary_response_mapping = 0. 0. 0. 0. 0. 0. 0. 0. 1.

variables,
        id_variables = 'OPTIM_V'
        continuous_design = 2
          initial_point    1.8    1.0
          upper_bounds     2.164  4.0
          lower_bounds     1.5    0.0
          descriptors      'd1'   'd2'

responses,
        id_responses = 'OPTIM_R'
        objective_functions = 1
        nonlinear_inequality_constraints = 1
        nonlinear_inequality_upper_bounds = .1
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-1
        no_hessians

method,
        id_method = 'UQ'
        model_pointer = 'UQ_M'
        sampling,
          samples = 50 seed = 1 sample_type lhs
          response_levels = 3.6e+11 1.2e+05 3.5e+05
          complementary distribution

model,
        id_model = 'UQ_M'
        single
          variables_pointer = 'UQ_V'
          interface_pointer = 'UQ_I'
          responses_pointer = 'UQ_R'

variables,
        id_variables = 'UQ_V'
        continuous_design = 2
        normal_uncertain = 2
          means          =  248.89, 593.33
          std_deviations =   12.4,  29.7
          descriptors    =  'nuv1'  'nuv2'
        uniform_uncertain = 2
          lower_bounds =  199.3,  474.63
          upper_bounds =  298.5,  712.
          descriptors  =  'uuv1'  'uuv2'
        weibull_uncertain = 2
          alphas       =    12., 30.
          betas        =   250., 590.
          descriptors  =  'wuv1'  'wuv2'

interface,
        id_interface = 'UQ_I'
        system asynch evaluation_concurrency = 5
          analysis_driver = 'text_book_ouu'

responses,
        id_responses = 'UQ_R'
        response_functions = 3
        no_gradients
        no_hessians
```

Figure 17.5: DAKOTA input file for the nested OUU example.

Figure 17.6: Formulations 2, 3, and 4 for Surrogate-based OUU.

SBOUU examples in the `dakota/test` directory include `dakota_sbouu2_tbch.in`, `dakota_sbouu3_tbch.in`, and `dakota_sbouu4_tbch.in`, which solve the textbook OUU problem, and `dakota_sbouu2_cantilever.in`, `dakota_sbouu3_cantilever.in`, and `dakota_sbouu4_cantilever.in`, which solve the cantilever OUU problem (see Section 23.10). For each of these files, the "2," "3," and "4" identify formulations 2, 3, and 4, which are short-hand for the "layered containing nested," "nested containing layered," and "layered containing nested containing layered" surrogate-based formulations, respectively. In general, the use of surrogates greatly reduces the computational expense of these OUU study. However, without restricting and verifying the steps in the approximate optimization cycles, weaknesses in the data fits can be exploited and poor solutions may be obtained. The need to maintain accuracy of results leads to the use of trust-region surrogate-based approaches.

17.2.3 Trust-Region Surrogate-Based OUU (TR-SBOUU)

The TR-SBOUU approach applies the trust region logic of deterministic SBO (see Section 9.2) to SBOUU. Trust-region verifications are applicable when surrogates are used at the optimization level, i.e., formulations 2 and 4. As a result of periodic verifications and surrogate rebuilds, these techniques are more expensive than SBOUU; however they are more reliable in that they maintain the accuracy of results. Relative to nested OUU (formulation 1), TR-SBOUU tends to be less expensive and less sensitive to initial seed and starting point.

TR-SBOUU examples in the `dakota/test` directory include `dakota_trsbouu2_tbch.in` and `dakota_trsbouu4_tbch.in`, which solve the textbook OUU problem, and `dakota_trsbouu2_cantilever.in` and `dakota_trsbouu4_cantilever.in`, which solve the cantilever OUU problem (see Section 23.10).

Computational results for several example problems are available in [27].

17.2.4 RBDO

Bi-level and sequential approaches to reliability-based design optimization (RBDO) and their associated sensitivity analysis requirements are described in the Optimization Under Uncertainty chapter of the DAKOTA Theory Manual [2].

A number of bi-level RBDO examples are provided in `dakota/test`. The `dakota_rbdo_cantilever.in`, `dakota_rbdo_short_column.in`, and `dakota_rbdo_steel_column.in` input files solve the cantilever (see Section 23.10), short column (see Section 23.9), and steel column (see Section 23.11) OUU problems using a bi-level RBDO approach employing numerical design gradients. The `dakota_rbdo_cantilever_analytic.in` and `dakota_rbdo_short_column_analytic.in` input files solve the cantilever and short column OUU problems using a bi-level RBDO approach with analytic design gradients and first-order limit state approximations. The `dakota_rbdo_cantilever_analytic2.in`, `dakota_rbdo_short_column_analytic2.in`, and `dakota_rbdo_steel_column_analytic2.in` input files also employ analytic design gradients, but are extended to employ second-order limit state approximations and integrations.

Sequential RBDO examples are also provided in `dakota/test`. The `dakota_rbdo_cantilever_trsb.in` and `dakota_rbdo_short_column_trsb.in` input files solve the cantilever and short column OUU problems using a first-order sequential RBDO approach with analytic design gradients and first-order limit state approximations. The `dakota_rbdo_cantilever_trsb2.in`, `dakota_rbdo_short_column_trsb2.in`, and `dakota_rbdo_steel_column_trsb2.in` input files utilize second-order sequential RBDO approaches that employ second-order limit state approximations and integrations (from analytic limit state Hessians with respect to the uncertain variables) and quasi-Newton approximations to the reliability metric Hessians with respect to design variables.

17.2.5 Stochastic Expansion-Based Design Optimization

For stochastic expansion-based approaches to optimization under uncertainty, bi-level, sequential, and multifidelity approaches and their associated sensitivity analysis requirements are described in the Optimization Under Uncertainty chapter of the DAKOTA Theory Manual [2].

In `dakota/test`, the `dakota_pcbdo_cantilever.in`, `dakota_pcbdo_rosenbrock.in`, `dakota_pcbdo_short_column.in`, and `dakota_pcbdo_steel_column.in` input files solve cantilever (see Section 23.10), Rosenbrock, short column (see Section 23.9), and steel column (see Section 23.11) OUU problems using a bi-level polynomial chaos-based approach, where the statistical design metrics are reliability indices based on moment projection (see Mean Value section in Reliability Methods Chapter of DAKOTA Theory Manual [2]). The test matrix in the former three input files evaluate design gradients of these reliability indices using several different approaches: analytic design gradients based on a PCE formed over only over the random variables, analytic design gradients based on a PCE formed over all variables, numerical design gradients based on a PCE formed only over the random variables, and numerical design gradients based on a PCE formed over all variables. In the cases where the expansion is formed over all variables, only a single PCE construction is required for the complete PCBDO process, whereas the expansions only over the random variables must be recomputed for each change in design variables. Sensitivities for "augmented" design variables (which are separate from and augment the random variables) may be handled using either analytic approach; however, sensitivities for "inserted" design variables (which define distribution parameters for the random variables) must be computed using $\frac{dR}{dx}\frac{dx}{ds}$ (refer to Stochastic Sensitivity Analysis section in Optimization Under Uncertainty chapter of DAKOTA Theory Manual [2]). Additional test input files include:

- `dakota_scbdo_cantilever.in`, `dakota_scbdo_rosenbrock.in`, `dakota_scbdo_short_column.in`, and `dakota_scbdo_steel_column.in` input files solve cantilever, Rosenbrock, short column, and steel column OUU problems using a bi-level stochastic collocation-based approach.
- `dakota_pcbdo_cantilever_trsb.in`, `dakota_pcbdo_rosenbrock_trsb.in`, `dakota_pcbdo_short_column_trsb.in`, `dakota_pcbdo_steel_column_trsb.in`, `dakota_scbdo_cantilever_trsb.in`, `dakota_scbdo_rosenbrock_trsb.in`,

dakota_scbdo_short_column_trsb.in, and dakota_scbdo_steel_column_trsb.in input files solve cantilever, Rosenbrock, short column, and steel column OUU problems using sequential polynomial chaos-based and stochastic collocation-based approaches.

- dakota_pcbdo_cantilever_mf.in, dakota_pcbdo_rosenbrock_mf.in, dakota_pcbdo_short_column_mf.in, dakota_scbdo_cantilever_mf.in, dakota_scbdo_rosenbrock_mf.in, and dakota_scbdo_short_column_mf.in input files solve cantilever, Rosenbrock, and short column OUU problems using multifidelity polynomial chaos-based and stochastic collocation-based approaches.

17.2.6 Epistemic OUU

An emerging capability is optimization under epistemic uncertainty. As described in the Nested Model section of the Reference Manual [3], epistemic and mixed aleatory/epistemic uncertainty quantification methods generate lower and upper interval bounds for all requested response, probability, reliability, and generalized reliability level mappings. Design for robustness in the presence of epistemic uncertainty could simply involve minimizing the range of these intervals (subtracting lower from upper using the nested model response mappings), and design for reliability in the presence of epistemic uncertainty could involve controlling the worst case upper or lower bound of the interval.

We now have the capability to perform epistemic analysis by using interval optimization on the "outer loop" to calculate bounding statistics of the aleatory uncertainty on the "inner loop." Preliminary studies [32] have shown this approach is more efficient and accurate than nested sampling (which was described in Section 17.1.2). This approach uses an efficient global optimization method for the outer loop and stochastic expansion methods (e.g. polynomial chaos or stochastic collocation on the inner loop). The interval optimization is described in Section 6.5.1. Example input files demonstrating the use of interval estimation for epistemic analysis, specifically in epistemic-aleatory nesting, are: dakota_uq_cantilever_sop_exp.in, and dakota_short_column_sop_exp.in.

17.3 Surrogate-Based Uncertainty Quantification

Many uncertainty quantification (UQ) methods are computationally costly. For example, sampling often requires many function evaluations to obtain accurate estimates of moments or percentile values of an output distribution. One approach to overcome the computational cost of sampling is to evaluate the true function (e.g. run the analysis driver) on a fixed, small set of samples, use these sample evaluations to create a response surface approximation (e.g. a surrogate model or meta-model) of the underlying "true" function, then perform random sampling (using thousands or millions of samples) on the approximation to obtain estimates of the mean, variance, and percentiles of the response.

This approach, called "surrogate-based uncertainty quantification" is easy to do in DAKOTA, and one can set up input files to compare the results using no approximation (e.g. determine the mean, variance, and percentiles of the output directly based on the initial sample values) with the results obtained by sampling a variety of surrogate approximations. Example input files of a standard UQ analysis based on sampling alone vs. sampling a surrogate are shown in the dakota_uq_sampling.in and dakota_surr_uq.in in the Dakota/examples/methods directory.

Note that one must exercise some caution when using surrogate-based methods for uncertainty quantification. In general, there is not a single, straightforward approach to incorporate the error of the surrogate fit into the uncertainty estimates of the output produced by sampling the surrogate. Two references which discuss some of the related issues are [51] and [108]. The first reference shows that statistics of a response based on a surrogate model

were less accurate, and sometimes biased, for surrogates constructed on very small sample sizes. In many cases, however, [51] shows that surrogate-based UQ performs well and sometimes generates more accurate estimates of statistical quantities on the output. The second reference goes into more detail about the interaction between sample type and response surface type (e.g., are some response surfaces more accurate when constructed on a particular sample type such as LHS vs. an orthogonal array?) In general, there is not a strong dependence of the surrogate performance with respect to sample type, but some sample types perform better with respect to some metrics and not others (for example, a Hammersley sample may do well at lowering root mean square error of the surrogate fit but perform poorly at lowering the maximum absolute deviation of the error). Much of this work is empirical and application dependent. If you choose to use surrogates in uncertainty quantification, we strongly recommend trying a variety of surrogates and examining diagnostic goodness-of-fit metrics.

Chapter 18

Advanced Simulation Code Interfaces

18.1 Building an Interface to a Engineering Simulation Code

To interface an engineering simulation package to DAKOTA using one of the black-box interfaces (system call or fork), pre- and post-processing functionality typically needs to be supplied (or developed) in order to transfer the parameters from DAKOTA to the simulator input file and to extract the response values of interest from the simulator's output file for return to DAKOTA (see Figures 1.1 and 12.1). This is often managed through the use of scripting languages, such as C-shell [5], Bourne shell [10], Perl [120], or Python [78]. While these are common and convenient choices for simulation drivers/filters, it is important to recognize that any executable file can be used. If the user prefers, the desired pre- and post-processing functionality may also be compiled or interpreted from any number of programming languages (C, C++, F77, F95, JAVA, Basic, etc.).

In the `Dakota/examples/script_interfaces/generic` directory, a simple example uses the Rosenbrock test function as a mock engineering simulation code. Several scripts have been included to demonstrate ways to accomplish the pre- and post-processing needs. Actual simulation codes will, of course, have different pre- and post-processing requirements, and as such, this example serves only to demonstrate the issues associated with interfacing a simulator. Modifications will almost surely be required for new applications.

18.1.1 Generic Script Interface Files

The `generic` directory contains four important files: `dakota_rosenbrock.in` (the DAKOTA input file), `simulator_script` (the simulation driver script), `dprepro` (a pre-processing utility), and `templatedir/ros.template` (a template simulation input file).

The `dakota_rosenbrock.in` file specifies the study that DAKOTA will perform and, in the interface section, describes the components to be used in performing function evaluations. In particular, it identifies `simulator_script` as its `analysis_driver`, as shown in Figure 18.1.

The `simulator_script` listed in Figure 18.2 is a short C-shell driver script that DAKOTA executes to perform each function evaluation. The names of the parameters and results files are passed to the script on its command line; they are referenced in the script by `$argv[1]` and `$argv[2]`, respectively. The `simulator_script` is divided into three parts: pre-processing, analysis, and post-processing.

In the pre-processing portion, the `simulator_script` uses `dprepro`, a parsing utility, to extract the current variable values from a parameters file (`$argv[1]`) and combine them with the simulator template input file

```
#  DAKOTA INPUT FILE - dakota_rosenbrock.in
#  This sample Dakota input file optimizes the Rosenbrock function.
#  See p. 95 in Practical Optimization by Gill, Murray, and Wright.

method,
        npsol_sqp
# if NPSOL is not available, comment the above and try the following instead:
##      conmin_frcg

variables,
        continuous_design = 2
          cdv_initial_point   -1.0      1.0
          cdv_lower_bounds    -2.0     -2.0
          cdv_upper_bounds     2.0      2.0
          cdv_descriptor       'x1'     'x2'

interface,
        system
#         asynchronous
          analysis_driver = 'simulator_script'
          parameters_file = 'params.in'
          results_file    = 'results.out'
          work_directory directory_tag
          template_directory = 'templatedir'
# uncomment to leave params.in and results.out files in work_dir subdirectories
#         named 'workdir' file_save  directory_save
          aprepro
# when using conmin_frcg (above) with analytic_gradients (below),
# need to turn off the active set vector as rosenbrock_bb does not parse it.
##        deactivate active_set_vector

responses,
        num_objective_functions = 1
        numerical_gradients
          fd_gradient_step_size = .000001
# to instead use analytic gradients returned by the simulator comment the
# preceding two lines and uncomment the following:
##      analytic_gradients

        no_hessians
```

Figure 18.1: The `dakota_rosenbrock.in` input file.

```
#!/bin/csh -f
# Sample simulator to Dakota system call script
# See Advanced Simulation Code Interfaces chapter in Users Manual

# $argv[1] is params.in FROM Dakota
# $argv[2] is results.out returned to Dakota

# --------------
# PRE-PROCESSING
# --------------
# Incorporate the parameters from DAKOTA into the template, writing ros.in
# Use the following line if SNL's APREPRO utility is used instead of DPrePro.
# ../aprepro -c '*' -q --nowarning ros.template ros.in

dprepro $argv[1] ros.template ros.in

# --------
# ANALYSIS
# --------

rosenbrock_bb

# ---------------
# POST-PROCESSING
# ---------------

# extract function value from the simulation output
grep 'Function value' ros.out | cut -c 18- >! results.tmp
# extract gradients from the simulation output (in this case will be ignored
# by DAKOTA if not needed)
grep -i 'Function g' ros.out | cut -c 21- >> results.tmp
mv results.tmp $argv[2]
```

Figure 18.2: The `simulator_script` sample driver script.

(ros.template) to create a new input file (ros.in) for the simulator. Internal to Sandia, the APREPRO utility is often used for this purpose. For external sites where APREPRO is not available, the DPrePro utility mentioned above is an alternative with many of the capabilities of APREPRO that is specifically tailored for use with DAKOTA and is distributed with it (in Dakota/examples/script_interfaces/generic/dprepro, or Dakota/bin in a binary distribution). Additionally, the BPREPRO utility is another alternative to APREPRO (see [122]), and at Lockheed Martin sites, the JPrePost utility is available as a JAVA pre- and post-processor [37]. The dprepro script partially listed in Figure 18.3 will be used here for simplicity of discussion. It can use either DAKOTA's aprepro parameters file format (see Section 11.6.2) or DAKOTA's standard format (see Section 11.6.1), so either option may be selected in the interface section of the DAKOTA input file. The ros.template file listed in Figure 18.4 is a template simulation input file which contains targets for the incoming variable values, identified by the strings "{x1}" and "{x2}". These identifiers match the variable descriptors specified in dakota_rosenbrock.in. The template input file is contrived as Rosenbrock has nothing to do with finite element analysis; it only mimics a finite element code to demonstrate the simulator template process. The dprepro script will search the simulator template input file for fields marked with curly brackets and then create a new file (ros.in) by replacing these targets with the corresponding numerical values for the variables. As noted in the usage information for dprepro and shown in simulator_script, the names for the DAKOTA parameters file ($argv[1]), template file (ros.template), and generated input file (ros.in) must be specified in the dprepro command line arguments.

The second part of the script executes the rosenbrock_bb simulator. The input and output file names, ros.in and ros.out, respectively, are hard-coded into the FORTRAN 77 program rosenbrock_bb.f. When the rosenbrock_bb simulator is executed, the values for x1 and x2 are read in from ros.in, the Rosenbrock function is evaluated, and the function value is written out to ros.out.

The third part performs the post-processing and writes the response results to a file for DAKOTA to read. Using the UNIX "grep" utility, the particular response values of interest are extracted from the raw simulator output and saved to a temporary file (results.tmp). When complete, this file is renamed $argv[2], which in this example is always "results.out". Note that moving or renaming the completed results file avoids any problems with read race conditions (see Section 19.2.1.2).

Because the DAKOTA input file dakota_rosenbrock.in (Figure 18.1) specifies work_directory and directory_tag in its interface section, each invocation of simulator_script wakes up in its own temporary directory, which DAKOTA has populated with the contents of directory templatedir. Having a separate directory for each invocation of simulator_script simplifies the script when the DAKOTA input file specifies asynchronous (so several instances of simulator_script might run simultaneously), as fixed names such as ros.in, ros.out, and results.tmp can be used for intermediate files. If neither asynchronous nor file_tag is specified, and if there is no need (e.g., for debugging) to retain intermediate files having fixed names, then directory_tag offers no benefit and can be omitted. An alternative to directory_tag is to proceed as earlier versions of this chapter — prior to DAKOTA 5.0's introduction of work_directory — recommended: add two more steps to the simulator_script, an initial one to create a temporary directory explicitly and copy templatedir to it if needed, and a final step to remove the temporary directory and any files in it.

When work_directory is specified, DAKOTA adjusts the $PATH seen by simulator_script so that simple program names (i.e., names not containing a slash) that are visible in DAKOTA's directory will also be visible in the work directory. Relative path names — involving an intermediate slash but not an initial one, such as ./rosenbrock_bb or a/bc/rosenbrock_bb — will only be visible in the work directory if a template_directory or template_files specification (see §12.5.5) has made them visible there.

As an example of the data flow on a particular function evaluation, consider evaluation 60. The parameters file for this evaluation consists of:

```
#!/usr/bin/perl

# See documentation at end, or run dprepro --man
#  dprepro [options] parameters_file template_input_file new_input_file

# Copyright (c) 2001, Sandia National Laboratories.
# This software is distributed with DAKOTA under the GNU LGPL.
# For more information, see the README file in the top Dakota directory.

use Getopt::Long;
use Pod::Usage;
# use the following for additional trig functions and floor/ceil/round
#use Math::Trig;
#use POSIX;

# default delimiters are curly braces { }
my $ld = "\{";
my $rd = "\}";
# default output does not reformat numeric fields
my $output_format;
undef $output_format;

# process options for delimiters, format, and check argument validity
process_command_line();

$params_file   = $ARGV[0]; # DAKOTA parameters file (aprepro or standard format)
$template_file = $ARGV[1]; # template simulation input file
$new_file      = $ARGV[2]; # new simulation input file with insertions


# Regular expressions for numeric fields
$e  = "-?(?:\\d+\\.?\\d*|\\.\\d+)[eEdD](?:\\+|-)?\\d+"; # exponential notation
$f  = "-?\\d+\\.\\d*|-?\\.\\d+";                        # floating point
$i  = "-?\\d+";                                         # integer
$ui = "\\d+";                                           # unsigned integer
# Note: these should be protected within () due to OR's
$v  = "$e|$f|$i";       # value (numeric field: exponential, float, or int)
$t  = "\\w+(?::\\w+)*"; # tag (colon separated alphanumeric blocks:)

# Regular expression for an even number of backslashes
my $enbs  = "(?<!\\\\)(\\\\\\\\)*";

################################
# Process DAKOTA parameters file
################################

# Open parameters file for input.
```

Figure 18.3: Partial listing of the `dprepro` script.

```
Title of Model: Rosenbrock black box
***********************************************************************
* Description:  This is an input file to the Rosenbrock black box
*               Fortran simulator.  This simulator is structured so
*               as to resemble the input/output from an engineering
*               simulation code, even though Rosenbrock's function
*               is a simple analytic function.  The node, element,
*               and material blocks are dummy inputs.
*
* Input:  x1 and x2
* Output: objective function value
***********************************************************************
node 1 location 0.0 0.0
node 2 location 0.0 1.0
node 3 location 1.0 0.0
node 4 location 1.0 1.0
node 5 location 2.0 0.0
node 6 location 2.0 1.0
node 7 location 3.0 0.0
node 8 location 3.0 1.0
element 1 nodes 1 3 4 2
element 2 nodes 3 5 6 4
element 3 nodes 5 7 8 6
element 4 nodes 7 9 10 8
material 1 elements 1 2
material 2 elements 3 4
variable 1 {x1}
variable 2 {x2}
end
```

Figure 18.4: Listing of the `ros.template` file

```
{ DAKOTA_VARS     =                     2 }
{ x1              =  1.638247697999295e-01 }
{ x2              =  2.197298209103481e-02 }
{ DAKOTA_FNS      =                     1 }
{ ASV_1           =                     1 }
{ DAKOTA_DER_VARS =                     2 }
{ DVV_1           =                     1 }
{ DVV_2           =                     2 }
{ DAKOTA_AN_COMPS =                     0 }
```

This file is called `workdir/workdir.60/params.in` if the line

```
named 'workdir' file_save  directory_save
```

in Figure 18.1 is uncommented. The first portion of the file indicates that there are two variables, followed by new values for variables `x1` and `x2`, and one response function (an objective function), followed by an active set vector (ASV) value of `1`. The ASV indicates the need to return the value of the objective function for these parameters (see Section 11.7). The `dprepro` script reads the variable values from this file, namely `1.638247697999295e-01` and `2.197298209103481e-02` for `x1` and `x2` respectively, and substitutes them in the {`x1`} and {`x2`} fields of the `ros.template` file. The final three lines of the resulting input file (`ros.in`) then appear as follows:

```
variable 1 1.638247697999295e-01
variable 2 2.197298209103481e-02
end
```

where all other lines are identical to the template file. The `rosenbrock_bb` simulator accepts `ros.in` as its input file and generates the following output to the file `ros.out`:

```
Beginning execution of model: Rosenbrock black box
Set up complete.
Reading nodes.
Reading elements.
Reading materials.
Checking connectivity...OK
****************************************************

Input value for x1 =  0.1638247697999295E+00
Input value for x2 =  0.2197298209103481E-01

Computing solution...Done
***************************************************
Function value =  0.7015563957680092E+00
Function gradient = [ -0.1353509902591768E+01 -0.9731146217930163E+00 ]
```

Next, the appropriate values are extracted from the raw simulator output and returned in the results file. This post-processing is relatively trivial in this case, and the `simulator_script` uses the `grep` and `cut` utilities to extract the value from the "`Function value`" line of the `ros.out` output file and save it to `$argv[2]`, which is the `results.out` file for this evaluation. This single value provides the objective function value requested by the ASV.

After 132 of these function evaluations, the following DAKOTA output shows the final solution using the `rosenbrock_bb` simulator:

```
 Exit NPSOL - Optimal solution found.

 Final nonlinear objective value =   0.1165704E-06

NPSOL exits with INFORM code = 0 (see "Interpretation of output" section in NPSOL manual)

NOTE: see Fortran device 9 file (fort.9 or ftn09)
      for complete NPSOL iteration history.

<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 132 total (132 new, 0 duplicate)
<<<<< Best parameters          =
                     9.9965861667e-01 x1
                     9.9931682203e-01 x2
<<<<< Best objective function  =
                  1.1657044253e-07
<<<<< Best data captured at function evaluation 130

<<<<< Iterator npsol_sqp completed.
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =       0.12 [parent =   0.116982, child =   0.003018]
  Total wall clock =    1.47497
```

18.1.2 Adapting These Scripts to Another Simulation

To adapt this approach for use with another simulator, several steps need to be performed:

1. Create a template simulation input file by identifying the fields in an existing input file that correspond to the variables of interest and then replacing them with {} identifiers (e.g. {cdv_1}, {cdv_2}, etc.) which match the DAKOTA variable descriptors. Copy this template input file to a templatedir that will be used to create working directories for the simulation.

2. Modify the `dprepro` arguments in `simulator_script` to reflect names of the DAKOTA parameters file (previously "`$argv[1]`"), template file name (previously "`ros.template`") and generated input file (previously "`ros.in`"). Alternatively, use APREPRO, BPREPRO, or JPrePost to perform this step (and adapt the syntax accordingly).

3. Modify the analysis section of `simulator_script` to replace the `rosenbrock_bb` function call with the new simulator name and command line syntax (typically including the input and output file names).

4. Change the post-processing section in `simulator_script` to reflect the revised extraction process. At a minimum, this would involve changing the `grep` command to reflect the name of the output file, the string to search for, and the characters to cut out of the captured output line. For more involved post-processing tasks, invocation of additional tools may have to be added to the script.

5. Modify the `dakota_rosenbrock.in` input file to reflect, at a minimum, updated variables and responses specifications.

These nonintrusive interfacing approaches can be used to rapidly interface with simulation codes. While generally custom for each new application, typical interface development time is on the order of an hour or two. Thus, this

approach is scalable when dealing with many different application codes. Weaknesses of this approach include the potential for loss of data precision (if care is not taken to preserve precision in pre- and post-processing file I/O), a lack of robustness in post-processing (if the data capture is too simplistic), and scripting overhead (only noticeable if the simulation time is on the order of a second or less).

If the application scope at a particular site is more focused and only a small number of simulation codes are of interest, then more sophisticated interfaces may be warranted. For example, the economy of scale afforded by a common simulation framework justifies additional effort in the development of a high quality DAKOTA interface. In these cases, more sophisticated interfacing approaches could involve a more thoroughly developed black box interface with robust support of a variety of inputs and outputs, or it might involve intrusive interfaces such as the direct simulation interface discussed below in Section 18.2 or the SAND interface described in Section 1.3.

18.1.3 Additional Examples

A variety of additional examples of black-box interfaces to simulation codes are maintained in the `Dakota/examples/script_interfaces` directory in the source code distribution.

18.2 Developing a Direct Simulation Interface

If a more efficient interface to a simulation is desired (e.g., to eliminate process creation and file I/O overhead) or if a targeted computer architecture cannot accommodate separate optimization and simulation processes (e.g., due to lightweight operating systems on compute nodes of large parallel computers), then linking a simulation code directly with DAKOTA may be desirable. This is an advanced capability of DAKOTA, and it requires a user to have access to (and knowledge of) the DAKOTA source code, as well as the source code of the simulation code.

Three approaches are outlined below for developing direct linking between DAKOTA and a simulation: extension, derivation, and sandwich. For additional information, refer to "Interfacing with DAKOTA as a Library" in the DAKOTA Developers Manual [4].

Once performed, DAKOTA can bind with the new direct simulation interface using the `direct` interface specification in combination with an `analysis_driver`, `input_filter` or `output_filter` specification that corresponds to the name of the new subroutine.

18.2.1 Extension

The first approach to using the direct function capability with a new simulation (or new internal test function) involves *extension* of the existing **DirectFnApplicInterface** class to include new simulation member functions. In this case, the following steps are performed:

1. The functions to be invoked (analysis programs, input and output filters, internal testers) must have their main programs changed into callable functions/subroutines.

2. The resulting callable function can then be added directly to the private member functions in **DirectFnApplicInterface** if this function will directly access the DAKOTA data structures (variables, active set, and response attributes of the class). It is more common to add a wrapper function to **DirectFnApplicInterface** which manages the DAKOTA data structures, but allows the simulator subroutine to retain a level of independence from DAKOTA (see Salinas, ModelCenter, and Matlab wrappers as examples).

3. The if-else blocks in the **derived_map_if()**, **derived_map_ac()**, and **derived_map_of()** member functions of the **DirectFnApplicInterface** class must be extended to include the new function names as options. If the new functions are class member functions, then DAKOTA data access may be performed through the existing class member attributes and data objects do not need to be passed through the function parameter list. In this case, the following function prototype is appropriate:

   ```
   int function_name();
   ```

   If, however, the new function names are not members of the **DirectFnApplicInterface** class, then an `extern` declaration may additionally be needed and the function prototype should include passing of the Variables, ActiveSet, and Response data members:

   ```
   int function_name(const Dakota::Variables& vars,
                     const Dakota::ActiveSet& set, Dakota::Response& response);
   ```

4. The DAKOTA system must be recompiled and linked with the new function object files or libraries.

Various header files may have to be included, particularly within the **DirectFnApplicInterface** class, in order to recognize new external functions and compile successfully. Refer to the DAKOTA Developers Manual [4] for additional information on the **DirectFnApplicInterface** class and the DAKOTA data types.

18.2.2 Derivation

As described in "Interfacing with DAKOTA as a Library" in the DAKOTA Developers Manual [4], a derivation approach can be employed to further increase the level of independence between DAKOTA and the host application. In this case, rather than *adding* a new function to the existing **DirectFnApplicInterface** class, a new interface class is derived from **DirectFnApplicInterface** which *redefines* the **derived_map_if()**, **derived_map_ac()**, and **derived_map_of()** virtual functions.

In the approach of Section 18.2.3 below, the class derivation approach avoids the need to recompile the DAKOTA library when the simulation or its direct interface class is modified.

18.2.3 Sandwich

In a "sandwich" implementation, a simulator provides both the "front end" and "back end" with DAKOTA sandwiched in the middle. To accomplish this approach, the simulation code is responsible for interacting with the user (the front end), links DAKOTA in as a library (refer to "Interfacing with DAKOTA as a Library" in the DAKOTA Developers Manual [4]), and plugs in a derived direct interface class to provide a closely-coupled mechanism for performing function evaluations (the back end). This approach makes DAKOTA services available to other codes and frameworks and is currently used by Sandia codes such as Xyce (electrical simulation), Sage (CFD), and SIERRA (multiphysics).

18.3 Existing Direct Interfaces to External Simulators

In addition to built-in polynomial test functions described in Section 12.3.1, DAKOTA includes direct interfaces to Sandia's Salinas code for structural dynamics, Phoenix Integration's ModelCenter framework, The Mathworks' Matlab scientific computing environment, and Python. While these can be interfaced to with a script-based approach, some usability and efficiency gains may be realized by re-compiling DAKOTA with these direct interfaces

enabled. Some details on Matlab and Python interfaces are provided here. Note that these capabilities permit using Matlab or Python to evaluate a parameter to response mapping; they do not make DAKOTA algorithms available as a service, i.e., as a Matlab toolbox or Python module.

18.3.1 Matlab

DAKOTA's direct function interface includes the capability to invoke Matlab for function evaluations, using the Matlab engine API. When using this close-coupling, the Matlab engine is started once when DAKOTA initializes, and then during analysis function evaluations are performed exchanging parameters and results through the Matlab C API. This eliminates the need to use the file system and the expense of initializing the Matlab engine for each function evaluation.

The DAKOTA/Matlab interface has been built and tested on 32-bit Linux with Matlab 7.0 (R14) and on 64-bit Linux with Matlab 7.1 (R14SP3). Configuration support for Solaris is included, but is untested. Builds on other platforms or with other versions of Matlab may require modifications to DAKOTA including its build system

To use the DAKOTA/Matlab interface, DAKOTA must be configured and compiled with the Matlab feature enabled (currently only supported in autotools builds of DAKOTA). The Mathworks only provides shared object libraries for its engine API, so DAKOTA must be dynamically linked to at least the Matlab libraries. To compile DAKOTA with the Matlab interface enabled, one might, for example:

1. Configure DAKOTA with (at minimum) the following directive:

   ```
   configure --with-matlab=$MATLAB_ROOT
   ```

   where `$MATLAB_ROOT` is the root of your Matlab installation (it should be a directory containing directories bin/YOURPLATFORM and extern/include).

2. Since the Matlab libraries are linked dynamically, they must be accessible at compile time and at run time. Make sure the path to the appropriate Matlab shared object libraries is on your `LD_LIBRARY_PATH`. For example to accomplish this in BASH on 32-bit Linux, one might type

   ```
   export LD_LIBRARY_PATH=/usr/local/matlab/bin/glnx86:$LD_LIBRARY_PATH
   ```

   or add such a command to the .bashrc file. Then proceed with compiling as usual.

Example files corresponding to the following tutorial are available in `Dakota/examples/linked_interfaces/Matlab`.

18.3.1.1 DAKOTA/Matlab input file specification

The use of the Matlab direct interface is specified through the combination of the `direct` and `analysis_driver` keywords in an interface specification. The Matlab m-file which performs the analysis is specified through the `analysis_components` keyword. Here is a sample DAKOTA `interface` specification:

```
interface,
  direct
    analysis_driver = 'matlab'
    analysis_components = 'myanalysis.m'
```

```
Dakota.
  numFns              number of functions (responses, constraints)
  numVars             total number of variables
  numACV              number active continuous variables
  numADIV             number active discrete integer variables
  numADRV             number active discrete real variables
  numDerivVars        number of derivative variables specified in directFnDVV
  xC                  continuous variable values ([1 x numACV])
  xDI                 discrete integer variable values ([1 x numADIV])
  xDR                 discrete real variable values ([1 x numADRV])
  xCLabels            continuous var labels (cell array of numACV strings)
  xDILabels           discrete integer var labels (cell array of numADIV strings)
  xDRLabels           discrete real var labels (cell array of numADIV strings)
  directFnASV         active set vector ([1 x numFns])
  directFnASM         active set matrix ([3 x numFns]) -- see below
  directFnDVV         derivative variables vector ([1 x numDerivVars])
  directFnDVV_bool    boolean DVV ([1 x numVars]) -- see below
  fnFlag              nonzero if function values requested
  gradFlag            nonzero if gradients requested
  hessFlag            nonzero if hessians requested
  currEvalId          current evaluation ID
```

Figure 18.5: DAKOTA/Matlab parameter data structure.

Multiple Matlab analysis drivers are supported and may be specified in combination with other direct analysis drivers. Multiple analysis components are supported as for other interfaces as described in Section 12.4. The `.m` extension in the `analysis_component` specification is optional; in fact any characters following the first period in the `analysis_component` specification will be stripped off by the interface, so `myanalysis`, `myanalysis.m`, and `myanalysis.fun.m` will all cause the interface to attempt to execute a Matlab script called `myanalysis.m` for the function evaluation.

18.3.1.2 Matlab .m file specification

The Matlab analysis file `myanalysis.m` must define a Matlab function that accepts a Matlab structure as its sole argument and returns the same structure in a variable called `Dakota`. A manual execution of the call to the analysis in Matlab should therefore look like:

```
>> Dakota = myanalysis(Dakota)
```

Note that the structure named Dakota will be pushed into the Matlab workspace before the analysis function is called. The structure passed from DAKOTA to the analysis m-function contains essentially the same information that would be passed to a DAKOTA direct function included in `DirectFnApplicInterface.C`, with fields shown in Figure 18.5.

The `directFnASM` field is a boolean matrix version of the active set vector, designed to make it easier for the Matlab user to determine which responses Dakota requires. Its rows 1, 2, and 3 correspond to functions, gradients, and Hessians, respectively, and its columns to each function (response, constraint) from 1–`numFns`. A `directFnASM` matrix entry contains a 1 if the response is needed for that function and a 0 otherwise. For example, for the fictitious

```
directFnASV = [ 1, 2, 3, 4, 5, 6, 7 ],
```

```
Dakota.
  fnVals      ([1 x numFns], required if function values requested)
  fnGrads     ([numFns x numDerivVars], required if gradients  requested)
  fnHessians  ([numFns x numDerivVars x numDerivVars],
               required if hessians requested)
  fnLabels    (cell array of numFns strings, optional)
  failure     (optional: zero indicates success, nonzero failure
```

Figure 18.6: DAKOTA/Matlab response data structure.

the corresponding

```
directFnASM = [ 1, 0, 1, 0, 1, 0, 1      (functions)
                0, 1, 1, 0, 0, 1, 1      (gradients)
                0, 0 ,0 ,1, 1, 1, 1 ]    (hessians).
```

The `directFnDVV_bool` field is a similar boolean structure for the `directFnDVV`. It is a `[1 x numVars]` vector with entry 1 if a derivative with respect to that variable is requested and a 0 otherwise. So for an example with 6 functions if

```
directFnDVV = [ 1, 3, 4, 6 ],
```

the corresponding

```
directFnDVV_bool = [ 1, 0, 1, 1, 0, 1 ].
```

The structure `Dakota` returned from the analysis must contain a subset of the fields shown in Figure 18.6. It may contain additional fields and in fact is permitted to be the structure passed in, augmented with any required outputs.

An example Matlab analysis driver `rosenbrock.m` for the Rosenbrock function is shown in Figure 18.7.

18.3.2 Python

DAKOTA's Python direct interface has been tested on Linux with Python 2.x. When enabled, it allows DAKOTA to make function evaluation calls directly to an analysis function in a user-provided Python module. Data may flow between DAKOTA and Python either in multiply-subscripted lists or NumPy arrays.

The Python direct interface must be enabled when compiling DAKOTA (and is only supported in autotools builds):

```
configure --with-python
```

Configure will query the first available Python installation on the PATH to determine availability of necessary development headers and libraries. If the available Python installation includes NumPy (detected via arrayobject.h), then DAKOTA will default to a NumPy array interface. If not available, or if the variable DirectApplicInterface::userNumpyFlag is forced to be false, DAKOTA will use multiply-subscripted lists for data flow.

An example of using the Python direct interface with both lists and arrays is included in `examples/linked_interfaces/Python`. The Python direct driver is selected with, for example,

```
function Dakota = rosenbrock(Dakota)

  Dakota.failure = 0;

  if ( Dakota.numVars ~= 2 | Dakota.numADV | ...
      ( ~isempty( find(Dakota.directFnASM(2,:)) | ...
      find(Dakota.directFnASM(3,:)) ) & Dakota.numDerivVars ~= 2 ) )

    sprintf('Error: Bad number of variables in rosenbrock.m fn.\n');
    Dakota.failure = 1;

  elseif (Dakota.numFns > 2)

    % 1 fn -> opt, 2 fns -> least sq
    sprintf('Error: Bad number of functions in rosenbrock.m fn.\n');
    Dakota.failure = 1;

  else

    if Dakota.numFns > 1
      least_sq_flag = true;
    else
      least_sq_flag = false;
    end

    f0 = Dakota.xC(2)-Dakota.xC(1)*Dakota.xC(1);
    f1 = 1.-Dakota.xC(1);

    % **** f:
    if (least_sq_flag)
      if Dakota.directFnASM(1,1)
        Dakota.fnVals(1) = 10*f0;
      end
      if Dakota.directFnASM(1,2)
        Dakota.fnVals(2) = f1;
      end
    else
      if Dakota.directFnASM(1,1)
        Dakota.fnVals(1) = 100.*f0*f0+f1*f1;
      end
    end

    % **** df/dx:
    if (least_sq_flag)
      if Dakota.directFnASM(2,1)
        Dakota.fnGrads(1,1) = -20.*Dakota.xC(1);
        Dakota.fnGrads(1,2) =  10.;
      end
      if Dakota.directFnASM(2,2)
        Dakota.fnGrads(2,1) = -1.;
        Dakota.fnGrads(2,2) =  0.;
      end

    else

      if Dakota.directFnASM(2,1)
        Dakota.fnGrads(1,1) = -400.*f0*Dakota.xC(1) - 2.*f1;
        Dakota.fnGrads(1,2) =  200.*f0;
      end

    end

    % **** d^2f/dx^2:
    if (least_sq_flag)

      if Dakota.directFnASM(3,1)
        Dakota.fnHessians(1,1,1) = -20.;
        Dakota.fnHessians(1,1,2) = 0.;
        Dakota.fnHessians(1,2,1) = 0.;
        Dakota.fnHessians(1,2,2) = 0.;
      end
      if Dakota.directFnASM(3,2)
        Dakota.fnHessians(2,1:2,1:2) = 0.;
      end

    else

      if Dakota.directFnASM(3,1)
        fx = Dakota.xC(2) - 3.*Dakota.xC(1)*Dakota.xC(1);
        Dakota.fnHessians(1,1,1) = -400.*fx + 2.0;
        Dakota.fnHessians(1,1,2) = -400.*Dakota.xC(1);
        Dakota.fnHessians(1,2,1) = -400.*Dakota.xC(1);
        Dakota.fnHessians(1,2,2) =  200.;
      end

    end

    Dakota.fnLabels = {'f1'};

  end
```

Figure 18.7: Sample Matlab implementation of the Rosenbrock test function for the DAKOTA/Matlab interface.

Table 18.1: Data dictionary passed to Python direct interface.

Entry Name	Description
functions	number of functions (responses, constraints)
variables	total number of variables
cv	list/array of continuous variable values
div	list/array of discrete integer variable values
drv	list/array of discrete real variable values
av	single list/array of all variable values
cv_labels	continuous variable labels
div_labels	discrete integer variable labels
drv_labels	discrete real variable labels
av_labels	all variable labels
asv	active set vector
dvv	derivative variables vector
currEvalId	current evaluation ID number

```
analysis_drivers = 'python'
  analysis_components = 'python_module:analysis_function'
```

where `python_module` denotes the module (file `python_module.py`) DAKOTA will attempt to import into the Python environment and `analysis_function` denotes the function to call when evaluating a parameter set. It is likely necessary to set the `PYTHONPATH` environment variable to include the location of the Python module you wish to load.

Whether using the list or array interface, data from DAKOTA is passed (via kwargs) into the user function in a dictionary containing the entries shown in Table 18.1. The `analysis_function` must return a dictionary containing the data specified by the active set vector with fields "fns", "fnGrads", and "fnHessians", corresponding to function values, gradients, and Hessians, respectively. The function may optionally include a failure code in "failure" (zero indicates success, nonzero failure) and function labels in "fnLabels". See the linked interfaces example referenced above for more details.

Chapter 19

Parallel Computing

19.1 Overview

This chapter describes the various parallel computing capabilities provided by DAKOTA. The range of capabilities is extensive and can be daunting at first; therefore, this chapter takes an incremental approach in first describing the simplest single-level parallel computing models (Section 19.2) using asynchronous local, message passing, and hybrid approaches. More advanced uses of DAKOTA can build on this foundation to exploit multiple levels of parallelism, as described in Section 19.3. The chapter concludes with a discussion of using DAKOTA with applications that run as independent MPI processes (parallel application tiling, for example on a large compute cluster). This last section is a good quick start for interfacing DAKOTA to your parallel (or serial) application on a cluster.

19.1.1 Categorization of parallelism

To understand the parallel computing possibilities, it is instructive to first categorize the opportunities for exploiting parallelism into four main areas [28], consisting of coarse-grained and fine-grained parallelism opportunities within algorithms and their function evaluations:

1. *Algorithmic coarse-grained parallelism*: This parallelism involves the concurrent execution of independent function evaluations, where a "function evaluation" is defined as a data request from an algorithm (which may involve value, gradient, and Hessian data from multiple objective and constraint functions). This concept can also be extended to the concurrent execution of multiple "iterators" within a "strategy." Examples of algorithms containing coarse-grained parallelism include:

    - *Gradient-based algorithms*: finite difference gradient evaluations, speculative optimization, parallel line search.
    - *Nongradient-based algorithms*: genetic algorithms (GAs), pattern search (PS), Monte Carlo sampling.
    - *Approximate methods*: design of computer experiments for building surrogate models.
    - *Concurrent-iterator strategies*: optimization under uncertainty, branch and bound, multi-start local search, Pareto set optimization, island-model GAs.

2. *Algorithmic fine-grained parallelism*: This involves computing the basic computational steps of an optimization algorithm (i.e., the internal linear algebra) in parallel. This is primarily of interest in large-scale optimization problems and simultaneous analysis and design (SAND).

3. *Function evaluation coarse-grained parallelism*: This involves concurrent computation of separable parts of a single function evaluation. This parallelism can be exploited when the evaluation of the response data set requires multiple independent simulations (e.g. multiple loading cases or operational environments) or multiple dependent analyses where the coupling is applied at the optimizer level (e.g., multiple disciplines in the individual discipline feasible formulation [18]).

4. *Function evaluation fine-grained parallelism*: This involves parallelization of the solution steps within a single analysis code. The DOE laboratories have developed parallel analysis codes in the areas of nonlinear mechanics, structural dynamics, heat transfer, computational fluid dynamics, shock physics, and many others.

By definition, coarse-grained parallelism requires very little inter-processor communication and is therefore "embarrassingly parallel," meaning that there is little loss in parallel efficiency due to communication as the number of processors increases. However, it is often the case that there are not enough separable computations on each algorithm cycle to utilize the thousands of processors available on MP machines. For example, a thermal safety application [29] demonstrated this limitation with a pattern search optimization in which the maximum speedup exploiting *only* coarse-grained algorithmic parallelism was shown to be limited by the size of the design problem (coordinate pattern search has at most $2n$ independent evaluations per cycle for n design variables).

Fine-grained parallelism, on the other hand, involves much more communication among processors and care must be taken to avoid the case of inefficient machine utilization in which the communication demands among processors outstrip the amount of actual computational work to be performed. For example, a chemically-reacting flow application [28] illustrated this limitation for a simulation of fixed size in which it was shown that, while simulation run time did monotonically decrease with increasing number of processors, the relative parallel efficiency $\hat{E}$ of the computation for fixed model size decreased rapidly (from $\hat{E} \approx 0.8$ at 64 processors to $\hat{E} \approx 0.4$ at 512 processors). This was due to the fact that the total amount of computation was approximately fixed, whereas the communication demands were increasing rapidly with increasing numbers of processors. Therefore, there is a practical limit on the number of processors that can be employed for fine-grained parallel simulation of a particular model size, and only for extreme model sizes can thousands of processors be efficiently utilized in studies exploiting fine-grained parallelism alone.

These limitations point us to the exploitation of multiple levels of parallelism, in particular the combination of coarse-grained and fine-grained approaches. This will allow us to execute fine-grained parallel simulations on sets of processors where they are most efficient and then replicate this efficiency with many coarse-grained instances. From a software perspective, coarse-grained parallelism by itself (many instances of a single-processor simulation) and fine-grained parallelism by itself (a single instance of a large multiprocessor simulation) can be considered to cover two ends of a spectrum, and we are interested in also supporting anywhere in between (any number of instances of any size simulation). Single-level parallelism approaches are described in Section 19.2, and multilevel parallelism approaches are discussed in Section 19.3.

The available concurrency in function evaluation parallelism is determined by the aspects of a particular systems analysis application, and is therefore highly application-dependent. Algorithmic parallelism, on the other hand, is largely determined by the selection and configuration of a particular algorithm. These selection possibilities within DAKOTA are outlined in the following section.

19.1.2 Parallel DAKOTA algorithms

In DAKOTA, the following parallel algorithms, comprised of iterators and strategies, provide support for coarse-grained algorithmic parallelism. Note that, even if a particular algorithm is serial in terms of its data request concurrency, other concurrency sources (e.g., function evaluation coarse-grained and fine-grained parallelism) may still be available.

19.1.2.1 Parallel iterators

- Gradient-based optimizers: CONMIN, DOT, NLPQL, NPSOL, and OPT++ can all exploit parallelism through the use of DAKOTA's native finite differencing routine (selected with `method_source dakota` in the responses specification), which will perform concurrent evaluations for each of the parameter offsets. For `n` variables, forward differences result in an $n+1$ concurrency and central differences result in a $2n+1$ concurrency. In addition, CONMIN, DOT, and OPT++ can use speculative gradient techniques [11] to obtain better parallel load balancing. By speculating that the gradient information associated with a given line search point will be used later and computing the gradient information in parallel at the same time as the function values, the concurrency during the gradient evaluation and line search phases can be balanced. NPSOL does not use speculative gradients since this approach is superseded by NPSOL's gradient-based line search in user-supplied derivative mode. NLPQL also supports a distributed line search capability for generating concurrency [99].

- Nongradient-based optimizers: HOPSPACK, JEGA methods, and most SCOLIB methods support parallelism. HOPSPACK and SCOLIB methods exploit parallelism through the use of DAKOTA's concurrent function evaluations; however, there are some limitations on the levels of concurrency and asynchrony that can be exploited. These are detailed in the DAKOTA Reference Manual. Serial SCOLIB methods include Solis-Wets (`coliny_solis_wets`) and certain `exploratory_moves` options (`adaptive_pattern` and `multi_step`) in pattern search (`coliny_pattern_search`). OPT++ PDS (`optpp_pds`) and NCSU DIRECT (`ncsu_direct`) are also currently serial due to incompatibilities in DAKOTA and OPT++/NCSU parallelism models. Finally, `coliny_pattern_search` and `asynch_pattern_search` support dynamic job queues managed with nonblocking synchronization.

- Least squares methods: in an identical manner to the gradient-based optimizers, NL2SOL, NLSSOL, and Gauss-Newton can exploit parallelism through the use of DAKOTA's native finite differencing routine. In addition, NL2SOL and Gauss-Newton can use speculative gradient techniques to obtain better parallel load balancing. NLSSOL does not use speculative gradients since this approach is superseded by NLSSOL's gradient-based line search in user-supplied derivative mode.

- Surrogate-based minimizers: `surrogate_based_local`, `surrogate_based_global`, and `efficient_global` all support parallelism in the initial surrogate construction, but subsequent concurrency varies. In the case of `efficient_global`, only a single point is generated for evaluation for each subsequent cycle and there is no derivatove concurrency for this point. In the case of `surrogate_based_local`, only a single point is generated per subsequent cycle, but derivative concurrency for numerical gradient or Hessian evaluations may be available. And in the case of `surrogate_based_global`, multiple points may be generated on each subsequent cycle, depending on the multipoint return capability of specific minimizers.

- Parameter studies: all parameter study methods (`vector`, `list`, `centered`, and `multidim`) support parallelism. These methods avoid internal synchronization points, so all evaluations are available for concurrent execution.

- Design of experiments: all `dace` (`grid`, `random`, `oas`, `lhs`, `oa_lhs`, `box_behnken`, and `central_composite`),

fsu_quasi_mc (halton and hammersley), fsu_cvt, and psuade_moat methods support parallelism.

- Uncertainty quantification: all nondeterministic methods (sampling, local_reliability, global_reliability, polynomial_chaos, stoch_collocation,local_interval_est, global_interval_est,local_evidence and global_evidence) support parallelism. In the case of local_reliability, gradient-based optimization is involved and parallelism can be exploited through the use of DAKOTA's native finite differencing routine. In the case of global_reliability, EGRA methods support parallelism in the initial surrogate construction, but subsequently only generate a single point for evaluation per cycle.

19.1.2.2 Parallel strategies

Certain strategies support concurrency in multiple iterator executions. Currently, the strategies which can exploit this level of parallelism are:

- Hybrid minimization: when the sequential hybrid transfers multiple solution points between methods, single-point minimizers will be executed concurrently using each of the transferred solution points.
- Pareto-set optimization: multiobjective optimization strategy for computing sets of points on the Pareto front of nondominated solutions.
- Multi-start iteration: strategy for executing multiple instances of an iterator from different starting points.

In the branch and bound case, the available iterator concurrency grows as the tree develops more branches, so some of the iterator servers may be idle in the initial phases. Similarly, hybrid minimization will display varying levels of iterator concurrency based on differing support of multipoint solution input/output between iterators; however, the use of multiple parallel configurations among the iterator sequence should prevent parallel inefficiencies. Finally, pareto-set and multi-start have a fixed set of jobs to perform and should exhibit good load balancing.

19.1.2.3 Parallel models

Parallelism support in model classes (see Chapter 10) is an important issue for variable scaling (see Section 7.3.2) and advanced model recursions such as surrogate-based minimization, optimization under uncertainty, and second-order probability (see Chapters 9 and 17). Support is as follows:

- Single model: parallelism is managed by the singe interface instance.
- Recast model: most parallelism is forwarded on to the sub-model. An exception to this is finite differencing in the presence of variable scaling. Since it is desirable to perform offsets in the scaled space (and avoid minimum step size tolerances), this parallelism is not forwarded to the sub-model, instead being enacted at the recast level.
- Data fit surrogate model: parallelism is supported in the construction of global surrogate models via the concurrent evaluation of points generated by design of experiments methods. Local and multipoint approximations evaluate only a single point at a time, so concurrency is available only from any numerical differencing required for gradient and Hessian data. Since the top-level iterator is interfaced only with the (inexpensive) surrogate, no parallelism is exploited here. Load balancing can be an important issue when performing evaluations for update of existing surrogate models.

- Hierarchical surrogate model: parallelism is supported for both the low and high fidelity models. Since the top-level iterator is interfaced only with the low-fidelity model, and the high-fidelity model is used only for verifications and correction updating, the algorithmic coarse-grained parallelism supported by the iterator is enacted on the low fidelity model and the only parallelism available for high fidelity executions arises from any numerical differencing required for high-fidelity gradient and Hessian data.

- Nested model: Currently, nested models only support concurrency in the sub-iterator execution on the sub-model. This means that the top-level iterator interfaced with the nested model is serialized. In future releases, concurrency will be supported at this top level, allowing techniques such as optimization under uncertainty and second-order probability (see Section 10.5) to support concurrent iterator parallelism.

19.2 Single-level parallelism

DAKOTA's parallel facilities support a broad range of computing hardware, from custom massively parallel supercomputers on the high end, to clusters and networks of workstations (NOWs) in the middle range, to desktop multiprocessors on the low end. Given the reduced scale in the middle to low ranges, it is more common to exploit only one of the levels of parallelism; however, this can still be quite effective in reducing the time to obtain a solution. Three single-level parallelism models will be discussed, and are depicted in Figure 19.1:

Figure 19.1: External, internal, and hybrid job management.

- *asynchronous local*: DAKOTA executes on a single processor, but launches multiple jobs concurrently using asynchronous job launching techniques.

- *message passing*: DAKOTA executes in parallel using message passing to communicate between processors. A single job is launched per processor using synchronous job launching techniques.

- *hybrid*: a combination of message passing and asynchronous local. DAKOTA executes in parallel across multiple processors and launches concurrent jobs on each processor.

In each of these cases, jobs are executing concurrently and must be collected in some manner for return to an algorithm. Blocking and nonblocking approaches are provided for this, where the blocking approach is used in most cases:

- *blocking synchronization*: all jobs in the queue are completed before exiting the scheduler and returning the set of results to the algorithm. The job queue fills and then empties completely, which provides a synchronization point for the algorithm.
- *nonblocking synchronization*: the job queue is dynamic, with jobs entering and leaving continuously. There are no defined synchronization points for the algorithm, which requires specialized algorithm logic (only currently supported by `coliny_pattern_search` and `asynch_pattern_search`, which are sometimes referred to as "fully asynchronous" algorithms).

Given these job management capabilities, it is worth noting that the popular term "asynchronous" can be ambiguous when used in isolation. In particular, it can be important to qualify whether one is referring to "asynchronous job launch" (synonymous with any of the three concurrent job launch approaches described above) or "asynchronous job recovery" (synonymous with the latter nonblocking job synchronization approach).

19.2.1 Asynchronous Local Parallelism

This section describes software components which manage simulation invocations local to a processor. These invocations may be either synchronous (i.e., blocking) or asynchronous (i.e., nonblocking). Synchronous evaluations proceed one at a time with the evaluation running to completion before control is returned to DAKOTA. Asynchronous evaluations are initiated such that control is returned to DAKOTA immediately, prior to evaluation completion, thereby allowing the initiation of additional evaluations which will execute concurrently.

The synchronous local invocation capabilities are used in two contexts: (1) by themselves to provide serial execution on a single processor, and (2) in combination with DAKOTA's message-passing schedulers to provide function evaluations local to each processor. Similarly, the asynchronous local invocation capabilities are used in two contexts: (1) by themselves to launch concurrent jobs from a single processor that rely on external means (e.g., operating system, job queues) for assignment to other processors, and (2) in combination with DAKOTA's message-passing schedulers to provide a hybrid parallelism (see Section 19.2.3). Thus, DAKOTA supports any of the four combinations of synchronous or asynchronous local combined with message passing or without.

Asynchronous local schedulers may be used for managing concurrent function evaluations requested by an iterator or for managing concurrent analyses within each function evaluation. The former iterator/evaluation concurrency supports either blocking (all jobs in the queue must be completed by the scheduler) or nonblocking (dynamic job queue may shrink or expand) synchronization, where blocking synchronization is used by most iterators and nonblocking synchronization is used by fully asynchronous algorithms such as `asynch_pattern_search` and `coliny_pattern_search`. The latter evaluation/analysis concurrency is restricted to blocking synchronization. The "Asynchronous Local" column in Table 19.1 summarizes these capabilities.

DAKOTA supports three local simulation invocation approaches based on the direct function, system call, and fork simulation interfaces. For each of these cases, an input filter, one or more analysis drivers, and an output filter make up the interface, as described in Section 12.4.

19.2.1.1 Direct function synchronization

The direct function capability may be used synchronously. Synchronous operation of the direct function simulation interface involves a standard procedure call to the input filter, if present, followed by calls to one or more

simulations, followed by a call to the output filter, if present (refer to Sections 12.3-12.4 for additional details and examples). Each of these components must be linked as functions within DAKOTA. Control does not return to the calling code until the evaluation is completed and the response object has been populated.

Asynchronous operation will be supported in the future and will involve the use of multithreading (e.g., POSIX threads) to accomplish multiple simultaneous simulations. When spawning a thread (e.g., using `pthread_create`), control returns to the calling code after the simulation is initiated. In this way, multiple threads can be created simultaneously. An array of responses corresponding to the multiple threads of execution would then be recovered in a synchronize operation (e.g., using `pthread_join`).

19.2.1.2 System call synchronization

The system call capability may be used synchronously or asynchronously. In both cases, the `system` utility from the standard C library is used. Synchronous operation of the system call simulation interface involves spawning the system call (containing the filters and analysis drivers bound together with parentheses and semi-colons) in the foreground. Control does not return to the calling code until the simulation is completed and the response file has been written. In this case, the possibility of a race condition (see below) does not exist and any errors during response recovery will cause an immediate abort of the DAKOTA process (note: detection of the string "fail" is not a response recovery error; see Chapter 22).

Asynchronous operation involves spawning the system call in the background, continuing with other tasks (e.g., spawning other system calls), periodically checking for process completion, and finally retrieving the results. An array of responses corresponding to the multiple system calls is recovered in a synchronize operation.

In this synchronize operation, completion of a function evaluation is detected by testing for the existence of the evaluation's results file using the `stat` utility [72]. Care must be taken when using asynchronous system calls since they are prone to the race condition in which the results file passes the existence test but the recording of the function evaluation results in the file is incomplete. In this case, the read operation performed by DAKOTA will result in an error due to an incomplete data set. In order to address this problem, DAKOTA contains exception handling which allows for a fixed number of response read failures per asynchronous system call evaluation. The number of allowed failures must have a limit, so that an actual response format error (unrelated to the race condition) will eventually abort the system. Therefore, to reduce the possibility of exceeding the limit on allowable read failures, *the user's interface should minimize the amount of time an incomplete results file exists in the directory where its status is being tested*. This can be accomplished through two approaches: (1) delay the creation of the results file until the simulation computations are complete and all of the response data is ready to be written to the results file, or (2) perform the simulation computations in a subdirectory, and as a last step, move the completed results file into the main working directory where its existence is being queried.

If concurrent simulations are executing in a shared disk space, then care must be taken to maintain independence of the simulations. In particular, the parameters and results files used to communicate between DAKOTA and the simulation, as well as any other files used by this simulation, must be protected from other files of the same name used by the other concurrent simulations. With respect to the parameters and results files, these files may be made unique through the use of the `file_tag` option (e.g., `params.in.1`, `results.out.1`, etc.) or the default UNIX temporary file option (e.g., `/var/tmp/aaa0b2Mfv`, etc.). However, if additional simulation files must be protected (e.g., `model.i`, `model.o`, `model.g`, `model.e`, etc.), then an effective approach is to create a tagged working subdirectory for each simulation instance. Section 18.1 provides an example system call interface that demonstrates both the use of tagged working directories and the relocation of completed results files to avoid the race condition.

19.2.1.3 Fork synchronization

The fork capability is quite similar to the system call; however, it has the advantage that asynchronous fork invocations can avoid the results file race condition that may occur with asynchronous system calls (see Section 12.3.5). The fork interface invokes the filters and analysis drivers using the `fork` and `exec` family of functions, and completion of these processes is detected using the `wait` family of functions. Since `wait` is based on a process id handle rather than a file existence test, an incomplete results file is not an issue.

Depending on the platform, the fork simulation interface executes either a `vfork` or a `fork` call. These calls generate a new child process with its own UNIX process identification number, which functions as a copy of the parent process (dakota). The `execvp` function is then called by the child process, causing it to be replaced by the analysis driver or filter. For synchronous operation, the parent dakota process then awaits completion of the forked child process through a blocking call to `waitpid`. On most platforms, the `fork/exec` procedure is efficient since it operates in a copy-on-write mode, and no copy of the parent is actually created. Instead, the parents address space is borrowed until the `exec` function is called.

The `fork/exec` behavior for asynchronous operation is similar to that for synchronous operation, the only difference being that dakota invokes multiple simulations through the `fork/exec` procedure prior to recovering response results for these jobs using the `wait` function. The combined use of `fork/exec` and `wait` functions in asynchronous mode allows the scheduling of a specified number of concurrent function evaluations and/or concurrent analyses.

19.2.1.4 Asynchronous Local Example

The test file `Dakota/test/dakota_dace.in` computes 49 orthogonal array samples, which may be evaluated concurrently using parallel computing. When executing DAKOTA with this input file on a single processor, the following execution syntax may be used:

```
dakota -i dakota_dace.in
```

For serial execution (the default), the interface specification within `dakota_dace.in` would appear similar to

```
interface,
        system
          analysis_driver = 'text_book'
```

which results in function evaluation output similar to the following (for `output` set to `quiet` mode):

```
>>>>> Running dace iterator.

------------------------------
Begin Function Evaluation    1
------------------------------
(text_book /tmp/fileG32LEp /tmp/fileP8uYDC)

------------------------------
Begin Function Evaluation    2
------------------------------
(text_book /tmp/fileiqIEEP /tmp/fileBEFlF2)

<snip>
```

```
------------------------------
Begin Function Evaluation   49
------------------------------
(text_book /tmp/file4Xyp2p /tmp/filezCohcE)

<<<<< Iterator dace completed.
```

where it is evident that each function evaluation is being performed sequentially.

For parallel execution using asynchronous local approaches, the DAKOTA execution syntax is unchanged as DAKOTA is still launched on a single processor. However, the interface specification is augmented to include the `asynchronous` keyword with optional concurrency limiter to indicate that multiple `analysis_driver` instances will be executed concurrently:

```
interface,
        system asynchronous evaluation_concurrency = 4
          analysis_driver = 'text_book'
```

which results in output excerpts similar to the following:

```
>>>>> Running dace iterator.

------------------------------
Begin Function Evaluation    1
------------------------------
(Asynchronous job 1 added to queue)

------------------------------
Begin Function Evaluation    2
------------------------------
(Asynchronous job 2 added to queue)

<snip>

------------------------------
Begin Function Evaluation   49
------------------------------
(Asynchronous job 49 added to queue)

Blocking synchronize of 49 asynchronous evaluations
First pass: initiating 4 asynchronous jobs
Initiating function evaluation 1
(text_book /tmp/fileG2uzVX /tmp/fileSqceY8) &
Initiating function evaluation 2
(text_book /tmp/filegFLu5j /tmp/fileeycMcv) &
Initiating function evaluation 3
(text_book /tmp/file8EI3kG /tmp/fileuY2ltR) &
Initiating function evaluation 4
(text_book /tmp/fileEZpDC2 /tmp/fileeMDVLd) &
Second pass: self-scheduling 45 remaining jobs
Waiting on completed jobs
Function evaluation 1 has completed
Initiating function evaluation 5
```

```
(text_book /tmp/file8SWrXo /tmp/filem00Y8z) &
Function evaluation 2 has completed
Initiating function evaluation 6
(text_book /tmp/file6PQ5kL /tmp/filegRydxW) &
Function evaluation 3 has completed
Initiating function evaluation 7
(text_book /tmp/filesjB8J7 /tmp/fileUpr4Wi) &
Function evaluation 4 has completed
Initiating function evaluation 8
(text_book /tmp/fileCI6Bbu /tmp/fileWSBaqF) &

<snip>

Function evaluation 49 has completed

<<<<< Iterator dace completed.
```

where it is evident that each of the 49 jobs is first queued and then a blocking synchronization is performed. This synchronization uses a simple scheduler that initiates 4 jobs and then replaces completing jobs with new ones until all 49 are complete.

The default job concurrency for asynchronous local parallelism is all that is available from the algorithm (49 in this case), which could be too many for the computational resources or their usage policies. The concurrency level specification (4 in this case) instructs the scheduler to keep 4 jobs running concurrently, which would be appropriate for, e.g., a dual-processor dual-core workstation. In this case, it is the operating system's responsibility to assign the concurrent `text_book` jobs to available processors/cores. Specifying greater concurrency than that supported by the hardware will result in additional context switching within a multitasking operating system and will generally degrade performance. Note however that, in this example, there are a total of 5 processes running, one for DAKOTA and four for the concurrent function evaluations. Since the DAKOTA process checks periodically for job completion and sleeps in between checks, it is relatively lightweight and does not require a dedicated processor.

19.2.1.5 Local evaluation scheduling options

The default behavior for asynchronous local parallelism is for DAKOTA to dispatch the next evaluation the local queue when one completes (and can optionally be specified by `local_evaluation_self_scheduling`. In some cases, the simulation code interface benefits from knowing which job number will replace a completed job. This includes some modes of application tiling with certain MPI implementations, where sending a job to the correct subset of available processors is done with relative node scheduling. The keyword `local_evaluation_static_scheduling` forces this behavior, so a completed evaluation will be replaced with one congruent module the evaluation concurrency. For example, with 7 concurrent jobs, eval number 2 will be replaced with eval number 9. Examples of this usage can be seen in `Dakota/examples/parallelism`.

19.2.2 Message Passing Parallelism

DAKOTA uses a "single program-multiple data" (SPMD) parallel programming model. It uses message-passing routines from the Message Passing Interface (MPI) standard [60], [105] to communicate data between processors. The SPMD designation simply denotes that the same DAKOTA executable is loaded on all processors during the parallel invocation. This differs from the MPMD model ("multiple program-multiple data") which would have the DAKOTA executable on one or more processors communicating directly with simulator executables on other

processors. The MPMD model has some advantages, but heterogeneous executable loads are not supported by all parallel environments. Moreover, the MPMD model requires simulation code intrusion on the same order as conversion to a subroutine, so subroutine conversion (see Section 18.2) in a direct-linked SPMD model is preferred.

19.2.2.1 Partitioning

A level of message passing parallelism can use either of two processor partitioning models:

- *Dedicated master*: a single processor is dedicated to scheduling operations and the remaining processors are split into server partitions.
- *Peer partition*: all processors are allocated to server partitions and the loss of a processor to scheduling is avoided.

These models are depicted in Figure 19.2. The peer partition is desirable since it utilizes all processors for computation; however, it requires either the use of sophisticated mechanisms for distributed scheduling or a problem for which static scheduling of concurrent work performs well (see *Scheduling* below). If neither of these characteristics is present, then use of the dedicated master partition supports a dynamic scheduling which assures that server idleness is minimized.

Figure 19.2: Communicator partitioning models.

19.2.2.2 Scheduling

The following scheduling approaches are available within a level of message passing parallelism:

- *Self-scheduling*: in the dedicated master model, the master processor manages a single processing queue and maintains a prescribed number of jobs (usually one) active on each slave. Once a slave server has completed a job and returned its results, the master assigns the next job to this slave. Thus, the slaves themselves determine the schedule through their job completion speed. This provides a simple dynamic scheduler in that heterogeneous processor speeds and/or job durations are naturally handled, provided there are sufficient instances scheduled through the servers to balance the variation.

- *Static scheduling*: if scheduling is statically determined at start-up, then no master processor is needed to direct traffic and a peer partitioning approach is applicable. If the static schedule is a good one (ideal conditions), then this approach will have superior performance. However, heterogeneity, when not known *a priori*, can very quickly degrade performance since there is no mechanism to adapt.

Message passing schedulers may be used for managing concurrent iterator executions within a strategy, concurrent evaluations within an iterator, or concurrent analyses within an evaluation. In each of these cases, the message passing scheduler is currently restricted to blocking synchronization, in that all jobs in the queue are completed before exiting the scheduler and returning the set of results to the algorithm. Nonblocking message-passing schedulers are under development for the iterator/evaluation concurrency level in support of fully asynchronous algorithms which do not contain synchronization points (e.g., `asynch_pattern_search` and `coliny_pattern_search`). Message passing is also used within a fine-grained parallel analysis code, although this does not involve the use of DAKOTA schedulers (DAKOTA may, at most, pass a communicator partition to the simulation). The "Message Passing" column in Table 19.1 summarizes these capabilities.

19.2.2.3 Message Passing Example

Revisiting the test file `dakota_dace.in`, DAKOTA will now compute the 49 orthogonal array samples using a message passing approach. In this case, a parallel launch utility is used to execute DAKOTA across multiple processors using syntax similar to the following:

```
mpirun -np 5 -machinefile machines dakota -i dakota_dace.in
```

Since the asynchronous local parallelism will not be used, the interface specification does not include the `asynchronous` keyword and would appear similar to:

```
interface,
        system
          analysis_driver = 'text_book'
```

The relevant excerpts from the DAKOTA output for a dedicated master partition and self-schedule, the default when the maximum concurrency (49) exceeds the available capacity (5), would appear similar to the following:

```
Running MPI executable in parallel on 5 processors.

------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1               5              peer/static
concurrent evaluations       4               1              ded. master/self
concurrent analyses          1               1              peer/static
multiprocessor analysis      1              N/A                N/A

Total parallelism levels =   1
------------------------------------------------------------------------------

>>>>> Running dace iterator.

------------------------------
```

```
Begin Function Evaluation    1
------------------------------
(Asynchronous job 1 added to queue)

------------------------------
Begin Function Evaluation    2
------------------------------
(Asynchronous job 2 added to queue)

<snip>

------------------------------
Begin Function Evaluation   49
------------------------------
(Asynchronous job 49 added to queue)

Blocking synchronize of 49 asynchronous evaluations
First pass: assigning 4 jobs among 4 servers
Master assigning function evaluation 1 to server 1
Master assigning function evaluation 2 to server 2
Master assigning function evaluation 3 to server 3
Master assigning function evaluation 4 to server 4
Second pass: self-scheduling 45 remaining jobs
Waiting on completed jobs
job 1 has returned from slave server 1
Master assigning function evaluation 5 to server 1
job 2 has returned from slave server 2
Master assigning function evaluation 6 to server 2
Waiting on completed jobs
job 3 has returned from slave server 3
Master assigning function evaluation 7 to server 3
job 4 has returned from slave server 4
Master assigning function evaluation 8 to server 4

<snip>

job 49 has returned from slave server 2

<<<<< Iterator dace completed.
```

where it is evident that each of the 49 jobs is first queued and then a blocking synchronization is performed. This synchronization uses a dynamic scheduler that initiates four jobs by sending a message from the master to each of the four servers and then replaces completing jobs with new ones until all 49 are complete. It is important to note that job execution local to each of the four servers is synchronous.

19.2.3 Hybrid Parallelism

The asynchronous local approaches described in Section 19.2.1 can be considered to rely on *external* scheduling mechanisms, since it is generally the operating system or some external queue/load sharing software that allocates jobs to processors. Conversely, the message-passing approaches described in Section 19.2.2 rely on *internal* scheduling mechanisms to distribute work among processors. These two approaches provide building blocks which can be combined in a variety of ways to manage parallelism at multiple levels. At one extreme, DAKOTA

can execute on a single processor and rely completely on external means to map all jobs to processors (i.e., using asynchronous local approaches). At the other extreme, DAKOTA can execute on many processors and manage all levels of parallelism, including the parallel simulations, using completely internal approaches (i.e., using message passing at all levels as in Figure 19.4). While all-internal or all-external approaches are common cases, many additional approaches exist between the two extremes in which some parallelism is managed internally and some is managed externally.

These combined approaches are referred to as *hybrid* parallelism, since the internal distribution of work based on message-passing is being combined with external allocation using asynchronous local approaches[1]. Figure 19.1 depicts the asynchronous local, message-passing, and hybrid approaches for a dedicated-master partition. Approaches (b) and (c) both use MPI message-passing to distribute work from the master to the slaves, and approaches (a) and (c) both manage asynchronous jobs local to a processor. The hybrid approach (c) can be seen to be a combination of (a) and (b) since jobs are being internally distributed to slave servers through message-passing and each slave server is managing multiple concurrent jobs using an asynchronous local approach. From a different perspective, one could consider (a) and (b) to be special cases within the range of configurations supported by (c). The hybrid approach is useful for supercomputers that maintain a service/compute node distinction and for supercomputers or networks of workstations that involve clusters of symmetric multiprocessors (SMPs). In the service/compute node case, concurrent multiprocessor simulations are launched into the compute nodes from the service node partition. While an asynchronous local approach from a single service node would be sufficient, spreading the application load by running DAKOTA in parallel across multiple service nodes results in better performance [30]. If the number of concurrent jobs to be managed in the compute partition exceeds the number of available service nodes, then hybrid parallelism is the preferred approach. In the case of a cluster of SMPs (or network of multiprocessor workstations), message-passing can be used to communicate between SMPs, and asynchronous local approaches can be used within an SMP. Hybrid parallelism can again result in improved performance, since the total number of DAKOTA MPI processes is reduced in comparison to a pure message-passing approach over all processors.

Hybrid schedulers may be used for managing concurrent evaluations within an iterator or concurrent analyses within an evaluation. In both of these cases, the scheduler is currently restricted to blocking synchronization, although as for message-passing schedulers described in Section 19.2.2.2, nonblocking schedulers are under development for the iterator/evaluation concurrency level. The "Hybrid" column in Table 19.1 summarizes these capabilities.

19.2.3.1 Hybrid Example

Revisiting the test file `dakota_dace.in`, DAKOTA will now compute the 49 orthogonal array samples using a hybrid approach. As for the message passing case, a parallel launch utility is used to execute DAKOTA across multiple processors:

```
mpirun -np 5 -machinefile machines dakota -i dakota_dace.in
```

Since the asynchronous local parallelism will also be used, the interface specification includes the `asynchronous` keyword and appears similar to

```
interface,
        system asynchronous evaluation_concurrency = 2
          analysis_driver = 'text_book'
```

[1]The term "hybrid parallelism" is often used to describe the combination of MPI message passing and OpenMP shared memory parallelism models. This can be considered to be a special case of the meaning here, as OpenMP is based on threads, which is analagous to asynchronous local usage of the direct simulation interface.

In the hybrid case, the specification of the desired concurrency level must be included, since the default is no longer all available (as it is for asynchronous local parallelism). Rather the default is to employ message passing parallelism, and hybrid parallelism is only available through the specification of asynchronous concurrency greater than one.

The relevant excerpts of the DAKOTA output for a dedicated master partition and self schedule, the default when the maximum concurrency (49) exceeds the maximum available capacity (10), would appear similar to the following:

```
Running MPI executable in parallel on 5 processors.

-------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1               5              peer/static
concurrent evaluations       4               1              ded. master/self
concurrent analyses          1               1              peer/static
multiprocessor analysis      1              N/A                N/A

Total parallelism levels =   1
-------------------------------------------------------------------------------

>>>>> Running dace iterator.

------------------------------
Begin Function Evaluation    1
------------------------------
(Asynchronous job 1 added to queue)

------------------------------
Begin Function Evaluation    2
------------------------------
(Asynchronous job 2 added to queue)

<snip>

------------------------------
Begin Function Evaluation   49
------------------------------
(Asynchronous job 49 added to queue)

Blocking synchronize of 49 asynchronous evaluations
First pass: assigning 8 jobs among 4 servers
Master assigning function evaluation 1 to server 1
Master assigning function evaluation 2 to server 2
Master assigning function evaluation 3 to server 3
Master assigning function evaluation 4 to server 4
Master assigning function evaluation 5 to server 1
Master assigning function evaluation 6 to server 2
Master assigning function evaluation 7 to server 3
Master assigning function evaluation 8 to server 4
Second pass: self-scheduling 41 remaining jobs
```

```
Waiting on completed jobs

<snip>

job 49 has returned from slave server 4

<<<<< Iterator dace completed.
```

where it is evident that each of the 49 jobs is first queued and then a blocking synchronization is performed. This synchronization uses a dynamic scheduler that initiates eight jobs by sending two messages to each of the four servers and then replaces completing jobs with new ones until all 49 are complete. It is important to note that job execution local to each of the four servers is asynchronous. If the available capacity was increased to meet or exceed the maximum concurrency (e.g., mpirun on 10 processors with `evaluation_concurrency = 5`), then a peer partition with static schedule would be selected by default.

19.3 Multilevel parallelism

Parallel computers within the Department of Energy national laboratories have exceeded a hundred trillion floating point operations per second (100 TeraFLOPS) in Linpack benchmarks and are expected to achieve PetaFLOPS speeds in the near future. This performance is achieved through the use of massively parallel (MP) processing using $O[10^3 - 10^4]$ processors. In order to harness the power of these machines for performing design, uncertainty quantification, and other systems analyses, parallel algorithms are needed which are scalable to thousands of processors.

DAKOTA supports a total of three tiers of scheduling and four levels of parallelism which, in combination, can minimize efficiency losses and achieve near linear scaling on MP computers. The four levels are:

1. concurrent iterators within a strategy (scheduling performed by DAKOTA)
2. concurrent function evaluations within each iterator (scheduling performed by DAKOTA)
3. concurrent analyses within each function evaluation (scheduling performed by DAKOTA)
4. multiprocessor analyses (work distributed by a parallel analysis code)

for which the first two are classified as algorithmic coarse-grained parallelism, the third is function evaluation coarse-grained parallelism, and the fourth is function evaluation fine-grained parallelism (see Section 19.1.1). Algorithmic fine-grained parallelism is not currently supported, although the development of large-scale parallel SAND techniques is a current research direction [8].

A particular application may support one or more of these parallelism types, and DAKOTA provides for convenient selection and combination of each of the supported levels. If multiple types of parallelism can be exploited, then the question may arise as to how the amount of parallelism at each level should be selected so as to maximize the overall parallel efficiency of the study. For performance analysis of multilevel parallelism formulations and detailed discussion of these issues, refer to [30]. In many cases, *the user may simply employ DAKOTA's automatic parallelism configuration facilities,* which implement the recommendations from the aforementioned paper.

Figure 19.3 shows typical fixed-size scaling performance using a modified version of the extended `text_book` problem (see Section 23.1). Three levels of parallelism (concurrent evaluations within an iterator, concurrent analyses within each evaluation, and multiprocessor analyses) are exercised. Despite the use of a fixed problem size and the presence of some idleness within the scheduling at multiple levels, the efficiency is still reasonably high[2].

[2]Note that overhead is reduced in these scaling studies by deactivating the evaluation cache and restart file logging.

Greater efficiencies are obtainable for scaled speedup studies (or for larger problems in fixed-size studies) and for problems optimized for minimal scheduler idleness (by, e.g., managing all concurrency in as few scheduling levels as possible). Note that speedup and efficiency are measured relative to the case of a single instance of a multiprocessor analysis, since it was desired to investigate the effectiveness of the DAKOTA schedulers independent from the efficiency of the parallel analysis.

(a) Relative speedup. (b) Relative efficiency.

Figure 19.3: Fixed-size scaling results for three levels of parallelism.

19.3.1 Asynchronous Local Parallelism

In most cases, the use of asynchronous local parallelism is the termination point for multilevel parallelism, in that any level of parallelism lower than an asynchronous local level will be serialized. The exception to this rule is reforking of forked processes for concurrent analyses within forked evaluations. In this case, a new process is created using fork for one of several concurrent evaluations; however, the new process is not replaced immediately using exec. Rather, the new process is reforked to create additional child processes for executing concurrent analyses within each concurrent evaluation process. This capability is not supported by system calls and provides one of the key advantages to using fork over system (see Section 12.3.5).

19.3.2 Message Passing Parallelism

19.3.2.1 Partitioning of levels

DAKOTA uses MPI communicators to identify groups of processors. The global `MPI_COMM_WORLD` communicator provides the total set of processors allocated to the DAKOTA run. `MPI_COMM_WORLD` can be partitioned into new intra-communicators which each define a set of processors to be used for a multiprocessor server. Each of these servers may be further partitioned to nest one level of parallelism within the next. At the lowest parallelism level, these intra-communicators can be passed into a simulation for use as the simulation's computational context, provided that the simulation has been designed, or can be modified, to be modular on a communicator (i.e., it does not assume ownership of `MPI_COMM_WORLD`). New intra-communicators are created with the `MPI_Comm_split` routine, and in order to send messages between these intra-communicators, new inter-communicators are created with calls to `MPI_Intercomm_create`. To minimize overhead, DAKOTA creates

new intra- and inter-communicators only when the parent communicator provides insufficient context for the scheduling at a particular level. In addition, multiple parallel configurations (containing a set of communicator partitions) can be allocated for use in studies with multiple iterators and models (e.g., 16 servers of 64 processors each could be used for iteration on a lower fidelity model, followed by two servers of 512 processors each for subsequent iteration on a higher fidelity model). Each of the parallel configurations are allocated at object construction time and are reported at the beginning of the DAKOTA output.

Each tier within DAKOTA's nested parallelism hierarchy can use the dedicated master and peer partition approaches described in Section 19.2.2.1. To recursively partition the subcommunicators of Figure 19.2, `COMM1/2/3` in the dedicated master or peer partition case would be further subdivided using the appropriate partitioning model for the next lower level of parallelism.

19.3.2.2 Scheduling within levels

Figure 19.4: Recursive partitioning for nested parallelism.

DAKOTA is designed to allow the freedom to configure each parallelism level with either the dedicated master partition/self-scheduling combination or the peer partition/static scheduling combination. In addition, certain external libraries may provide additional options. As an example, Figure 19.4 shows a case in which a branch and bound strategy employs peer partition/distributed scheduling at level 1, each optimizer partition employs concurrent function evaluations in a dedicated master partition/self-scheduling model at level 2, and each function evaluation partition employs concurrent multiprocessor analyses in a peer partition/static scheduling model at level 3. In this case, `MPI_COMM_WORLD` is subdivided into `optCOMM1/2/3/.../`τ_1, each `optCOMM` is further subdivided into `evalCOMM0` (master) and `evalCOMM1/2/3/.../`τ_2 (slaves), and each slave `evalCOMM` is further subdivided into `analCOMM1/2/3/.../`τ_3. Logic for selection of τ_i is discussed in [30].

19.3.3 Hybrid Parallelism

Hybrid parallelism approaches can take several forms when used in the multilevel parallel context. A conceptual boundary can be considered to exist for which all parallelism above the boundary is managed internally using message-passing and all parallelism below the boundary is managed externally using asynchronous local approaches. Hybrid parallelism approaches can then be categorized based on whether this boundary between internal and external management occurs within a parallelism level (*intra-level*) or between two parallelism levels (*inter-level*). In the intra-level case, the jobs for the parallelism level containing the boundary are scheduled using a hybrid scheduler, in which a capacity multiplier is used for the number of jobs to assign to each server. Each server is then responsible for concurrently executing its capacity of jobs using an asynchronous local approach. In the inter-level case, one level of parallelism manages its parallelism internally using a message-passing approach

and the next lower level of parallelism manages its parallelism externally using an asynchronous local approach. That is, the jobs for the higher level of parallelism are scheduled using a standard message-passing scheduler, in which a single job is assigned to each server. However, each of these jobs has multiple components, as managed by the next lower level of parallelism, and each server is responsible for executing these sub-components concurrently using an asynchronous local approach.

For example, consider a multiprocessor DAKOTA run which involves an iterator scheduling a set of concurrent function evaluations across a cluster of SMPs. A hybrid parallelism approach will be applied in which message-passing parallelism is used between SMPs and asynchronous local parallelism is used within each SMP. In the hybrid intra-level case, multiple function evaluations would be scheduled to each SMP, as dictated by the capacity of the SMPs, and each SMP would manage its own set of concurrent function evaluations using an asynchronous local approach. Any lower levels of parallelism would be serialized. In the hybrid inter-level case, the function evaluations would be scheduled one per SMP, and the analysis components within each of these evaluations would be executed concurrently using asynchronous local approaches within the SMP. Thus, the distinction can be viewed as whether the concurrent jobs on each server in Figure 19.1c reflect the same level of parallelism as that being scheduled by the master (intra-level) or one level of parallelism below that being scheduled by the master (inter-level).

19.4 Capability Summary

Table 19.1 shows a matrix of the supported job management approaches for each of the parallelism levels, with supported simulation interfaces and synchronization approaches shown in parentheses. The concurrent iterator and multiprocessor analysis parallelism levels can only be managed with message-passing approaches. In the former case, this is due to the fact that a separate process or thread for an iterator is not currently supported. The latter case reflects a finer point on the definition of external parallelism management. While a multiprocessor analysis can most certainly be launched (e.g., using `mpirun`/`yod`) from one of DAKOTA's analysis drivers, resulting in a parallel analysis external to DAKOTA (which is consistent with asynchronous local and hybrid approaches), this parallelism is not visible to DAKOTA and therefore does not qualify as parallelism that DAKOTA manages (and therefore is not included in Table 19.1). The concurrent evaluation and analysis levels can be managed either with message-passing, asynchronous local, or hybrid techniques, with the exceptions that the direct interface does not support asynchronous operations (asynchronous local or hybrid) at either of these levels and the system call interface does not support asynchronous operations (asynchronous local or hybrid) at the concurrent analysis level. The direct interface restrictions are present since multithreading in not yet supported and the system call interface restrictions result from the inability to manage concurrent analyses within a nonblocking function evaluation system call. Finally, nonblocking synchronization is only currently supported for asynchronous local parallelism at the concurrent function evaluation level. In time, message passing and hybrid parallelism approaches will also support nonblocking synchronization at this level.

19.5 Running a Parallel DAKOTA Job

Section 19.2 provides a few examples of serial and parallel execution of DAKOTA using asynchronous local, message passing, and hybrid approaches to single-level parallelism. The following sections provides a more complete discussion of the parallel execution syntax and available specification controls.

Table 19.1: Support of job management approaches within parallelism levels. Shown in parentheses are supported simulation interfaces and supported synchronization approaches.

Parallelism Level	Asynchronous Local	Message Passing	Hybrid
concurrent iterators within a strategy		**X** (blocking only)	
concurrent function evaluations within an iterator	**X** (system, fork) (blocking, nonblocking)	**X** (system, fork, direct) (blocking only)	**X** (system, fork) (blocking only)
concurrent analyses within a function evaluation	**X** (fork only) (blocking only)	**X** (system, fork, direct) (blocking only)	**X** (fork only) (blocking only)
fine-grained parallel analysis		**X**	

19.5.1 Single-processor execution

The command for running DAKOTA on a single-processor and exploiting asynchronous local parallelism is the same as for running DAKOTA on a single-processor for a serial study, e.g.:

```
dakota -i dakota.in > dakota.out
```

See Section 2.1.2 for additional information on single-processor command syntax.

19.5.2 Multiprocessor execution

Running a DAKOTA job on multiple processors requires the use of an executable loading facility such as `mpirun`, `mpiexec`, `poe`, or `yod`. On a network of workstations, the `mpirun` script is commonly used to initiate a parallel DAKOTA job, e.g.:

```
mpirun -np 12 dakota -i dakota.in > dakota.out
mpirun -machinefile machines -np 12 dakota -i dakota.in > dakota.out
```

where both examples specify the use of 12 processors, the former selecting them from a default system resources file and the latter specifying particular machines in a machine file (see [59] for details).

On a massively parallel computer such as ASCI Red, similar facilities are available from the Cougar operating system via the `yod` executable loading facility:

```
yod -sz 512 dakota -i dakota.in > dakota.out
```

In each of these cases, MPI command line arguments are used by MPI (extracted first in the call to `MPI_Init`) and DAKOTA command line arguments are used by DAKOTA (extracted second by DAKOTA's command line handler). An issue that can arise with these command line arguments is that the mpirun script distributed with MPICH has been observed to have problems with certain file path specifications (e.g., a relative path such as "`../some_file`"). These path problems are most easily resolved by using local linkage (all referenced files or soft links to these files appear in the same directory).

Finally, when running on computer resources that employ NQS/PBS batch schedulers, the single-processor `dakota` command syntax or the multiprocessor `mpirun` command syntax might be contained within an executable script file which is submitted to the batch queue. For example, on Cplant, the command

```
qsub -l size=512 run_dakota
```

could be submitted to the PBS queue for execution. On ASCI Red, the NQS syntax is similar:

```
qsub -q snl -lP 512 -lT 6:00:00 run_dakota
```

These commands allocate 512 compute nodes for the study, and execute the `run_dakota` script on a service node. If this script contains a single-processor `dakota` command, then DAKOTA will execute on a single service node from which it can launch parallel simulations into the compute nodes using analysis drivers that contain `yod` commands (any `yod` executions occurring at any level underneath the `run_dakota` script are mapped to the 512 compute node allocation). If the script submitted to `qsub` contains a multiprocessor `mpirun` command, then DAKOTA will execute across multiple service nodes so that it can spread the application load in either a message-passing or hybrid parallelism approach. Again, analysis drivers containing `yod` commands would be responsible for utilizing the 512 compute nodes. And, finally, if the script submitted to `qsub` contains a `yod` of the `dakota` executable, then DAKOTA will execute directly on the compute nodes and manage all of the parallelism internally (note that a `yod` of this type without a `qsub` would be mapped to the interactive partition, rather than to the batch partition).

Not all supercomputers employ the same model for service/compute partitions or provide the same support for tiling of concurrent multiprocessor simulations within a single NQS/PBS allocation. For this reason, templates for parallel job configuration are being catalogued within `Dakota/examples/script_interfaces` and `Dakota/examples/parallelism` (in the software distributions) that are intended to provide guidance for individual machine idiosyncrasies.

19.6 Specifying Parallelism

Given an allotment of processors, DAKOTA contains logic based on the theoretical work in [30] to automatically determine an efficient parallel configuration, consisting of partitioning and scheduling selections for each of the parallelism levels. This logic accounts for problem size, the concurrency supported by particular iterative algorithms, and any user inputs or overrides. The following points are important components of the automatic configuration logic which can be helpful in estimating the total number of processors to allocate and in selecting configuration overrides:

- If the capacity of the servers in a peer configuration is sufficient to schedule all jobs in one pass, then a peer partition and static schedule will be selected. If this capacity is not sufficient, then a dedicated-master partition and dynamic schedule will be used. These selections can be overridden with self/static scheduling request specifications for the concurrent iterator, evaluation, and analysis parallelism levels. For example, if it is known that processor speeds and job durations have little variability, then overriding the automatic configuration with a static schedule request could eliminate the unnecessary loss of a processor to scheduling.
- With the exception of the concurrent-iterator parallelism level (iterator executions tend to have high variability in duration), concurrency is pushed up. That is, available processors will be assigned to concurrency at the higher parallelism levels first. If more processors are available than needed for concurrency at a level, then the server size is increased to support concurrency in the next lower level of parallelism. This process is continued until all available processors have been assigned. These assignments can be overridden with a servers specification for the concurrent iterator, evaluation, and analysis parallelism levels and with a processors per analysis specification for the multiprocessor analysis parallelism level. For example, if it is desired to parallelize concurrent analyses within each function evaluation, then an `evaluation_servers`

= 1 override would serialize the concurrent function evaluations level and assure processor availability for concurrent analyses.

In the following sections, the user inputs and overrides are described, followed by specification examples for single and multi-processor DAKOTA executions.

19.6.1 The interface specification

Specifying parallelism within an interface can involve the use of the `asynchronous`, `evaluation_concurrency`, and `analysis_concurrency` keywords to specify concurrency local to a processor (i.e., asynchronous local parallelism). This `asynchronous` specification has dual uses:

- When running DAKOTA on a single-processor, the `asynchronous` keyword specifies the use of asynchronous invocations local to the processor (these jobs then rely on external means to be allocated to other processors). The default behavior is to simultaneously launch all function evaluations available from the iterator as well as all available analyses within each function evaluation. In some cases, the default behavior can overload a machine or violate a usage policy, resulting in the need to limit the number of concurrent jobs using the `evaluation_concurrency` and `analysis_concurrency` specifications.
- When executing DAKOTA across multiple processors and managing jobs with a message-passing scheduler, the `asynchronous` keyword specifies the use of asynchronous invocations local to each server processor, resulting in a hybrid parallelism approach (see Section 19.2.3). In this case, the default behavior is one job per server, which must be overridden with an `evaluation_concurrency` specification and/or an `analysis_concurrency` specification. When a hybrid parallelism approach is specified, the capacity of the servers (used in the automatic configuration logic) is defined as the number of servers times the number of asynchronous jobs per server.

In addition, `evaluation_servers`, `evaluation_self_scheduling`, and `evaluation_static_scheduling` keywords can be used to override the automatic parallelism configuration for concurrent function evaluations; `analysis_servers`, `analysis_self_scheduling`, and `analysis_static_scheduling` keywords can be used to override the automatic parallelism configuration for concurrent analyses; and the `processors_per_analysis` keyword can be used to override the automatic parallelism configuration for the size of multiprocessor analyses used in a direct function simulation interface. Each of these keywords appears as part of the interface commands specification in the DAKOTA Reference Manual [3].

19.6.2 The strategy specification

To specify concurrency in iterator executions, the `iterator_servers`, `iterator_self_scheduling`, and `iterator_static_scheduling` keywords are used to override the automatic parallelism configuration. See the strategy commands specification in the DAKOTA Reference Manual [3] for additional information.

19.6.3 Single-processor DAKOTA specification

Specifying a single-processor DAKOTA job that exploits parallelism through asynchronous local approaches (see Figure 19.1a) requires inclusion of the `asynchronous` keyword in the interface specification. Once the input file is defined, single-processor DAKOTA jobs are executed using the command syntax described previously in Section 19.5.1.

19.6.3.1 Example 1

For example, the following specification runs an NPSOL optimization which will perform asynchronous finite differencing:

```
method,
        npsol_sqp

variables,
        continuous_design = 5
          initial_point  0.2  0.05 0.08 0.2  0.2
          lower_bounds   0.15 0.02 0.05 0.1  0.1
          upper_bounds   2.0  2.0  2.0  2.0  2.0

interface,
        system,
          asynchronous
          analysis_drivers = 'text_book'

responses,
        num_objective_functions = 1
        num_nonlinear_inequality_constraints = 2
        numerical_gradients
          interval_type central
          method_source dakota
          fd_gradient_step_size = 1.e-4
        no_hessians
```

Note that `method_source dakota` selects DAKOTA's internal finite differencing routine so that the concurrency in finite difference offsets can be exploited. In this case, central differencing has been selected and 11 function evaluations (one at the current point plus two offsets in each of five variables) can be performed simultaneously for each NPSOL response request. These 11 evaluations will be launched with system calls in the background and presumably assigned to additional processors through the operating system of a multiprocessor compute server or other comparable method. The concurrency specification may be included if it is necessary to limit the maximum number of simultaneous evaluations. For example, if a maximum of six compute processors were available, the command

```
evaluation_concurrency = 6
```

could be added to the `asynchronous` specification within the `interface` keyword from the preceding example.

19.6.3.2 Example 2

If, in addition, multiple analyses can be executed concurrently within a function evaluation (e.g., from multiple load cases or disciplinary analyses that must be evaluated to compute the response data set), then an input specification similar to the following could be used:

```
method,
        npsol_sqp

variables,
```

```
        continuous_design = 5
          initial_point  0.2  0.05 0.08 0.2  0.2
          lower_bounds   0.15 0.02 0.05 0.1  0.1
          upper_bounds   2.0  2.0  2.0  2.0  2.0

interface,
        fork
          asynchronous
            evaluation_concurrency = 6
            analysis_concurrency = 3
          analysis_drivers = 'text_book1' 'text_book2' 'text_book3'

responses,
        num_objective_functions = 1
        num_nonlinear_inequality_constraints = 2
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-4
        no_hessians
```

In this case, the default concurrency with just an `asynchronous` specification would be all 11 function evaluations and all 3 analyses, which can be limited by the `evaluation_concurrency` and `analysis_concurrency` specifications. The input file above limits the function evaluation concurrency, but not the analysis concurrency (a specification of 3 is the default in this case and could be omitted). Changing the input to `evaluation_concurrency = 1` would serialize the function evaluations, and changing the input to `analysis_concurrency = 1` would serialize the analyses.

19.6.4 Multiprocessor DAKOTA specification

In multiprocessor executions, server evaluations are synchronous (Figure 19.1b) by default and the `asynchronous` keyword is only used if a hybrid parallelism approach (Figure 19.1c) is desired. Multiprocessor DAKOTA jobs are executed using the command syntax described previously in Section 19.5.2.

19.6.4.1 Example 3

To run Example 1 using a message-passing approach, the `asynchronous` keyword would be removed (since the servers will execute their evaluations synchronously), resulting in the following interface specification:

```
interface,
        system,
          analysis_drivers = 'text_book'
```

Running DAKOTA on 4 processors (syntax: `mpirun -np 4 dakota -i dakota.in`) would result in the following parallel configuration report from the DAKOTA output:

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                    num_servers    procs_per_server    partition/schedule
```

```
-----                    -----------    ----------------    ------------------
concurrent iterators         1               4              peer/static
concurrent evaluations       3               1              ded. master/self
concurrent analyses          1               1              peer/static
multiprocessor analysis      1              N/A                N/A

Total parallelism levels =   1
-------------------------------------------------------------------------------
```

The dedicated master partition and self-scheduling algorithm are automatically selected for the concurrent evaluations parallelism level since the number of function evaluations (11) is greater than the maximum capacity of the servers (4). Since one of the processors is dedicated to being the master, only 3 processors are available for computation and the 11 evaluations can be completed in approximately 4 passes through the servers. If it is known that there is little variability in evaluation duration, then this logic could be overridden to use a static schedule through use of the `evaluation_static_scheduling` specification:

```
interface,
        system,
          evaluation_static_scheduling
          analysis_drivers = 'text_book'
```

Running DAKOTA again on 4 processors (syntax: `mpirun -np 4 dakota -i dakota.in`) would now result in this parallel configuration report:

```
-------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                    num_servers    procs_per_server    partition/schedule
-----                    -----------    ----------------    ------------------
concurrent iterators         1               4              peer/static
concurrent evaluations       4               1              peer/static
concurrent analyses          1               1              peer/static
multiprocessor analysis      1              N/A                N/A

Total parallelism levels =   1
-------------------------------------------------------------------------------
```

Now the 11 jobs will be statically distributed among 4 peer servers, since the processor previously dedicated to scheduling has been converted to a compute server. This could be more efficient if the evaluation durations are sufficiently similar, but there is no mechanism to adapt to heterogeneity in processor speeds or simulation expense.

As a related example, consider the case where each of the workstations used in the parallel execution has multiple processors. In this case, a hybrid parallelism approach which combines message-passing parallelism with asynchronous local parallelism (see Figure 19.1c) would be a good choice. To specify hybrid parallelism, one uses the same `asynchronous` specification as was used for the single-processor examples, e.g.:

```
interface,
         system
           asynchronous evaluation_concurrency = 3
           analysis_drivers = 'text_book'
```

With 3 function evaluations concurrent on each server, the capacity of a 4 processor DAKOTA execution (syntax: `mpirun -np 4 dakota -i dakota.in`) has increased to 12 evaluations. Since all 11 jobs can now be scheduled in a single pass, a static schedule is automatically selected (without any override request):

```
-------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1               4              peer/static
concurrent evaluations       4               1              peer/static
concurrent analyses          1               1              peer/static
multiprocessor analysis      1              N/A                N/A

Total parallelism levels =   1
-------------------------------------------------------------------------------
```

19.6.4.2 Example 4

To run Example 2 using a message-passing approach, the `asynchronous` specification is again removed:

```
interface,
         fork
           analysis_drivers = 'text_book1' 'text_book2' 'text_book3'
```

Running this example on 6 processors (syntax: `mpirun -np 6 dakota -i dakota.in`) would result in the following parallel configuration report:

```
-------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1               6              peer/static
concurrent evaluations       5               1              ded. master/self
concurrent analyses          1               1              peer/static
multiprocessor analysis      1              N/A                N/A

Total parallelism levels =   1
-------------------------------------------------------------------------------
```

in which all of the processors have been assigned to support evaluation concurrency due to the "push up" automatic configuration logic. Note that the default configuration could be a poor choice in this case, since 11 jobs scheduled through 5 servers will likely have significant idleness towards the end of the scheduling. To assign some of the available processors to the concurrent analysis level, the following input could be used:

```
interface,
         fork
           analysis_drivers = 'text_book1' 'text_book2' 'text_book3'
           evaluation_static_scheduling
           evaluation_servers = 2
```

which results in the following 2-level parallel configuration:

```
-------------------------------------------------------------------------------
DAKOTA parallel configuration:
```

```
Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1               6              peer/static
concurrent evaluations       2               3              peer/static
concurrent analyses          3               1              peer/static
multiprocessor analysis      1              N/A                 N/A

Total parallelism levels =   2
-------------------------------------------------------------------------------
```

The six processors available have been split into two evaluation servers of three processors each, where the three processors in each evaluation server manage the three analyses, one per processor.

Next, consider the following 3-level parallel case, in which `text_book1`, `text_book2`, and `text_book3` from the previous examples now execute on two processors each. In this case, the `processors_per_analysis` keyword is added and the `fork` interface is changed to a `direct` interface since the fine-grained parallelism of the three simulations is managed internally:

```
interface,
         direct
           analysis_drivers = 'text_book1' 'text_book2' 'text_book3'
           evaluation_static_scheduling
           evaluation_servers = 2
           processors_per_analysis = 2
```

This results in the following parallel configuration for a 12 processor DAKOTA run
(syntax: `mpirun -np 12 dakota -i dakota.in`):

```
-------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1              12              peer/static
concurrent evaluations       2               6              peer/static
concurrent analyses          3               2              peer/static
multiprocessor analysis      2              N/A                 N/A

Total parallelism levels =   3
-------------------------------------------------------------------------------
```

An important point to recognize is that, since each of the parallel configuration inputs has been tied to the interface specification up to this point, these parallel configurations can be reallocated for each interface in a multi-iterator/multi-model strategy. For example, a DAKOTA execution on 40 processors might involve the following two interface specifications:

```
interface,
        direct,
          id_interface = 'COARSE'
          analysis_driver = 'sim1'
          processors_per_analysis = 5
```

```
interface,
        direct,
          id_interface = 'FINE'
          analysis_driver = 'sim2'
          processors_per_analysis = 10
```

for which the coarse model would employ 8 servers of 5 processors each and the fine model would employ 4 servers of 10 processors each.

Next, consider the following 4-level parallel case that employs the Pareto set optimization strategy. In this case, `iterator_servers` and `iterator_static_scheduling` requests are included in the strategy specification:

```
strategy,
         pareto_set
           iterator_servers = 2
           iterator_static_scheduling
           opt_method_pointer = 'NLP'
           random_weight_sets = 4
```

Adding this strategy specification to the input file from the previous 12 processor example results in the following parallel configuration for a 24 processor DAKOTA run
(syntax: `mpirun -np 24 dakota -i dakota.in`):

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         2               12               peer/static
concurrent evaluations       2                6               peer/static
concurrent analyses          3                2               peer/static
multiprocessor analysis      2               N/A                 N/A

Total parallelism levels =   4
-----------------------------------------------------------------------------
```

19.6.4.3 Example 5

As a final example, consider a multi-start optimization conducted on 384 processors of ASCI Red. A job of this size must be submitted to the batch queue, using syntax similar to:

```
qsub -q snl -lP 384 -lT 6:00:00 run_dakota
```

where the `run_dakota` script appears as

```
#!/bin/sh
cd /scratch/<some_workdir>
yod -sz 384 dakota -i dakota.in > dakota.out
```

and the strategy and interface specifications from the `dakota.in` input file appear as

```
strategy,
        multi_start
          method_pointer = 'CPS'
          iterator_servers = 8
          random_starts = 8

interface,
        direct,
          analysis_drivers = 'text_book1' 'text_book2' 'text_book3'
          evaluation_servers = 8
          evaluation_static_scheduling
          processors_per_analysis = 2
```

The resulting parallel configuration is reported as

```
------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators          8               48               peer/static
concurrent evaluations        8                6               peer/static
concurrent analyses           3                2               peer/static
multiprocessor analysis       2               N/A                 N/A

Total parallelism levels =   4
------------------------------------------------------------------------------
```

Since the concurrency at each of the nested levels has a multiplicative effect on the number of processors that can be utilized, it is easy to see how large numbers of processors can be put to effective use in reducing the time to reach a solution, even when, as in this example, the concurrency per level is relatively low.

19.7 Application Parallelism Use Cases

This section describes several common use cases for running DAKOTA on parallel computing clusters with various combinations of DAKOTA and application parallelism. In three of the four cases addressed, the application launched by DAKOTA is assumed MPI-enabled and run as an independent parallel process. For demonstration purposes, the following characteristics are shared among the usage examples:

- DAKOTA performs a vector parameter study requiring 20 model evaluations (application runs).
- For each evaluation, DAKOTA uses a fork simulation interface to call a shell script `text_book_par_driver` or `text_book_driver` to launch the application. This script is a stand-in for a typical DAKOTA-application black box interface (as described in Chapter 18.1), and includes mock application input preparation, execution, and postprocessing to return necessary metrics to DAKOTA.
- The application executed is a modified version of the text book example driver, `text_book_simple_par`, capable of parallel execution, or the standard `text_book` driver for serial demonstration.

The combinations of DAKOTA and application parallelism are summarized in Table 19.2. In each case, M denotes the total number of processors allocated and N denotes the number of processors used by a single application

Table 19.2: Cases for DAKOTA and application-level parallelism with M available processors and each application job requiring N processors. Cases 1–3 assume that DAKOTA and any application runs will execute wholly within a single scheduled job, whereas Case 4 is relevant when analysis jobs must be individually submitted to a scheduler.

Case	DAKOTA	Application	Notes
1	parallel	serial	$M-1$ (or M) simultaneous application instances each $N=1$ processor
2	serial	parallel	1 simultaneous application instance on N processors
3	serial	parallel	$\approx (M-1)/N$ or $\approx M/N$ simultaneous N processor jobs
4	serial	parallel	submit *expensive* N processor application jobs to a scheduler (e.g., qsub)

analysis. For most scenarios, Cases 1–3, where DAKOTA and the application jobs run within a single cluster processor allocation (queued job), are preferred. However for particularly long-running or large jobs, or platforms that not supporting the first scheduling modes, Case 4 may be most appropriate.

Relevant example files for each case are included in directories `Dakota/examples/parallelism/Case*` with the DAKOTA distribution. These typically include a PBS or SLURM job submission script to launch the DAKOTA study, a DAKOTA input file, and a driver script.

19.7.1 Case 1: Multiple serial analysis jobs

In this case, DAKOTA will launch multiple simultaneous single processor application runs (massively serial analysis code execution, an embarrassingly parallel model). DAKOTA is run in parallel, making this example an elaboration of the message-passing single-level parallel mode described in Section 19.2. Specifically in this example, DAKOTA is run in parallel with $M = 6$ processors (`pbs_submission`):

```
mpiexec -n 6 dakota dakota_pstudy.in
```

and its default master-slave schedule will launch $M-1$ simultaneous analysis jobs, and as each job completes, another will be launched, until all jobs are complete. Several options are possible in this case:

- If the possible DAKOTA application concurrency equals M, DAKOTA will use a peer-to-peer scheduler, and run the M jobs concurrently. When the possible concurrency is greater than M, DAKOTA will by default launch $M-1$ jobs with a master-slave model. Specifying `static_schedule` in the DAKOTA input, will override the default master-slave scheduler and DAKOTA will launch M jobs, but jobs will be launched blocking, so all M will complete, then another M will be scheduled.
- If the analysis is extremely inexpensive, performance may be improved by launching multiple evaluation jobs local to each DAKOTA MPI process, specifying

  ```
  asynchronous evaluation_concurrency = [2 or more]
  ```

- It is also possible to launch only one DAKOTA process per node, and then use either asynchronous local as above, or launch the application in parallel using only the local processors (shared-memory MPI parallelism):

  ```
  mpiexec -pernode -n 3 dakota dakota_pstudy.in
  ```

Caveat: This example assumes the application is capable of serial execution (does not call MPI_Init), which on some platforms or MPI implementations is not equivalent to `mpiexec -n 1`. Some MPI/scheduler combinations will not permit another MPI process to run on a resource assigned to the DAKOTA processes.

19.7.2 Case 2: One simultaneous parallel analysis job

This case is relevant for multi-processor analysis jobs, typically where the analysis is expensive (i.e., is long-running or sufficient processors are not available to run more than one simultaneous analysis). Note that for extremely long-running parallel jobs, Case 4 below may be more appropriate.

In this case, DAKOTA runs in serial

```
dakota dakota_pstudy.in
```

and the driver script launches the application with `mpiexec -n K`, where $K \leq M$, to launch the application code within the processor allocation:

```
mpiexec -n 6 text_book_par application.in application.out
```

19.7.3 Case 3: Multiple simultaneous parallel analysis jobs

In this "job tiling" case, a single scheduled processor allocation is partitioned to run $\approx (M-1)/N$ or $\approx M/N$ parallel application jobs, each requiring N processors. We describe two current ways to accomplish this (though other solutions exist): use option (a) if the application will work correctly in an MPICH/MVAPICH environment and option (b) otherwise.

19.7.3.1 Mpiexec server mode

Mpiexec (http://www.osc.edu/ pw/mpiexec/) works in concert with MPICH implementations, extending mpirun to run jobs in a PBS environment with additional features. It offers a background server option which can be used to tile multiple MPI jobs within a single parallel resource allocation. (Note that with MPICH, there is a difference between `mpirun` and `mpiexec`, unlike with OpenMPI, where both are typically aliases for `orterun`.) See the example in `Case3_MPICH`.

In this case, an `mpiexec` server process is started and backgrounded to service application requests for processors; DAKOTA runs in serial (`pbs_submission`):

```
mpiexec -server &

dakota dakota_pstudy.in
```

and asynchronously launches $M/N = 3$ evaluations (`dakota_pstudy.in`):

```
interface, application fork, asynchronous evaluation_concurrency = 3
  analysis_driver = 'text_book_par_driver'
```

The simulator script calls `mpiexec -n 2` to run the analysis in parallel and the mpiexec server assigns a subset of the available processors to the particular MPI task (`text_book_par`):

```
mpiexec -n 2 text_book_simple_par application.in application.out
```

An error will result if more application tasks are launched than the processor allocation permits. An error may result if the application does not exit cleanly. At present similar capability is not supported by OpenMPI, although a daemon mode similar to Mpiexec has been proposed.

19.7.3.2 Relative node scheduling

This Case 3 variant uses OpenMPI 1.3.3 or newer or SLURM srun relative node scheduling capability. It leverages DAKOTA's `local_evaluation_static_scheduling` option together with integer arithmetic to schedule each evaluation on the right subset of the processor allocation. For examples, see `Case3_MPICH` (srun variant) and `Case3_OpenMPI`. Similar approaches work with some AIX/POE installations as well.

19.7.3.3 Machinefile management

This Case 3 variant applies when the application must be compiled with OpenMPI or another MPI implementation that does not support a server mode for job tiling, but does support the use of machine files specifying the resources on which to run the application job. A set of scripts are used to manage the partitioning of the M processor allocation among N analysis jobs, each with a machines file consisting of a unique subset of the assigned resources. Note that this will not work with early OpenMPI versions with some resource managers (e.g., OpenMPI 1.2 with Torque), where machinefiles, even if a proper subset of `$PBS_NODEFILE`, are ignored. This will however work with OpenMPI 1.3 and newer. See the example in `Case3_MachinefileMgmt`.

In this case the `pbs_submission` script defines variables specifying how to create a separate node file for each job and sets up a set of nodefiles for use by each evaluation. Similarly to Case 3a, DAKOTA runs in serial and uses asynchronous evaluation concurrency to launch the jobs. The `text_book_par_driver` now contains logic to lock a node file for the application run and return it when complete. As each job completes, the next is scheduled.

19.7.4 Case 4: Parallel analysis jobs submitted to a queue

This case describes running DAKOTA to submit parallel jobs to a batch queue. This option is likely only useful when the cost of an individual analysis evaluation is high (such that the job requires far too many processors or hours to run all the evaluations) and there is no feedback to DAKOTA required to generate subsequent evaluation points. So this scenario is likely more relevant for sensitivity analysis and uncertainty quantification than optimization.

In the first pass, DAKOTA runs (likely interactively) in serial on a login node or other node capable of job submission:

```
dakota dakota_pstudy.in
```

For each evaluation, the simulator script (`text_book_par_driver`) will generate a `pbs_submission` script and submit it to the scheduler. Dummy results are returned to DAKOTA which will exit when all jobs have been scheduled.

In the second pass, when analysis is complete, the analysis driver is changed to `post_process` and DAKOTA is executed on a login node to collect the results of the study.

Chapter 20

DAKOTA Usage Guidelines

20.1 Problem Exploration

The first objective in an analysis is to characterize the problem so that appropriate algorithms can be chosen. In the case of optimization, typical questions that should be addressed include: Are the design variables continuous, discrete, or mixed? Is the problem constrained or unconstrained? How expensive are the response functions to evaluate? Will the response functions behave smoothly as the design variables change or will there be nonsmoothness and/or discontinuities? Are the response functions likely to be multimodal, such that global optimization may be warranted? Is analytic gradient data available, and if not, can I calculate gradients accurately and cheaply? Additional questions that are pertinent for characterization of uncertainty quantification problems include: Can I accurately model the probabilistic distributions of my uncertain variables? Are the response functions relatively linear? Am I interested in a full random process characterization of the response functions, or just statistical results?

If there is not sufficient information from the problem description to answer these questions, then additional problem characterization activities may be warranted. One particularly useful characterization activity that DAKOTA enables is parameter space exploration through the use of parameter studies and design of experiments methods. The parameter space can be systematically interrogated to create sufficient information to evaluate the trends in the response functions and to determine if these trends are noisy or smooth, unimodal or multimodal, relatively linear or highly nonlinear, etc. In addition, the parameter studies may reveal that one or more of the parameters do not significantly affect the results and can be removed from the problem formulation. This can yield a potentially large savings in computational expense for the subsequent studies. Refer to Chapters 4 and 5 for additional information on parameter studies and design of experiments methods.

20.2 Optimization Method Selection

In selecting an optimization method, important considerations include the type of variables in the problem (continuous, discrete, mixed), whether a global search is needed or a local search is sufficient, and the required constraint support (unconstrained, bound constrained, or generally constrained). Less obvious, but equally important, considerations include the efficiency of convergence to an optimum (i.e., convergence rate) and the robustness of the method in the presence of challenging design space features (e.g., nonsmoothness).

Gradient-based optimization methods are highly efficient, with the best convergence rates of all of the optimization

methods. If analytic gradient and Hessian information can be provided by an application code, a full Newton method will provide quadratic convergence rates near the solution. More commonly, only gradient information is available and a quasi-Newton method is chosen in which the Hessian information is approximated from an accumulation of gradient data. In this case, superlinear convergence rates can be obtained. These characteristics make gradient-based optimization methods the methods of choice when the problem is smooth, unimodal, and well-behaved. However, when the problem exhibits nonsmooth, discontinuous, or multimodal behavior, these methods can also be the least robust since inaccurate gradients will lead to bad search directions, failed line searches, and early termination, and the presence of multiple minima will be missed.

Thus, for gradient-based optimization, a critical factor is the gradient accuracy. Analytic gradients are ideal, but are often unavailable. For many engineering applications, a finite difference method will be used by the optimization algorithm to estimate gradient values. DAKOTA allows the user to select the step size for these calculations, as well as choose between forward-difference and central-difference algorithms. The finite difference step size should be selected as small as possible, to allow for local accuracy and convergence, but not so small that the steps are "in the noise." This requires an assessment of the local smoothness of the response functions using, for example, a parameter study method. Central differencing, in general, will produce more reliable gradients than forward differencing, but at roughly twice the expense.

Nongradient-based methods exhibit much slower convergence rates for finding an optimum, and as a result, tend to be much more computationally demanding than gradient-based methods. Nongradient local optimization methods, such as pattern search algorithms, often require from several hundred to a thousand or more function evaluations, depending on the number of variables, and nongradient global optimization methods such as genetic algorithms may require from thousands to tens-of-thousands of function evaluations. Clearly, for nongradient optimization studies, the computational cost of the function evaluation must be relatively small in order to obtain an optimal solution in a reasonable amount of time. In addition, nonlinear constraint support in nongradient methods is an open area of research and, while supported by many nongradient methods in DAKOTA, is not as refined as constraint support in gradient-based methods. However, nongradient methods can be more robust and more inherently parallel than gradient-based approaches. They can be applied in situations were gradient calculations are too expensive or unreliable. In addition, some nongradient-based methods can be used for global optimization which gradient-based techniques, by themselves, cannot. For these reasons, nongradient-based methods deserve consideration when the problem may be nonsmooth, multimodal, or poorly behaved.

Approaches that seek to improve the effectiveness or efficiency of optimizers and least squares methods through the use of surrogate models include the surrogate-based local, surrogate-based global, and efficient global methods. Chapter 9 provides further information on these approaches. The surrogate-based local approach (see Section 9.2) brings the efficiency of gradient-based optimization/least squares methods to nonsmooth or poorly behaved problems by smoothing noisy or discontinuous response results with a data fit surrogate model (e.g., a quadratic polynomial) and then minimizing on the smooth surrogate using efficient gradient-based techniques. The surrogate-based global approach (see Section 9.3) similarly employs optimizers/least squares methods with surrogate models, but rather than localizing through the use of trust regions, seeks global solutions using global methods. And the efficient global approach (see Section 9.4) uses the specific combination of Gaussian process surrogate models in combination with the DIRECT global optimizer. Similar to these surrogate-based approaches, the hybrid and multistart optimization strategies seek to bring the efficiency of gradient-based optimization methods to global optimization problems. In the former case, a global optimization method can be used for a few cycles to locate promising regions and then local gradient-based optimization is used to efficiently converge on one or more optima. In the latter case, a stratification technique is used to disperse a series of local gradient-based optimization runs through parameter space. Without surrogate data smoothing, however, these strategies are best for smooth multimodal problems. Section 16.2 and Section 16.3 provide more information on these approaches.

Table 20.1 provides a convenient reference for choosing an optimization method or strategy to match the characteristics of the user's problem, where blank fields inherit the value from above. With respect to constraint support,

it should be understood that the methods with more advanced constraint support are also applicable to the lower constraint support levels; they are listed only at their highest level of constraint support for brevity.

20.3 UQ Method Selection

The need for computationally efficient methods is further amplified in the case of the quantification of uncertainty in computational simulations. Sampling-based methods are the most robust uncertainty techniques available, are applicable to almost all simulations, and possess rigorous error bounds; consequently, they should be used whenever the function is relatively inexpensive to compute. However, in the case of terascale computational simulations, the number of function evaluations required by traditional techniques such as Monte Carlo and Latin hypercube sampling (LHS) quickly becomes prohibitive, especially if tail statistics are needed. Additional sampling options include quasi-Monte Carlo (QMC) sampling and importance sampling (IS), and incremental sampling may also be used to incrementally add samples to an existing sample set.

Alternatively, one can apply the traditional sampling techniques to a surrogate function approximating the expensive computational simulation (see Section 17.3). However, if this approach is selected, the user should be aware that it is very difficult to assess the accuracy of the results obtained. Unlike the case of surrogate-based local minimization (see Section 9.2), there is no simple pointwise calculation to verify the accuracy of the approximate results. This is due to the functional nature of uncertainty quantification, i.e. the accuracy of the surrogate over the entire parameter space needs to be considered, not just around a candidate optimum as in the case of surrogate-based local. This issue especially manifests itself when trying to estimate low probability events such as the catastrophic failure of a system.

Another class of UQ methods known as local reliability methods (e.g., MV, AMV/AMV^2, AMV+/AMV^2+, TANA, and FORM/SORM) are more computationally efficient in general than the sampling methods and are effective when applied to reasonably well-behaved response functions; i.e., functions that are smooth, unimodal, and linear or mildly nonlinear. They can be used to provide qualitative sensitivity information concerning which uncertain variables are important (with relatively few function evaluations), or compute full cumulative or complementary cumulative response functions (with additional computational effort). Since they rely on gradient calculations to compute local optima (most probable points of failure), they scale well for increasing numbers of random variables, but issues with nonsmooth, discontinuous, and multimodal response functions are relevant concerns. In addition, even if there is a single MPP and it is calculated accurately, first-order and second-order integrations may fail to accurately capture the shape of the failure domain. In these cases, adaptive importance sampling around the MPP can be helpful. Overall, local reliability methods should be used with some care and their accuracy should be verified whenever possible.

An effective alternative to local reliability analysis when confronted with nonsmooth, multimodal, and/or highly nonlinear response functions is efficient global reliability analysis (EGRA). This technique employs Gaussian process global surrogate models to accurately resolve the failure domain and then employs multimodal adaptive importance sampling to resolve the probabilities. For relatively low dimensional problems (i.e, on the order of 10 variables), this method displays the efficiency of local reliability analysis with the accuracy of exhaustive sampling. While extremely promising, this method is still relatively new and is the subject of ongoing refinements as we deploy it to additional applications.

The next class of UQ methods available in DAKOTA is comprised of stochastic expansion methods (polynomial chaos and stochastic collocation), which are general purpose techniques provided that the response functions possess finite second order moments. Further, these methods approximate the full random process/field and capture the underlying functional relationship between a key response metric and its random variables, rather than just approximating statistics such as mean and standard deviation. This class of methods parallels traditional variational methods in mechanics; in that vein, efforts are underway to compute rigorous error bounds of the approximations

Table 20.1: Guidelines for optimization and nonlinear least squares method selection.

Variable Type	Function Surface	Solution Type	Constraints	Applicable Methods
continuous	smooth	local opt	none	optpp_cg
			bounds	dot_bfgs, dot_frcg, conmin_frcg
			general	npsol_sqp, nlpql_sqp, dot_mmfd, dot_slp, dot_sqp, conmin_mfd, optpp_newton, optpp_q_newton, optpp_fd_newton
		local least sq	bounds	nl2sol
			general	nlssol_sqp, optpp_g_newton
		local multiobjective	general	weighted sums (one soln), pareto_set strategy (multiple solns)
		global opt	general	hybrid strategy, multi_start strategy
		global least sq	general	hybrid strategy, multi_start strategy
	nonsmooth	local opt	bounds	optpp_pds
			general	asynch_pattern_search, coliny_cobyla, coliny_pattern_search, coliny_solis_wets
		local/global opt	general	surrogate_based_local
		local/global least sq	general	surrogate_based_local
		global opt	bounds	ncsu_direct
			general	coliny_ea, coliny_direct, efficient_global, soga, surrogate_based_global
		global least sq	general	efficient_global, surrogate_based_global
		global multiobjective	general	moga (multiple solns)
discrete categorical	n/a	global opt	general	soga, coliny_ea
		global multiobjective	general	moga (multiple solns)
discrete noncategorical	n/a	local opt	general	branch_and_bound strategy
mixed categorical	nonsmooth	global opt	general	soga, coliny_ea
		global multiobjective	general	moga (multiple solns)
mixed noncategorical	smooth	local opt	general	branch_and_bound strategy

Table 20.2: Guidelines for UQ method selection.

Method Classification	Desired Problem Characteristics	Applicable Methods
Sampling	nonsmooth, multimodal response functions; response evaluations are relatively inexpensive	sampling (Monte Carlo or LHS)
Local reliability	smooth, unimodal response functions; larger sets of random variables	local_reliability (MV, AMV/AMV^2, AMV+/AMV^2+, TANA, FORM/SORM)
Global reliability	nonsmooth, multimodal response functions; low dimensional	global_reliability
Stochastic expansions	nonsmooth, multimodal response functions; low dimensional; capture of functional form useful for subsequent analyses	polynomial_chaos, stoch_collocation
Epistemic	uncertainties are poorly characterized	interval: local_interval_est, global_interval_est, sampling; BPA: local_evidence, global_evidence
Mixed UQ	some uncertainties are poorly characterized	nested UQ (second-order probability, mixed evidence) with epistemic outer loop and aleatory inner loop, sampling

produced by the methods. Another strength of these methods is their potential use in a multiphysics environment as a means to propagate the uncertainty through a series of simulations while retaining as much information as possible at each stage of the analysis. The current challenge in the development of these methods, as for other global surrogate-based methods, is effective scaling for large numbers of random variables. Recent advances in adaptive sparse grid methods address some of the scaling issues for stochastic expansion.

The final class of UQ methods available in DAKOTA are focused on epistemic uncertainties, or uncertainties resulting from a lack of knowledge. In these problems, the assignment of input probability distributions when data is sparse can be somewhat suspect. One approach to handling epistemic uncertainties is Dempster-Shafer theory of evidence (DAKOTA methods `local_evidence` and `global_evidence`). Another method is pure interval analysis (`local_interval_est` and `global_interval_est`), where intervals on inputs are mapped to intervals on outputs using optimization methods

For problems with a mixture of epistemic and aleatoric uncertainties, it is desirable to segregate the two uncertainty types within a nested analysis, allowing stronger probabilistic inferences for the portion of the problem where they are appropriate. In this nested approach, an outer epistemic level selects realizations of epistemic parameters (augmented variables) and/or realizations of random variable distribution parameters (inserted variables) from intervals. These realizations define the probability distributions for an inner aleatoric level performing probabilistic analyses. In the case where the outer loop is an interval propagation approach (`local_interval_est` or `global_interval_est`), the nested approach is known as second-order probability (see also Section 10.5) and the study generates a family of CDF/CCDF respresentations known as a "horse tail" plot. In the case where the outer loop is an evidence-based approach (`local_evidence` or `global_evidence`), the approach generates epistemic belief and plausibility bounds on aleatory statistics.

The recommendations for UQ methods are summarized in Table 20.2.

Table 20.3: Guidelines for selection of parameter study, DOE, DACE, and sampling methods.

Method Classification	Applications	Applicable Methods
parameter study	sensitivity analysis, directed parameter space investigations	centered_parameter_study, list_parameter_study, multidim_parameter_study, vector_parameter_study
classical design of experiments	physical experiments (parameters are uniformly distributed)	dace (box_behnken, central_composite)
design of computer experiments	variance analysis, space filling designs (parameters are uniformly distributed)	dace (grid, random, oas, lhs, oa_lhs), fsu_quasi_mc (halton, hammersley), fsu_cvt, psuade_moat
sampling	space filling designs (parameters have general probability distributions)	sampling (Monte Carlo or LHS) with all_variables flag

20.4 Parameter Study/DOE/DACE/Sampling Method Selection

Parameter studies, classical design of experiments (DOE), design/analysis of computer experiments (DACE), and sampling methods share the purpose of exploring the parameter space. If directed studies with a defined structure are desired, then parameter study methods (see Chapter 4) are recommended. For example, a quick assessment of the smoothness of a response function is best addressed with a vector or centered parameter study. Also, performing local sensitivity analysis is best addressed with these methods. If, however, a global space-filling set of samples is desired, then the DOE, DACE, and sampling methods are recommended (see Chapter 5). These techniques are useful for scatter plot and variance analysis as well as surrogate model construction. The distinction between DOE and DACE methods is that the former are intended for physical experiments containing an element of nonrepeatability (and therefore tend to place samples at the extreme parameter vertices), whereas the latter are intended for repeatable computer experiments and are more space-filling in nature. The distinction between DOE/DACE and sampling is drawn based on the distributions of the parameters. DOE/DACE methods typically assume uniform distributions, whereas the sampling approaches in DAKOTA support a broad range of probability distributions. To use `sampling` in a design of experiments mode (as opposed to an uncertainty quantification mode), the `all_variables` flag should be included in the method specification of the DAKOTA input file.

These method selection recommendations are summarized in Table 20.3.

20.5 Surrogate Model Selection

Surrogate models provide an approximation to an original, high fidelity ”truth” model. These are useful both directly, to interpolate or extrapolate data without running the possibly expensive simulator, and as a way to reduce the cost of optimization (see Chapter 9 for more details) and uncertainty quantification (see Chapter 6). This section is intended to provide some rudimentary advice regarding when to choose which model.

Recommendations:

- For Surrogate Based Local Optimization, i.e. the `surrogate_based_local` method, with a trust region, either `surrogate local taylor_series` or `surrogate multipoint tana` will probably work best. If for some reason you wish or need to use a global surrogate (not recommended) then the best of these options is likely to be either `surrogate global gaussian_process surfpack` or `surrogate global moving_least_squares`.
- For Efficient Global Optimization (EGO), i.e. the `efficient_global` method, the default `gaussian_process surfpack` is likely to find a more optimal value and/or use fewer true function evaluations than the alternative, `gaussian_process dakota`. However, the `surfpack` version will likely take more time to build than the `dakota` version. Note that currently the `use_derivatives` keyword is not recommended for use with EGO based methods.
- For EGO based global interval estimation (EGIE), i.e. the `global_interval_est ego` method, the default `gaussian_process surfpack` will likely work better than the alternative `gaussian_process dakota`.
- For Efficient Global Reliability Analysis (EGRA), i.e. the `global_reliability` method the `surfpack` and `dakota` versions of the gaussian process tend to give similar answers with the `dakota` version tending to use fewer true function evaluations. Since this is based on EGO, it is likely that the default `surfpack` version is more accurate, although this has not been rigorously demonstrated.
- For EGO based Dempster-Shafer Theory of Evidence, i.e. the `global_evidence ego` method, the default `gaussian_process surfpack` will often use significantly fewer true function evaluations than the alternative `gaussian_process dakota`.
- When using a global surrogate to extrapolate, either the `gaussian_process surfpack` or `polynomial quadratic` or `polynomial cubic` is recommended.
- When there is over roughly two or three thousand data points and you wish to interpolate (or approximately interpolate) then a Taylor series, Radial Basis Function Network, or Moving Least Squares fit is recommended. The only reason that the `gaussian_process surfpack` model is not recommended is that it can take a considerable amount of time to construct when the number of data points is very large. Use of the third party MARS package included in DAKOTA is generally discouraged.
- In other situations that call for a global surrogate, the `gaussian_process surfpack` is generally recommended. The `use_derivatives` keyword will only be useful if accurate and an inexpensive derivatives are available. Finite difference derivatives are disqualified on both counts. However, derivatives generated by analytical, automatic differentiation, or continuous adjoint techniques can be appropriate. Currently, first order derivatives, i.e. gradients, are the highest order derivatives that can be used to construct the `gaussian_process surfpack` model; Hessians will not be used even if they are available.

Chapter 21

Restart Capabilities and Utilities

21.1 Restart Management

DAKOTA was developed for solving problems that require multiple calls to computationally expensive simulation codes. In some cases you may want to conduct the same optimization, but to a tighter final convergence tolerance. This would be costly if the entire optimization analysis had to be repeated. Interruptions imposed by computer usage policies, power outages, and system failures could also result in costly delays. However, DAKOTA automatically records the variable and response data from all function evaluations so that new executions of DAKOTA can pick up where previous executions left off.

The DAKOTA restart file (e.g., `dakota.rst`) is written in a portable binary format. The portability derives from use of the XDR standard. As shown in Section 2.1.2, the primary restart commands for DAKOTA are `-read_restart`, `-write_restart`, and `-stop_restart`.

To write a restart file using a particular name, the `-write_restart` command line input (may be abbreviated as `-w`) is used:

```
dakota -i dakota.in -write_restart my_restart_file
```

If no `-write_restart` specification is used, then DAKOTA will still write a restart file, but using the default name `dakota.rst` instead of a user-specified name. To turn restart recording off, the user may select `deactivate restart_file` in the `interface` specification (refer to the Interface Commands chapter in the DAKOTA Reference Manual [3] for additional information). This can increase execution speed and reduce disk storage requirements, but at the expense of a loss in the ability to recover and continue a run that terminates prematurely. Obviously, this option is not recommended when function evaluations are costly or prone to failure. Please note that using the `deactivate restart_file` specification will result in a zero length restart file with the default name dakota.rst.

To restart DAKOTA from a restart file, the `-read_restart` command line input (may be abbreviated as `-r`) is used:

```
dakota -i dakota.in -read_restart my_restart_file
```

If no `-read_restart` specification is used, then DAKOTA will not read restart information from any file (i.e., the default is no restart processing).

If the -write_restart and -read_restart specifications identify the same file (including the case where -write_restart is not specified and -read_restart identifies dakota.rst), then new evaluations will be appended to the existing restart file. If the -write_restart and -read_restart specifications identify different files, then the evaluations read from the file identified by -read_restart are first written to the -write_restart file. Any new evaluations are then appended to the -write_restart file. In this way, restart operations can be chained together indefinitely with the assurance that all of the relevant evaluations are present in the latest restart file.

To read in only a portion of a restart file, the -stop_restart control (may be abbreviated as -s) is used to specify the number of entries to be read from the database. Note that this integer value corresponds to the restart record processing counter (as can be seen when using the print utility; see Section 21.2.1 below), which may differ from the evaluation numbers used in the previous run if, for example, any duplicates were detected (since these duplicates are not recorded in the restart file). In the case of a -stop_restart specification, it is usually desirable to specify a new restart file using -write_restart so as to remove the records of erroneous or corrupted function evaluations. For example, to read in the first 50 evaluations from dakota.rst:

```
dakota -i dakota.in -r dakota.rst -s 50 -w dakota_new.rst
```

The dakota_new.rst file will contain the 50 processed evaluations from dakota.rst as well as any new evaluations. All evaluations following the 50th in dakota.rst have been removed from the latest restart record.

DAKOTA's restart algorithm relies on its duplicate detection capabilities. Processing a restart file populates the list of function evaluations that have been performed. Then, when the study is restarted, it is started from the beginning (not a "warm" start) and many of the function evaluations requested by the iterator are intercepted by the duplicate detection code. This approach has the primary advantage of restoring the complete state of the iteration (including the ability to correctly detect subsequent duplicates) for all iterators and multi-iterator strategies without the need for iterator-specific restart code. However, the possibility exists for numerical round-off error to cause a divergence between the evaluations performed in the previous and restarted studies. This has been extremely rare to date.

21.2 The DAKOTA Restart Utility

The DAKOTA restart utility program provides a variety of facilities for managing restart files from DAKOTA executions. The executable program name is dakota_restart_util and it has the following options, as shown by the usage message returned when executing the utility without any options:

```
Usage: "dakota_restart_util print <restart_file>"
       "dakota_restart_util to_neutral <restart_file> <neutral_file>"
       "dakota_restart_util from_neutral <neutral_file> <restart_file>"
       "dakota_restart_util to_pdb <restart_file> <pdb_file>"
       "dakota_restart_util to_tabular <restart_file> <text_file>"
       "dakota_restart_util remove <double> <old_restart_file>
          <new_restart_file>"
       "dakota_restart_util remove_ids <int_1> ... <int_n> <old_restart_file>
          <new_restart_file>"
       "dakota_restart_util cat <restart_file_1> ... <restart_file_n>
          <new_restart_file>"
```

Several of these functions involve format conversions. In particular, the binary format used for restart files can be converted to ASCII text and printed to the screen, converted to and from a neutral file format, converted to a PDB format for use at Lawrence Livermore National Laboratory, or converted to a tabular format for importing

into 3rd-party graphics programs. In addition, a restart file with corrupted data can be repaired by value or id, and multiple restart files can be combined to create a master database.

21.2.1 Print

The `print` option is quite useful for interrogating the contents of a particular restart file, since the binary format is not convenient for direct inspection. The restart data is printed in full precision, so that exact matching of points is possible for restarted runs or corrupted data removals. For example, the following command

```
dakota_restart_util print dakota.rst
```

results in output similar to the following (from the `Dakota/test/dakota_cyl_head.in` example problem):

```
----------------------------------------
Restart record    1  (evaluation id    1):
----------------------------------------
Parameters:
                      1.8000000000000000e+00 intake_dia
                      1.0000000000000000e+00 flatness

Active response data:
Active set vector = { 3 3 3 3 }
                     -2.4355973813420619e+00 obj_fn
                     -4.7428486677140930e-01 nln_ineq_con_1
                     -4.5000000000000001e-01 nln_ineq_con_2
                      1.3971143170299741e-01 nln_ineq_con_3
 [ -4.3644298963447897e-01  1.4999999999999999e-01 ] obj_fn gradient
 [  1.3855136437818300e-01  0.0000000000000000e+00 ] nln_ineq_con_1 gradient
 [  0.0000000000000000e+00  1.4999999999999999e-01 ] nln_ineq_con_2 gradient
 [  0.0000000000000000e+00 -1.9485571585149869e-01 ] nln_ineq_con_3 gradient

----------------------------------------
Restart record    2  (evaluation id    2):
----------------------------------------
Parameters:
                      2.1640000000000001e+00 intake_dia
                      1.7169994018008317e+00 flatness

Active response data:
Active set vector = { 3 3 3 3 }
                     -2.4869127192988878e+00 obj_fn
                      6.9256958799989843e-01 nln_ineq_con_1
                     -3.4245008972987528e-01 nln_ineq_con_2
                      8.7142207937157910e-03 nln_ineq_con_3
 [ -4.3644298963447897e-01  1.4999999999999999e-01 ] obj_fn gradient
 [  2.9814239699997572e+01  0.0000000000000000e+00 ] nln_ineq_con_1 gradient
 [  0.0000000000000000e+00  1.4999999999999999e-01 ] nln_ineq_con_2 gradient
 [  0.0000000000000000e+00 -1.6998301774282701e-01 ] nln_ineq_con_3 gradient

...<snip>...

Restart file processing completed: 11 evaluations retrieved.
```

21.2.2 To/From Neutral File Format

A DAKOTA restart file can be converted to a neutral file format using a command like the following:

```
dakota_restart_util to_neutral dakota.rst dakota.neu
```

which results in a report similar to the following:

```
Writing neutral file dakota.neu
Restart file processing completed: 11 evaluations retrieved.
```

Similarly, a neutral file can be returned to binary format using a command like the following:

```
dakota_restart_util from_neutral dakota.neu dakota.rst
```

which results in a report similar to the following:

```
Reading neutral file dakota.neu
Writing new restart file dakota.rst
Neutral file processing completed: 11 evaluations retrieved.
```

The contents of the generated neutral file are similar to the following (from the first two records for the `Dakota/test/dakota_cyl_head.in` example problem):

```
6 7 2 1.8000000000000000e+00 intake_dia 1.0000000000000000e+00 flatness 0 0 0 0
NULL 4 2 1 0 3 3 3 3 1 2 obj_fn nln_ineq_con_1 nln_ineq_con_2 nln_ineq_con_3
  -2.4355973813420619e+00 -4.7428486677140930e-01 -4.5000000000000001e-01
   1.3971143170299741e-01 -4.3644298963447897e-01  1.4999999999999999e-01
   1.3855136437818300e-01  0.0000000000000000e+00  0.0000000000000000e+00
   1.4999999999999999e-01  0.0000000000000000e+00 -1.9485571585149869e-01 1
6 7 2 2.1640000000000001e+00 intake_dia 1.7169994018008317e+00 flatness 0 0 0 0
NULL 4 2 1 0 3 3 3 3 1 2 obj_fn nln_ineq_con_1 nln_ineq_con_2 nln_ineq_con_3
  -2.4869127192988878e+00 6.9256958799989843e-01 -3.4245008972987528e-01
   8.7142207937157910e-03 -4.3644298963447897e-01  1.4999999999999999e-01
   2.9814239699997572e+01  0.0000000000000000e+00  0.0000000000000000e+00
   1.4999999999999999e-01  0.0000000000000000e+00 -1.6998301774282701e-01 2
```

This format is not intended for direct viewing (`print` should be used for this purpose). Rather, the neutral file capability has been used in the past for managing portability of restart data across platforms (recent use of the XDR standard for portable binary formats has eliminated this need) or for advanced repair of restart records (in cases where the techniques of Section 21.2.5 were insufficient).

21.2.3 To Tabular Format

Conversion of a binary restart file to a tabular format enables convenient import of this data into 3rd-party post-processing tools such as Matlab, TECplot, Excel, etc. This facility is nearly identical to the `tabular_graphics_data` option in the DAKOTA input file specification (described in Section 15.3), but with two important differences:

1. No function evaluations are suppressed as they are with `tabular_graphics_data` (i.e., any internal finite difference evaluations are included).

2. The conversion can be performed posthumously, i.e., for DAKOTA runs executed previously.

An example command for converting a restart file to tabular format is:

```
dakota_restart_util to_tabular dakota.rst dakota.m
```

which results in a report similar to the following:

```
Writing tabular text file dakota.m
Restart file processing completed: 10 evaluations tabulated.
```

The contents of the generated tabular file are similar to the following (from the `Dakota/test/dakota_textbook.in` example problem). Note that, while evaluations resulting from numerical derivative offsets would be reported (as described above), derivatives returned as part of the evaluations are not reported (since they do not readily fit within a compact tabular format):

```
%eval_id            x1             x2         obj_fn nln_ineq_con_1 nln_ineq_con_2
       1           0.9            1.1         0.0002           0.26           0.76
       2   0.6433962264   0.6962264151   0.0246865569  0.06584549662   0.1630331079
       3   0.5310576935   0.5388046558  0.09360081618  0.01261994597  0.02478161031
       4    0.612538853   0.6529854907  0.03703861037  0.04871110113   0.1201206246
       5   0.5209215947   0.5259311717   0.1031862798  0.00839372202  0.01614279999
       6   0.5661606434   0.5886684401  0.06405197568  0.02620365411  0.06345021064
       7   0.5083873357    0.510239856   0.1159458957 0.003337755086 0.006151042802
       8   0.5001577143   0.5001800249   0.1248312163 6.772666885e-05 0.0001012002012
       9   0.5000000547   0.5000000598   0.1249999428 2.485652461e-08 3.238746073e-08
      10            0.5            0.5          0.125 2.942091015e-15 3.60822483e-15
```

21.2.4 Concatenation of Multiple Restart Files

In some instances, it is useful to combine restart files into a single master function evaluation database. For example, when constructing a data fit surrogate model, data from previous studies can be pulled in and reused to create a combined data set for the surrogate fit. An example command for concatenating multiple restart files is:

```
dakota_restart_util cat dakota.rst.1 dakota.rst.2 dakota.rst.3 dakota.rst.all
```

which results in a report similar to the following:

```
Writing new restart file dakota.rst.all
dakota.rst.1 processing completed: 10 evaluations retrieved.
dakota.rst.2 processing completed: 110 evaluations retrieved.
dakota.rst.3 processing completed: 65 evaluations retrieved.
```

The `dakota.rst.all` database now contains 185 evaluations and can be read in for use in a subsequent DAKOTA study using the `-read_restart` option to the `dakota` executable (see Section 21.1).

21.2.5 Removal of Corrupted Data

On occasion, a simulation or computer system failure may cause a corruption of the DAKOTA restart file. For example, a simulation crash may result in failure of a post-processor to retrieve meaningful data. If 0's (or other

erroneous data) are returned from the user's `analysis_driver`, then this bad data will get recorded in the restart file. If there is a clear demarcation of where corruption initiated (typical in a process with feedback, such as gradient-based optimization), then use of the `-stop_restart` option for the `dakota` executable can be effective in continuing the study from the point immediately prior to the introduction of bad data. If, however, there are interspersed corruptions throughout the restart database (typical in a process without feedback, such as sampling), then the `remove` and `remove_ids` options of `dakota_restart_util` can be useful.

An example of the command syntax for the `remove` option is:

```
dakota_restart_util remove 2.e-04 dakota.rst dakota.rst.repaired
```

which results in a report similar to the following:

```
Writing new restart file dakota.rst.repaired
Restart repair completed: 65 evaluations retrieved, 2 removed, 63 saved.
```

where any evaluations in `dakota.rst` having an active response function value that matches `2.e-04` within machine precision are discarded when creating `dakota.rst.repaired`.

An example of the command syntax for the `remove_ids` option is:

```
dakota_restart_util remove_ids 12 15 23 44 57 dakota.rst dakota.rst.repaired
```

which results in a report similar to the following:

```
Writing new restart file dakota.rst.repaired
Restart repair completed: 65 evaluations retrieved, 5 removed, 60 saved.
```

where evaluation ids `12`, `15`, `23`, `44`, and `57` have been discarded when creating `dakota.rst.repaired`. An important detail is that, unlike the `-stop_restart` option which operates on restart record numbers (see Section 21.1)), the `remove_ids` option operates on evaluation ids. Thus, removal is not necessarily based on the order of appearance in the restart file. This distinction is important when removing restart records for a run that contained either asynchronous or duplicate evaluations, since the restart insertion order and evaluation ids may not correspond in these cases (asynchronous evaluations have ids assigned in the order of job creation but are inserted in the restart file in the order of job completion, and duplicate evaluations are not recorded which introduces offsets between evaluation id and record number). This can also be important if removing records from a concatenated restart file, since the same evaluation id could appear more than once. In this case, all evaluation records with ids matching the `remove_ids` list will be removed.

If neither of these removal options is sufficient to handle a particular restart repair need, then the fallback position is to resort to direct editing of a neutral file (refer to Section 21.2.2) to perform the necessary modifications.

Chapter 22

Simulation Failure Capturing

DAKOTA provides the capability to manage failures in simulation codes within its system call, fork, and direct simulation interfaces (see Section 12.3 for simulation interface descriptions). Failure capturing consists of three operations: failure detection, failure communication, and failure mitigation.

22.1 Failure detection

Since the symptoms of a simulation failure are highly code and application dependent, it is the user's responsibility to detect failures within their `analysis_driver`, `input_filter`, or `output_filter`. One popular example of simulation monitoring is to rely on a simulation's internal detection of errors. In this case, the UNIX `grep` utility can be used within a user's driver/filter script to detect strings in output files which indicate analysis failure. For example, the following simple C shell script excerpt

```
grep ERROR analysis.out > /dev/null
if ( $status == 0 )
  echo "FAIL" > results.out
endif
```

will pass the `if` test and communicate simulation failure to DAKOTA if the `grep` command finds the string `ERROR` anywhere in the `analysis.out` file. The `/dev/null` device file is called the "bit bucket" and the `grep` command output is discarded by redirecting it to this destination. The `$status` shell variable contains the exit status of the last command executed [5], which is the exit status of `grep` in this case (0 if successful in finding the error string, nonzero otherwise). For Bourne shells [10], the `$?` shell variable serves the same purpose as `$status` for C shells. In a related approach, if the return code from a simulation can be used directly for failure detection purposes, then `$status` or `$?` could be queried immediately following the simulation execution using an `if` test like that shown above.

If the simulation code is not returning error codes or providing direct error diagnostic information, then failure detection may require monitoring of simulation results for sanity (e.g., is the mesh distorting excessively?) or potentially monitoring for continued process existence to detect a simulation segmentation fault or core dump. While this can get complicated, the flexibility of DAKOTA's interfaces allows for a wide variety of user-defined monitoring approaches.

22.2 Failure communication

Once a failure is detected, it must be communicated so that DAKOTA can take the appropriate corrective action. The form of this communication depends on the type of simulation interface in use.

In the system call and fork simulation interfaces, a detected simulation failure is communicated to DAKOTA through the results file. Instead of returning the standard results file data, the string "`fail`" should appear at the beginning of the results file. Any data appearing after the fail string will be ignored. Also, DAKOTA's detection of this string is case insensitive, so "`FAIL`", "`Fail`", etc., are equally valid.

In the direct simulation interface case, a detected simulation failure is communicated to DAKOTA through the return code provided by the user's `analysis_driver`, `input_filter`, or `output_filter`. As shown in Section 18.2.1, the prototype for simulations linked within the direct interface includes an integer return code. This code has the following meanings: zero (false) indicates that all is normal and nonzero (true) indicates an exception (i.e., a simulation failure).

22.3 Failure mitigation

Once the analysis failure has been communicated, DAKOTA will attempt to recover from the failure using one of the following four mechanisms, as governed by the interface specification in the user's input file (see the Interface Commands chapter in the DAKOTA Reference Manual [3] for additional information).

22.3.1 Abort (default)

If the `abort` option is active (the default), then DAKOTA will terminate upon detecting a failure. Note that if the problem causing the failure can be corrected, DAKOTA's restart capability (see Chapter 21) can be used to continue the study.

22.3.2 Retry

If the `retry` option is specified, then DAKOTA will re-invoke the failed simulation up to the specified number of retries. If the simulation continues to fail on each of these retries, DAKOTA will terminate. The retry option is appropriate for those cases in which simulation failures may be resulting from transient computing environment issues, such as shared disk space, software license access, or networking problems.

22.3.3 Recover

If the `recover` option is specified, then DAKOTA will not attempt the failed simulation again. Rather, it will return a "dummy" set of function values as the results of the function evaluation. The dummy function values to be returned are specified by the user. Any gradient or Hessian data requested in the active set vector will be zero. This option is appropriate for those cases in which a failed simulation may indicate a region of the design space to be avoided and the dummy values can be used to return a large objective function or constraint violation which will discourage an optimizer from further investigating the region.

22.3.4 Continuation

If the `continuation` option is specified, then DAKOTA will attempt to step towards the failing "target" simulation from a nearby "source" simulation through the use of a continuation algorithm. This option is appropriate for those cases in which a failed simulation may be caused by an inadequate initial guess. If the "distance" between the source and target can be divided into smaller steps in which information from one step provides an adequate initial guess for the next step, then the continuation method can step towards the target in increments sufficiently small to allow for convergence of the simulations.

When the failure occurs, the interval between the last successful evaluation (the source point) and the current target point is halved and the evaluation is retried. This halving is repeated until a successful evaluation occurs. The algorithm then marches towards the target point using the last interval as a step size. If a failure occurs while marching forward, the interval will be halved again. Each invocation of the continuation algorithm is allowed a total of ten failures (ten halvings result in up to 1024 evaluations from source to target) prior to aborting the DAKOTA process.

While DAKOTA manages the interval halving and function evaluation invocations, the user is responsible for managing the initial guess for the simulation program. For example, in a GOMA input file [100], the user specifies the files to be used for reading initial guess data and writing solution data. When using the last successful evaluation in the continuation algorithm, the translation of initial guess data can be accomplished by simply copying the solution data file leftover from the last evaluation to the initial guess file for the current evaluation (and in fact this is useful for all evaluations, not just continuation). However, a more general approach would use the *closest* successful evaluation (rather than the *last* successful evaluation) as the source point in the continuation algorithm. This will be especially important for nonlocal methods (e.g., genetic algorithms) in which the last successful evaluation may not necessarily be in the vicinity of the current evaluation. This approach will require the user to save and manipulate previous solutions (likely tagged with evaluation number) so that the results from a particular simulation (specified by DAKOTA after internal identification of the closest point) can be used as the current simulation's initial guess. This more general approach is not yet supported in DAKOTA.

22.4 Special values

In IEEE arithmetic, "NaN" indicates "not a number" and ±"Inf" or ±"Infinity" indicates positive or negative infinity. These special values may be returned directly in function evaluation results from a simulation interface or they may be specified in a user's input file within the `recover` specification described in Section 22.3.3. There is a key difference between these two cases. In the former case of direct simulation return, failure mitigation can be managed on a per response function basis. When using `recover`, however, the failure applies to the complete set of simulation results.

In both of these cases, the handling of NaN or Inf is managed using iterator-specific approaches. Currently, nondeterministic sampling methods (see Section 6.2), polynomial chaos expansions using either regression approaches or spectral projection with random sampling (see Section 6.4), and the NL2SOL method for nonlinear least squares (see §8.2.3) are the only methods with special numerical exception handling: the sampling methods simply omit any samples that are not finite from the statistics generation, the polynomial chaos methods omit any samples that are not finite from the coefficient estimation, and NL2SOL treats NaN or Infinity in a residual vector (i.e., values in a results file for a function evaluation) computed for a trial step as an indication that the trial step was too long and violates an unstated constraint; NL2SOL responds by trying a shorter step.

Chapter 23

Additional Examples

This chapter contains additional examples of DAKOTA methods and verification test problems. While the test problems and binary analysis drivers are principally managed in `Dakota/test`, many extracted input files are also included in `Dakota/examples/tutorial` and `Dakota/examples/methods` for user convenience.

23.1 Textbook Example

Equation 2.3 presents the 2-dimensional form of the textbook problem. An extended formulation is stated as

$$\begin{aligned} \texttt{minimize} \quad & f = \sum_{i=1}^{n} (x_i - 1)^4 \\ \texttt{subject to} \quad & g_1 = x_1^2 - \frac{x_2}{2} \leq 0 \\ & g_2 = x_2^2 - \frac{x_1}{2} \leq 0 \\ & 0.5 \leq x_1 \leq 5.8 \\ & -2.9 \leq x_2 \leq 2.9 \end{aligned} \tag{23.1}$$

where n is the number of design variables. The objective function is designed to accommodate an arbitrary number of design variables in order to allow flexible testing of a variety of data sets. Contour plots for the $n = 2$ case have been shown previously in Figure 2.2.

This example problem may also be used to exercise least squares solution methods by modifying the problem formulation to:

$$\texttt{minimize} \;\; (f)^2 + (g_1)^2 + (g_2)^2 \tag{23.2}$$

This modification is performed by simply changing the responses specification for the three functions from `num_objective_functions = 1` and `num_nonlinear_inequality_constraints = 2` to `num_least_squares_terms = 3`. Note that the two problem formulations are not equivalent and have different solutions.

Another way to exercise the least squares methods which would be equivalent to the optimization formulation would be to select the residual functions to be $(x_i - 1)^2$. However, this formulation requires modification to `Dakota/test/text_book.C` and will not be presented here. Equation 23.2, on the other hand, can use the

existing `text_book.C` without modification. Refer to Section 23.2 for an example of minimizing the same objective function using both optimization and least squares approaches.

23.1.1 Methods

The `dakota_textbook.in` file provided in the `Dakota/examples/tutorial` directory selects a `conmin_mfd` optimizer to perform constrained minimization using the `text_book` simulator. Additional gradient-based methods that can be used include methods from DOT, NPSOL, NLPQL, and OPT++. In addition the unconstrained least squares formulation of Equation 23.2 can be solved using OPT++ Gauss-Newton, NLSSOL, and NL2SOL methods.

A hybrid optimization can also be demonstrated on the `text_book` problem. The `dakota_hybrid.in` file provided in `Dakota/examples/methods` starts with a `coliny_ea` solution which feeds its best point into a `coliny_pattern_search` optimization which feeds its best point into `optpp_newton`. While this approach is overkill for such a simple problem, it is useful for demonstrating the coordination between multiple methods in the hybrid strategy.

In addition, `Dakota/test/dakota_textbook_3pc.in` demonstrates the use of a 3-piece interface to perform the parameter to response mapping, and `Dakota/test/dakota_textbook_lhs.in` demonstrates the use of Latin hypercube Monte Carlo sampling for assessing probability of failure as measured by specified response levels.

23.1.2 Optimization Results

For the optimization problem given in Equation 23.1, the unconstrained solution (`num_nonlinear_inequality_constraints` set to zero) for two design variables is:

$$\begin{aligned} x_1 &= 1.0 \\ x_2 &= 1.0 \end{aligned}$$

with

$$f^* = 0.0$$

The solution for the optimization problem constrained by g_1 (`num_nonlinear_inequality_constraints` set to one) is:

$$\begin{aligned} x_1 &= 0.763 \\ x_2 &= 1.16 \end{aligned}$$

with

$$\begin{aligned} f^* &= 0.00388 \\ g_1^* &= 0.0 \ \text{(active)} \end{aligned}$$

The solution for the optimization problem constrained by g_1 and g_2 (`num_nonlinear_inequality_constraints` set to two) is:

$$\begin{aligned} x_1 &= 0.500 \\ x_2 &= 0.500 \end{aligned}$$

with

$$\begin{aligned} f^* &= 0.125 \\ g_1^* &= 0.0 \ \text{(active)} \\ g_2^* &= 0.0 \ \text{(active)} \end{aligned}$$

Note that as constraints are added, the design freedom is restricted (the additional constraints are active at the solution) and an increase in the optimal objective function is observed.

23.1.3 Least Squares Results

The solution for the least squares problem given in Equation 23.2 is:

$$\begin{aligned} x_1 &= 0.566 \\ x_2 &= 0.566 \end{aligned}$$

with the residual functions equal to

$$\begin{aligned} f^* &= 0.0713 \\ g_1^* &= 0.0371 \\ g_2^* &= 0.0371 \end{aligned}$$

and a minimal sum of the squares of 0.00783.

This study requires selection of `num_least_squares_terms = 3` in the responses specification and selection of either `optpp_g_newton`, `nlssol_sqp`, or `nl2sol` in the method specification.

23.2 Rosenbrock Example

The Rosenbrock function [46] is a well known benchmark problem for optimization algorithms. Its standard two-dimensional formulation can be stated as

$$\texttt{minimize} \ \ f = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{23.3}$$

Two n-dimensional formulations are present in the literature. First, [86] formulates an "extended Rosenbrock" as:

$$f = \sum_{i=1}^{n/2} \left[\alpha(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2\right] \tag{23.4}$$

Second, [98] formulates a "generalized Rosenbrock" as:

$$f = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \tag{23.5}$$

These formulations are not currently supported in DAKOTA's system/fork interfaces, but are supported in the direct interface.

Surface and contour plots for this function have been shown previously in Figure 2.1. This example problem may also be used to exercise least squares solution methods by recasting the problem formulation into:

$$\texttt{minimize} \;\; f = (f_1)^2 + (f_2)^2 \tag{23.6}$$

where

$$f_1 = 10(x_2 - x_1^2) \tag{23.7}$$

and

$$f_2 = 1 - x_1 \tag{23.8}$$

are residual terms. In this case (unlike the least squares modification in Section 23.1), the two problem formulations are equivalent and have identical solutions.

23.2.1 Methods

In the `Dakota/test` directory, the `rosenbrock` executable (compiled from `rosenbrock.C`) checks the number of response functions passed in the parameters file and returns either an objective function (as computed from Equation 23.3) for use with optimization methods or two least squares terms (as computed from Equations 23.7-23.8) for use with least squares methods. Both cases support analytic gradients of the function set with respect to the design variables. The `dakota_rosenbrock.in` input file can be used to solve both problems by toggling settings in the method and responses specifications. To run the optimization solution, select `num_objective_functions = 1` in the responses specification, and select an optimizer (e.g., `optpp_q_newton`) in the method specification, e.g., as shown in `Dakota/examples/methods/dakota_addtnl_rosen_opt.in`:

```
method,
        optpp_q_newton
          convergence_tolerance = 1e-10

variables,
        continuous_design = 2
          initial_point -1.2   1.0
          lower_bounds  -2.0  -2.0
          upper_bounds   2.0   2.0
          descriptors    'x1'  'x2'

interface,
        system
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        analytic_gradients
        no_hessians
```

To run the least squares solution, the responses specification is changed to `num_least_squares_terms = 2` and the method specification is changed to a least squares method, e.g., `optpp_g_newton`, as shown in `Dakota/examples/methods/dakota_addtnl_rosen_ls.in`:

```
method,
        optpp_g_newton
          convergence_tolerance = 1e-10

variables,
        continuous_design = 2
          initial_point -1.2   1.0
          lower_bounds  -2.0  -2.0
          upper_bounds   2.0   2.0
          descriptors    'x1'  'x2'

interface,
        system
          analysis_driver = 'rosenbrock'

responses,
        calibration_terms = 2
        analytic_gradients
        no_hessians
```

23.2.2 Results

The optimal solution, solved either as a least squares problem or an optimization problem, is:

$$\begin{aligned} x_1 &= 1.0 \\ x_2 &= 1.0 \end{aligned}$$

with

$$f^* = 0.0$$

In comparing the two approaches, one would expect the Gauss-Newton approach to be more efficient since it exploits the special-structure of a least squares objective function and, in this problem, the Gauss-Newton Hessian is a good approximation since the least squares residuals are zero at the solution. From a good initial guess, this expected behavior is clearly demonstrated. Starting from `cdv_initial_point = 0.8, 0.7`, the `optpp_g_newton` method converges in only 3 function and gradient evaluations while the `optpp_q_newton` method requires 27 function and gradient evaluations to achieve similar accuracy. Starting from a poorer initial guess (e.g., `cdv_initial_point = -1.2, 1.0`), the trend is less obvious since both methods spend several evaluations finding the vicinity of the minimum (total function and gradient evaluations = 45 for `optpp_q_newton` and 29 for `optpp_g_newton`). However, once the vicinity is located and the Hessian approximation becomes accurate, convergence is much more rapid with the Gauss-Newton approach.

Shown below is the complete DAKOTA output for the `optpp_g_newton` method starting from `cdv_initial_point = 0.8, 0.7`:

```
Running MPI executable in serial mode.
DAKOTA version 5.0+ developmental release.
Subversion revision 172 built Dec  8 2010 17:35:13.
Constructing Single Method Strategy...
Writing new restart file dakota.rst
```

```
methodName = optpp_g_newton
gradientType = analytic
hessianType = none

>>>>> Running Single Method Strategy.

>>>>> Running optpp_g_newton iterator.

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      8.0000000000e-01 x1
                      7.0000000000e-01 x2

rosenbrock /tmp/file4t29aW /tmp/filezGeFpF

Active response data for function evaluation 1:
Active set vector = { 3 3 } Deriv vars vector = { 1 2 }
                      6.0000000000e-01 least_sq_term_1
                      2.0000000000e-01 least_sq_term_2
 [ -1.6000000000e+01  1.0000000000e+01 ] least_sq_term_1 gradient
 [ -1.0000000000e+00  0.0000000000e+00 ] least_sq_term_2 gradient




------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                      9.9999528206e-01 x1
                      9.5999243139e-01 x2

rosenbrock /tmp/fileSaxQHo /tmp/fileHnU1Z7

Active response data for function evaluation 2:
Active set vector = { 3 3 } Deriv vars vector = { 1 2 }
                     -3.9998132761e-01 least_sq_term_1
                      4.7179363810e-06 least_sq_term_2
 [ -1.9999905641e+01  1.0000000000e+01 ] least_sq_term_1 gradient
 [ -1.0000000000e+00  0.0000000000e+00 ] least_sq_term_2 gradient




------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                      9.9999904377e-01 x1
                      9.9999808276e-01 x2

rosenbrock /tmp/filedtRtlR /tmp/file3kUVGA
```

```
Active response data for function evaluation 3:
Active set vector = { 3 3 } Deriv vars vector = { 1 2 }
                    -4.7950734494e-08 least_sq_term_1
                     9.5622502239e-07 least_sq_term_2
 [ -1.9999980875e+01  1.0000000000e+01 ] least_sq_term_1 gradient
 [ -1.0000000000e+00  0.0000000000e+00 ] least_sq_term_2 gradient




------------------------------
Begin Function Evaluation    4
------------------------------
Parameters for function evaluation 4:
                     9.9999904377e-01 x1
                     9.9999808276e-01 x2

Duplication detected: analysis_drivers not invoked.

Active response data retrieved from database:
Active set vector = { 2 2 } Deriv vars vector = { 1 2 }
 [ -1.9999980875e+01  1.0000000000e+01 ] least_sq_term_1 gradient
 [ -1.0000000000e+00  0.0000000000e+00 ] least_sq_term_2 gradient


<<<<< Function evaluation summary: 4 total (3 new, 1 duplicate)
<<<<< Best parameters          =
                     9.9999904377e-01 x1
                     9.9999808276e-01 x2
<<<<< Best residual norm =  9.5742653315e-07; 0.5 * norm^2 =  4.5833278319e-13
<<<<< Best residual terms      =
                    -4.7950734494e-08
                     9.5622502239e-07
<<<<< Best data captured at function evaluation 3
Confidence Interval for x1 is [  9.9998687852e-01,  1.0000112090e+00 ]
Confidence Interval for x2 is [  9.9997372187e-01,  1.0000224436e+00 ]

<<<<< Iterator optpp_g_newton completed.
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =       0.01 [parent =   0.017997, child =  -0.007997]
  Total wall clock =  0.0672231
```

23.3 Herbie, Smooth Herbie, and Shubert Example

Lee, et al. [76] developed the Herbie function as a 2D test problem for surrogate-based optimization. However, since it is separable and each dimension is identical it is easily generalized to an arbitrary number of dimensions.

Figure 23.1: Plots of the `herbie` (left) and `smooth_herbie` (right) test functions in 2 dimensions. They can accept an arbitrary number of inputs. The direction of the z-axis has been reversed (negative is up) to better view the functions' minima.

The generalized (to M dimensions) Herbie function is

$$\text{herb}(\underline{x}) = -\prod_{k=1}^{M} w_{herb}\left(x_k\right)$$

where

$$w_{herb}\left(x_k\right) = \exp(-(x_k-1)^2) + \exp(-0.8(x_k+1)^2) - 0.05\sin\left(8\left(x_k+0.1\right)\right).$$

The Herbie function's high frequency sine component creates a large number of local minima and maxima, making it a significantly more challenging test problem. However, when testing a method's ability to exploit smoothness in the true response, it is desirable to have a less oscillatory function. For this reason, the "smooth Herbie" test function omits the high frequency sine term but is otherwise identical to the Herbie function. The formula for smooth Herbie is

$$\text{herb}_{\text{sm}}(\underline{x}) = -\prod_{k=1}^{M} w_{sm}\left(x_k\right)$$

where

$$w_{sm}\left(x_k\right) = \exp(-(x_k-1)^2) + \exp(-0.8(x_k+1)^2).$$

Two dimensional versions of the `herbie` and `smooth_herbie` test functions are plotted in Figure 23.1.

Shubert is another separable (and therefore arbitrary dimensional) test function. Its analytical formula is

$$\text{shu}(\underline{x}) = \prod_{k=1}^{M} w_{shu}\left(x_k\right)$$

where

$$w_{shu}\left(x_k\right) = \sum_{i=1}^{5} i\cos((i+1)x_k + i)$$

The 2D version of the `shubert` function is shown in Figure 23.2.

The DAKOTA input file `dakota_separable_ego_5D.in` shows how to use efficient global optimization (ego) to minimize the 5D version of any of these 3 separable functions. Note that in the variables section the `5*` preceding the values -2.0 and 2.0 for the `lower_bounds` and `upper_bounds`, respectively, tells DAKOTA to repeat them 5 times. The "interesting" region for each of these functions is $-2 \leq x_k \leq 2$ for all dimensions.

```
# Example of using EGO to find the minumum of the 5 dimensional version
# of the abitrary-dimensional/separable 'herbie' OR 'smooth_herbie' OR
# 'shubert' test functions

strategy,
        single_method
          tabular_graphics_data

method,
        efficient_global                # EGO Efficient Global Optimization
          seed = 123456

variables,
        continuous_design = 5               # 5 dimensions
          lower_bounds      5*-2.0          # use 5 copies of -2.0 for bound
          upper_bounds      5*2.0           # use 5 copies of 2.0 for bound

interface,
        direct
#         analysis_driver = 'herbie'        # use this for herbie
          analysis_driver = 'smooth_herbie' # use this for smooth_herbie
#         analysis_driver = 'shubert'       # use this for shubert

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

23.4 Cylinder Head Example

The cylinder head example problem is stated as:

$$
\begin{aligned}
\texttt{minimize} \quad & f = -1\left(\frac{\texttt{horsepower}}{250} + \frac{\texttt{warranty}}{100000}\right) \\
\texttt{subject to} \quad & \sigma_{max} \leq 0.5\sigma_{yield} \\
& \texttt{warranty} \geq 100000 \\
& \texttt{time}_{\texttt{cycle}} \leq 60 \\
& 1.5 \leq \texttt{d}_{\texttt{intake}} \leq 2.164 \\
& 0.0 \leq \texttt{flatness} \leq 4.0
\end{aligned}
\tag{23.9}
$$

This formulation seeks to simultaneously maximize normalized engine horsepower and engine warranty over variables of valve intake diameter ($\texttt{d}_{\texttt{intake}}$) in inches and overall head flatness (`flatness`) in thousandths of an

Figure 23.2: Plot of the `shubert` test function in 2 dimensions. It can accept an arbitrary number of inputs.

inch subject to inequality constraints that the maximum stress cannot exceed half of yield, that warranty must be at least 100000 miles, and that manufacturing cycle time must be less than 60 seconds. Since the constraints involve different scales, they should be nondimensionalized (note: the nonlinear constraint scaling described in Section 7.3.2 can now do this automatically). In addition, they can be converted to the standard 1-sided form $g(\mathbf{x}) \leq 0$ as follows:

$$\begin{aligned} g_1 &= \frac{2\sigma_{\texttt{max}}}{\sigma_{\texttt{yield}}} - 1 \leq 0 \\ g_2 &= 1 - \frac{\texttt{warranty}}{100000} \leq 0 \\ g_3 &= \frac{\texttt{time}_{\texttt{cycle}}}{60} - 1 \leq 0 \end{aligned} \tag{23.10}$$

The objective function and constraints are related analytically to the design variables according to the following simple expressions:

$$\begin{aligned} \texttt{warranty} &= 100000 + 15000(4 - \texttt{flatness}) \\ \texttt{time}_{\texttt{cycle}} &= 45 + 4.5(4 - \texttt{flatness})^{1.5} \\ \texttt{horsepower} &= 250 + 200\left(\frac{\texttt{d}_{\texttt{intake}}}{1.833} - 1\right) \\ \sigma_{\texttt{max}} &= 750 + \frac{1}{(\texttt{t}_{\texttt{wall}})^{2.5}} \\ \texttt{t}_{\texttt{wall}} &= \texttt{offset}_{\texttt{intake}} - \texttt{offset}_{\texttt{exhaust}} - \frac{(\texttt{d}_{\texttt{intake}} - \texttt{d}_{\texttt{exhaust}})}{2} \end{aligned} \tag{23.11}$$

where the constants in Equation 23.10 and Equation 23.11 assume the following values: $\sigma_{\texttt{yield}} = 3000$, $\texttt{offset}_{\texttt{intake}} = 3.25$, $\texttt{offset}_{\texttt{exhaust}} = 1.34$, and $\texttt{d}_{\texttt{exhaust}} = 1.556$.

23.4.1 Methods

In the `Dakota/test` directory, the `dakota_cyl_head.in` input file is used to execute a variety of tests using the cylinder head example. One of these tests is shown below and is available directly in `Dakota/examples/methods/dakota_addtnl_cylhead.in`:

```
method,
        npsol_sqp
          convergence_tolerance = 1.e-8

variables,
        continuous_design = 2
          initial_point  1.8   1.0
          upper_bounds   2.164 4.0
          lower_bounds   1.5   0.0
          descriptors'intake_dia' 'flatness'

interface,
        fork asynchronous
          analysis_driver = 'cyl_head'

responses,
        objective_functions = 1
        nonlinear_inequality_constraints = 3
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-4
        no_hessians
```

The interface keyword specifies use of the `cyl_head` executable (compiled from `Dakota/test/cyl_head.C`) as the simulator. The variables and responses keywords specify the data sets to be used in the iteration by providing the initial point, descriptors, and upper and lower bounds for two continuous design variables and by specifying the use of one objective function, three inequality constraints, and numerical gradients in the problem. The method keyword specifies the use of the `npsol_sqp` method to solve this constrained optimization problem. No strategy keyword is specified, so the default `single_method` strategy is used.

23.4.2 Optimization Results

The solution for the constrained optimization problem is:

$$\begin{aligned} \text{intake_dia} &= 2.122 \\ \text{flatness} &= 1.769 \end{aligned}$$

with

$$\begin{aligned} f^* &= -2.461 \\ g_1^* &= 0.0 \ \text{(active)} \\ g_2^* &= -0.3347 \ \text{(inactive)} \\ g_3^* &= 0.0 \ \text{(active)} \end{aligned}$$

which corresponds to the following optimal response quantities:

$$\begin{aligned} \text{warranty} &= 133472 \\ \text{cycle_time} &= 60 \\ \text{wall_thickness} &= 0.0707906 \\ \text{horse_power} &= 281.579 \\ \text{max_stress} &= 1500 \end{aligned}$$

The final report from the DAKOTA output is as follows:

```
<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 55 total (55 new, 0 duplicate)
<<<<< Best parameters          =
                     2.1224188322e+00 intake_dia
                     1.7685568331e+00 flatness
<<<<< Best objective function  =
                    -2.4610312954e+00
<<<<< Best constraint values   =
                     1.8407497748e-13
                    -3.3471647504e-01
                     0.0000000000e+00
<<<<< Best data captured at function evaluation 51
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =       0.04 [parent =   0.031995, child =   0.008005]
  Total wall clock =   0.232134
```

23.5 Container Example

For this example, suppose that a high-volume manufacturer of light weight steel containers wants to minimize the amount of raw sheet material that must be used to manufacture a 1.1 quart cylindrical-shaped can, including waste material. Material for the container walls and end caps is stamped from stock sheet material of constant thickness. The seal between the end caps and container wall is manufactured by a press forming operation on the end caps. The end caps can then be attached to the container wall forming a seal through a crimping operation.

For preliminary design purposes, the extra material that would normally go into the container end cap seals is approximated by increasing the cut dimensions of the end cap diameters by 12% and the height of the container wall by 5%, and waste associated with stamping the end caps in a specialized pattern from sheet stock is estimated as 15% of the cap area. The equation for the area of the container materials including waste is

Figure 23.3: Container wall-to-end-cap seal

$$A = 2 \times \begin{pmatrix} \texttt{end cap} \\ \texttt{waste} \\ \texttt{material} \\ \texttt{factor} \end{pmatrix} \times \begin{pmatrix} \texttt{end cap} \\ \texttt{seal} \\ \texttt{material} \\ \texttt{factor} \end{pmatrix} \times \begin{pmatrix} \texttt{nominal} \\ \texttt{end cap} \\ \texttt{area} \end{pmatrix} + \begin{pmatrix} \texttt{container} \\ \texttt{wall seal} \\ \texttt{material} \\ \texttt{factor} \end{pmatrix} \times \begin{pmatrix} \texttt{nominal} \\ \texttt{container} \\ \texttt{wall area} \end{pmatrix}$$

or

$$A = 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)\pi DH \tag{23.12}$$

where D and H are the diameter and height of the finished product in units of inches, respectively. The volume of the finished product is specified to be

$$V = \pi\frac{D^2 H}{4} = (1.1\texttt{qt})(57.75\texttt{in}^3/\texttt{qt}) \tag{23.13}$$

The equation for area is the objective function for this problem; it is to be minimized. The equation for volume is an equality constraint; it must be satisfied at the conclusion of the optimization problem. Any combination of D and H that satisfies the volume constraint is a **feasible** solution (although not necessarily the optimal solution) to the area minimization problem, and any combination that does not satisfy the volume constraint is an **infeasible** solution. The area that is a minimum subject to the volume constraint is the **optimal** area, and the corresponding values for the parameters D and H are the optimal parameter values.

It is important that the equations supplied to a numerical optimization code be limited to generating only physically realizable values, since an optimizer will not have the capability to differentiate between meaningful and nonphysical parameter values. It is often up to the engineer to supply these limits, usually in the form of parameter bound constraints. For example, by observing the equations for the area objective function and the volume constraint, it can be seen that by allowing the diameter, D, to become negative, it is algebraically possible to generate relatively small values for the area that also satisfy the volume constraint. Negative values for D are of course physically meaningless. Therefore, to ensure that the numerically-solved optimization problem remains meaningful, a bound constraint of $-D \leq 0$ must be included in the optimization problem statement. A positive value for H is implied since the volume constraint could never be satisfied if H were negative. However, a bound constraint of $-H \leq 0$ can be added to the optimization problem if desired. The optimization problem can then

Figure 23.4: A graphical representation of the container optimization problem.

be stated in a standardized form as

$$\begin{aligned} \texttt{minimize} \quad & 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)^2\pi DH \\ \texttt{subject to} \quad & \pi\frac{D^2H}{4} = (1.1\texttt{qt})(57.75\texttt{in}^3/\texttt{qt}) \\ & -D \leq 0, \; -H \leq 0 \end{aligned} \tag{23.14}$$

A graphical view of the container optimization problem appears in Figure 23.4. The 3-D surface defines the area, A, as a function of diameter and height. The curved line that extends across the surface defines the areas that satisfy the volume equality constraint, V. Graphically, the container optimization problem can be viewed as one of finding the point along the constraint line with the smallest 3-D surface height in Figure 23.4. This point corresponds to the optimal values for diameter and height of the final product.

The input file for this test problem is named `dakota_container.in` in the directory `Dakota/examples/methods`. The solution to this example problem is $(H, D) = (4.99, 4.03)$, with a minimum area of 98.43 $\texttt{in}^2$.

The final report from the DAKOTA output is as follows:

```
<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 40 total (40 new, 0 duplicate)
<<<<< Best parameters          =
                     4.9873894231e+00 H
                     4.0270846274e+00 D
<<<<< Best objective function  =
                     9.8432498116e+01
<<<<< Best constraint values   =
                    -9.6301439045e-12
<<<<< Best data captured at function evaluation 36
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =      0.18 [parent =      0.18, child =         0]
  Total wall clock =  0.809126
```

23.6 Log Ratio Example

This test problem, mentioned previously in Section 6.3.3, has a limit state function defined by the ratio of two lognormally-distributed random variables.

$$g(\mathbf{x}) = \frac{x_1}{x_2} \tag{23.15}$$

The distributions for both x_1 and x_2 are Lognormal(1, 0.5) with a correlation coefficient between the two variables of 0.3.

First-order and second-order reliability analysis are performed in the `dakota_logratio.in` and `dakota_logratio_taylor2.in` input files in `Dakota/test`, respectively. For RIA, 24 response levels (.4, .5, .55, .6, .65, .7, .75, .8, .85, .9, 1, 1.05, 1.15, 1.2, 1.25, 1.3, 1.35, 1.4, 1.5, 1.55, 1.6, 1.65, 1.7, and 1.75) are mapped into the corresponding cumulative probability levels. For PMA, these 24 probability levels (the fully converged results from RIA FORM) are mapped back into the original response levels. Figure 23.5 overlays the computed CDF values for a number of first-order reliability method variants as well as a Latin Hypercube reference solution of 10^6 samples.

23.7 Steel Section Example

This test problem is used extensively in [62]. It involves a W16x31 steel section of A36 steel that must carry an applied deterministic bending moment of 1140 kip-in. For DAKOTA, it has been used as a verification test for second-order integrations in reliability methods. The limit state function is defined as:

$$g(\mathbf{x}) = F_y Z - 1140 \tag{23.16}$$

where F_y is Lognormal(38., 3.8), Z is Normal(54., 2.7), and the variables are uncorrelated.

The `Dakota/test/dakota_steel_section.in` input file computes a first-order CDF probability of $p(g \leq 0.)$ = 1.297e-07 and a second-order CDF probability of $p(g \leq 0.)$ = 1.375e-07. This second-order result differs from that reported in [62], since DAKOTA uses the Nataf nonlinear transformation to u-space (see MPP Search Methods section in Reliability Methods chapter of DAKOTA Theory Manual [2]) and [62] uses a linearized transformation.

(a) RIA methods (b) PMA methods

Figure 23.5: Lognormal ratio cumulative distribution function, RIA/PMA methods.

23.8 Portal Frame Example

This test problem is taken from [115, 67]. It involves a plastic collapse mechanism of a simple portal frame. It also has been used as a verification test for second-order integrations in reliability methods. The limit state function is defined as:

$$g(\mathbf{x}) = x_1 + 2x_2 + 2x_3 + x_4 - 5x_5 - 5x_6 \tag{23.17}$$

where $x_1 - x_4$ are Lognormal(120., 12.), x_5 is Lognormal(50., 15.), x_6 is Lognormal(40., 12.), and the variables are uncorrelated.

While the limit state is linear in x-space, the nonlinear transformation of lognormals to u-space induces curvature. The `Dakota/test/dakota_portal_frame.in` input file computes a first-order CDF probability of $p(g \leq 0.)$ = 9.433e-03 and a second-order CDF probability of $p(g \leq 0.)$ = 1.201e-02. These results agree with the published results from the literature.

23.9 Short Column Example

This test problem involves the plastic analysis and design of a short column with rectangular cross section (width b and depth h) having uncertain material properties (yield stress Y) and subject to uncertain loads (bending moment M and axial force P) [74]. The limit state function is defined as:

$$g(\mathbf{x}) = 1 - \frac{4M}{bh^2Y} - \frac{P^2}{b^2h^2Y^2} \tag{23.18}$$

The distributions for P, M, and Y are Normal(500, 100), Normal(2000, 400), and Lognormal(5, 0.5), respectively, with a correlation coefficient of 0.5 between P and M (uncorrelated otherwise). The nominal values for b and h are 5 and 15, respectively.

(a) RIA methods (b) PMA methods

Figure 23.6: Short column cumulative distribution function, RIA/PMA methods.

23.9.1 Uncertainty Quantification

First-order and second-order reliability analysis are performed in the `dakota_short_column.in` and `dakota_short_column_taylor2.in` input files in `Dakota/test`, respectively. For RIA, 43 response levels (-9.0, -8.75, -8.5, -8.0, -7.75, -7.5, -7.25, -7.0, -6.5, -6.0, -5.5, -5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.9, -1.8, -1.7, -1.6, -1.5, -1.4, -1.3, -1.2, -1.1, -1.0, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0.0, 0.05, 0.1, 0.15, 0.2, 0.25) are mapped into the corresponding cumulative probability levels. For PMA, these 43 probability levels (the fully converged results from RIA FORM) are mapped back into the original response levels. Figure 23.6 overlays the computed CDF values for several first-order reliability method variants as well as a Latin Hypercube reference solution of 10^6 samples.

23.9.2 Reliability-Based Design Optimization

The short column example problem is also amenable to RBDO. An objective function of cross-sectional area and a target reliability index of 2.5 (cumulative failure probability $p(g \leq 0) \leq 0.00621$) are used in the design problem:

$$\begin{aligned} \min \quad & bh \\ \text{s.t.} \quad & \beta \geq 2.5 \\ & 5.0 \leq b \leq 15.0 \\ & 15.0 \leq h \leq 25.0 \end{aligned} \tag{23.19}$$

As is evident from the UQ results shown in Figure 23.6, the initial design of $(b, h) = (5, 15)$ is infeasible and the optimization must add material to obtain the target reliability at the optimal design $(b, h) = (8.68, 25.0)$. Simple bi-level, fully analytic bi-level, and sequential RBDO methods are explored in `Dakota/test` inputs `dakota_rbdo_short_column.in`, `dakota_rbdo_short_column_analytic.in`, and `dakota_rbdo_short_column_trsb.in`, with results as described in [24, 25].

Figure 23.7: Cantilever beam test problem.

23.10 Cantilever Example

This test problem is adapted from the reliability-based design optimization literature [107], [125] and involves a simple uniform cantilever beam as shown in Figure 23.7.

The design problem is to minimize the weight (or, equivalently, the cross-sectional area) of the beam subject to a displacement constraint and a stress constraint. Random variables in the problem include the yield stress R of the beam material, the Young's modulus E of the material, and the horizontal and vertical loads, X and Y, which are modeled with normal distributions using $N(40000, 2000)$, $N(2.9E7, 1.45E6)$, $N(500, 100)$, and $N(1000, 100)$, respectively. Problem constants include $L = 100\texttt{in}$ and $D_0 = 2.2535\texttt{in}$. The constraints have the following analytic form:

$$\begin{aligned} \texttt{stress} &= \frac{600}{wt^2}Y + \frac{600}{w^2t}X \leq R \qquad (23.20) \\ \texttt{displacement} &= \frac{4L^3}{Ewt}\sqrt{\left(\frac{Y}{t^2}\right)^2 + \left(\frac{X}{w^2}\right)^2} \leq D_0 \end{aligned}$$

or when scaled:

$$\begin{aligned} g_S &= \frac{\texttt{stress}}{R} - 1 \leq 0 \qquad (23.21) \\ g_D &= \frac{\texttt{displacement}}{D_0} - 1 \leq 0 \qquad (23.22) \end{aligned}$$

23.10.1 Deterministic Optimization Results

If the random variables E, R, X, and Y are fixed at their means, the resulting deterministic design problem can be formulated as

$$\begin{aligned} \texttt{minimize} \quad & f = wt \\ \texttt{subject to} \quad & g_S \leq 0 \qquad (23.23) \\ & g_D \leq 0 \\ & 1.0 \leq w \leq 4.0 \\ & 1.0 \leq t \leq 4.0 \end{aligned}$$

and can be solved using the `Dakota/test/dakota_cantilever.in` file. This input file manages a variety of tests, of which a sample is shown below (`Dakota/examples/methods/dakota_addtnl_cantilever.in`):

```
method,
        npsol_sqp
          convergence_tolerance = 1.e-8

variables,
        continuous_design = 2
          initial_point  4.0   4.0
          upper_bounds  10.0  10.0
          lower_bounds   1.0   1.0
          descriptors'w' 't'
        continuous_state = 4
          initial_state  40000. 29.E+6 500. 1000.
          descriptors      'R'   'E'   'X'   'Y'

interface,
        system
          asynchronous evaluation_concurrency = 2
          analysis_driver = 'cantilever'

responses,
        objective_functions = 1
        nonlinear_inequality_constraints = 2
        numerical_gradients
          method_source dakota
          interval_type forward
          fd_gradient_step_size = 1.e-4
        no_hessians
```

The deterministic solution is $(w, t) = (2.35, 3.33)$ with an objective function of 7.82. The final report from the DAKOTA output is as follows:

```
    <<<<< Iterator npsol_sqp completed.
    <<<<< Function evaluation summary: 33 total (33 new, 0 duplicate)
    <<<<< Best parameters          =
                          2.3520341271e+00 beam_width
                          3.3262784077e+00 beam_thickness
                          4.0000000000e+04 R
                          2.9000000000e+07 E
                          5.0000000000e+02 X
                          1.0000000000e+03 Y
    <<<<< Best objective function  =
                          7.8235203313e+00
    <<<<< Best constraint values   =
                         -1.6009000260e-02
                         -3.7083558446e-11
    <<<<< Best data captured at function evaluation 31
    <<<<< Single Method Strategy completed.
    DAKOTA execution time in seconds:
      Total CPU        =       0.03 [parent =   0.027995, child =   0.002005]
      Total wall clock =   0.281375
```

23.10.2 Stochastic Optimization Results

If the normal distributions for the random variables E, R, X, and Y are included, a stochastic design problem can be formulated as

$$\begin{array}{rl} \texttt{minimize} & f = wt \\ \texttt{subject to} & \beta_D \geq 3 \\ & \beta_S \geq 3 \\ & 1.0 \leq w \leq 4.0 \\ & 1.0 \leq t \leq 4.0 \end{array} \tag{23.24}$$

where a 3-sigma reliability level (probability of failure = 0.00135 if responses are normally-distributed) is being sought on the scaled constraints. Optimization under uncertainty solutions to the stochastic problem are described in [27, 24, 25], for which the solution is $(w, t) = (2.45, 3.88)$ with an objective function of 9.52. This demonstrates that a more conservative design is needed to satisfy the probabilistic constraints.

23.11 Steel Column Example

This test problem involves the trade-off between cost and reliability for a steel column [74]. The cost is defined as

$$Cost = bd + 5h \tag{23.25}$$

where b, d, and h are the means of the flange breadth, flange thickness, and profile height, respectively. Nine uncorrelated random variables are used in the problem to define the yield stress F_s (lognormal with μ/σ = 400/35 MPa), dead weight load P_1 (normal with μ/σ = 500000/50000 N), variable load P_2 (gumbel with μ/σ = 600000/90000 N), variable load P_3 (gumbel with μ/σ = 600000/90000 N), flange breadth B (lognormal with μ/σ = b/3 mm), flange thickness D (lognormal with μ/σ = d/2 mm), profile height H (lognormal with μ/σ = h/5 mm), initial deflection F_0 (normal with μ/σ = 30/10 mm), and Young's modulus E (Weibull with μ/σ = 21000/4200 MPa). The limit state has the following analytic form:

$$g = F_s - P\left(\frac{1}{2BD} + \frac{F_0}{BDH}\frac{E_b}{E_b - P}\right) \tag{23.26}$$

where

$$P = P_1 + P_2 + P_3 \tag{23.27}$$

$$E_b = \frac{\pi^2 EBDH^2}{2L^2} \tag{23.28}$$

and the column length L is 7500 mm.

This design problem (`dakota_rbdo_steel_column.in` in `Dakota/test`) demonstrates design variable insertion into random variable distribution parameters through the design of the mean flange breadth, flange thickness, and profile height. The RBDO formulation maximizes the reliability subject to a cost constraint:

$$\begin{array}{rl} \text{maximize} & \beta \\ \text{subjectto} & Cost \leq 4000. \\ & 200.0 \leq b \leq 400.0 \\ & 10.0 \leq d \leq 30.0 \\ & 100.0 \leq h \leq 500.0 \end{array} \tag{23.29}$$

which has the solution (b, d, h) = (200.0, 17.50, 100.0) with a maximal reliability of 3.132.

23.12 Multiobjective Examples

There are three examples in the test directory (`Dakota/test/dakota_mogatest.in`) that are taken from a multiobjective evolutionary algorithm (MOEA) test suite described by Van Veldhuizen et. al. in [13]. These three problems are good examples to illustrate the different forms that the Pareto set may take. For each problem, we describe the DAKOTA input and show a graph of the Pareto front. These problems are all solved with the `moga` method. In Van Veldhuizen's notation, the set of all Pareto optimal design configurations (design variable values only) is denoted $\mathtt{P}^*$ or $\mathtt{P}_{\mathtt{true}}$ and is defined as:

$$P^* := \{x \in \Omega \,|\, \neg\exists\ x' \in \Omega \quad \bar{f}(x') \preceq \bar{f}(x)\}$$

The Pareto front, which is the set of objective function values associated with the Pareto optimal design configurations, is denoted $\mathtt{PF}^*$ or $\mathtt{PF}_{\mathtt{true}}$ and is defined as:

$$PF^* := \{\bar{u} = \bar{f} = (f_1(x), \ldots, f_k(x)) \,|\, x \in P^*\}$$

The values calculated for the Pareto set and the Pareto front using the moga method are close to but not always exactly the true values, depending on the number of generations the moga is run, the various settings governing the GA, and the complexity of the Pareto set.

23.12.1 Multiobjective Test Problem 1

The first test problem is a case where P_{true} is connected and PF_{true} is concave. The problem is to simultaneously optimize f_1 and f_2 given three input variables, x_1, x_2, and x_3, where the inputs are bounded by $-4 \leq x_i \leq 4$:

$$f_1(x) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i - \frac{1}{\sqrt{3}}\right)^2\right)$$
$$f_2(x) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i + \frac{1}{\sqrt{3}}\right)^2\right)$$

The input file for this example is shown in Figure 23.8 and provided in `Dakota/examples/tutorial/dakota_mogatest1.in`. The interface keyword specifies the use of the `mogatest1` executable (compiled from `Dakota/test/mogatest1.C`) as the simulator. The Pareto front is shown in Figure 23.9. Note that the keyword `final_solutions` specifies the number of best solutions reported by moga. This differs from the entire Pareto front: the top three solutions are simply the best three as ranked by their distance to the utopia point, which is the point of minimum objective value in each dimension.

23.12.2 Multiobjective Test Problem 2

The second test problem is a case where both $\mathtt{P}_{\mathtt{true}}$ and $\mathtt{PF}_{\mathtt{true}}$ are disconnected. $\mathtt{PF}_{\mathtt{true}}$ has four separate Pareto curves. The problem is to simultaneously optimize f_1 and f_2 given two input variables, x_1 and x_2, where the inputs are bounded by $0 \leq x_i \leq 1$, and:

$$\begin{aligned} f_1(x) &= x_1 \\ f_2(x) &= (1 + 10x_2) \times \left[1 - \left(\frac{x_1}{1 + 10x_2}\right)^2 - \frac{x_1}{1 + 10x_2}\sin(8\pi x_1)\right] \end{aligned}$$

```
##  DAKOTA INPUT FILE - dakota_mogatest1.in



strategy,
        single
        graphics tabular_graphics_data

method,
        moga
        output silent
        seed = 10983
        final_solutions = 3
        max_function_evaluations = 2500
        initialization_type unique_random
        crossover_type shuffle_random
          num_offspring = 2 num_parents = 2
          crossover_rate = 0.8
        mutation_type replace_uniform
          mutation_rate = 0.1
        fitness_type domination_count
        replacement_type below_limit = 6
          shrinkage_percentage = 0.9
        convergence_type metric_tracker
          percent_change = 0.05 num_generations = 40

variables,
        continuous_design = 3
          initial_point     0            0       0
          upper_bounds      4    4       4
          lower_bounds     -4   -4      -4
          descriptors      'x1'         'x2'    'x3'

interface,
        system
          analysis_driver = 'mogatest1'

responses,
        objective_functions = 2
        no_gradients
        no_hessians
```

Figure 23.8: DAKOTA input file specifying the use of MOGA on mogatest1

Figure 23.9: Pareto Front showing Tradeoffs between Function F1 and Function F2 for mogatest1

The input file for this example is shown in Figure 23.10 and provided in `Dakota/examples/methods/dakota_mogatest2.in`. It differs from Figure 23.8 in the variables specification, in the use of the `mogatest2` executable (compiled from `Dakota/test/mogatest2.C`) as the simulator, and in the `max_function_evaluations` and `crossover_type` MOGA controls. The Pareto front is shown in Figure 23.11. Note the discontinuous nature of the front in this example.

23.12.3 Multiobjective Test Problem 3

The third test problem is a case where $\mathtt{P_{true}}$ is disconnected but $\mathtt{PF_{true}}$ is connected. It is called the Srinivas problem in the literature (cite). This problem also has two nonlinear constraints. The problem is to simultaneously optimize f_1 and f_2 given two input variables, x_1 and x_2, where the inputs are bounded by $-20 \leq x_i \leq 20$, and:

$$\begin{aligned} f_1(x) &= (x_1 - 2)^2 + (x_2 - 1)^2 + 2 \\ f_2(x) &= 9x_1 - (x_2 - 1)^2 \end{aligned}$$

The constraints are:

$$\begin{aligned} 0 &\leq x_1^2 + x_2^2 - 225 \\ 0 &\leq x_1 - 3x_2 + 10 \end{aligned}$$

The input file for this example is shown in Figure 23.12 and provided in `Dakota/examples/methods/dakota_mogatest3.in`. It differs from Figure 23.10 in the variables and responses specifications, in the use of the `mogatest3` executable (compiled from `Dakota/test/mogatest3.C`) as the simulator, and in the `max_function_evaluations` and `mutation_type` MOGA controls. The Pareto set is shown in Figure 23.13. Note the discontinuous nature of the Pareto set (in the design space) in this example. The Pareto front is shown in Figure 23.14.

23.13 Morris example

Morris [82] includes a screening design test problem with a single-output analytical test function. The output depends on 20 inputs with first- through fourth-order interaction terms, some having large fixed coefficients and others small random coefficients. Thus the function values generated depend on the random number generator employed in the function evaluator. The computational model is:

$$\begin{aligned} y = \beta_0 &+ \sum_{i=1}^{20} \beta_i w_i + \sum_{i<j}^{20} \beta_{i,j} w_i w_j + \sum_{i<j<l}^{20} \beta_{i,j,l} w_i w_j w_l \\ &+ \sum_{i<j<l<s}^{20} \beta_{i,j,l,s} w_i w_j w_l w_s, \end{aligned}$$

where $w_i = 2(x_i - 0.5)$ except for $i = 3, 5,$ and 7, where $w_i = 2(1.1x_i/(x_i + 0.1) - 0.5)$. Large-valued coefficients are assigned as

$$\begin{array}{llll} \beta_i = +20 & i = 1, \ldots, 10; & \beta_{i,j} = -15 & i, j = 1, \ldots, 6; \\ \beta_{i,j,l} = -10 & i, j, l = 1, \ldots, 5; & \beta_{i,j,l,s} = +5 & i, j, l, s = 1, \ldots, 4. \end{array}$$

```
strategy,
        single
        graphics tabular_graphics_data

method,
        moga
        output silent
        seed = 10983
        final_solutions = 3
        max_function_evaluations = 3000
        initialization_type unique_random
        crossover_type
          multi_point_parameterized_binary = 2
          crossover_rate = 0.8
        mutation_type replace_uniform
          mutation_rate = 0.1
        fitness_type domination_count
        replacement_type below_limit = 6
          shrinkage_percentage = 0.9
        convergence_type metric_tracker
          percent_change = 0.05 num_generations = 10

variables,
        continuous_design = 2
          initial_point    0.5          0.5
          upper_bounds     1    1
          lower_bounds     0    0
          descriptors      'x1'         'x2'

interface,
        system
          analysis_driver = 'mogatest2'

responses,
        objective_functions = 2
        no_gradients
        no_hessians
```

Figure 23.10: DAKOTA input file specifying the use of MOGA on mogatest2

Figure 23.11: Pareto Front showing Tradeoffs between Function F1 and Function F2 for mogatest2

```
strategy,
        single
        graphics tabular_graphics_data

method,
        moga
        output silent
        seed = 10983
        final_solutions = 3
        max_function_evaluations = 2000
        initialization_type unique_random
        crossover_type
          multi_point_parameterized_binary = 2
          crossover_rate = 0.8
        mutation_type offset_normal
          mutation_scale = 0.1
        fitness_type domination_count
        replacement_type below_limit = 6
          shrinkage_percentage = 0.9
        convergence_type metric_tracker
          percent_change = 0.05 num_generations = 10

variables,
        continuous_design = 2
          descriptors      'x1'         'x2'
          initial_point      0    0
          upper_bounds      20           20
          lower_bounds     -20  -20

interface,
        system
          analysis_driver = 'mogatest3'

responses,
        objective_functions = 2
        nonlinear_inequality_constraints = 2
        nonlinear_inequality_upper_bounds = 0.0 0.0
        no_gradients
        no_hessians
```

Figure 23.12: DAKOTA input file specifying the use of MOGA on mogatest3

Figure 23.13: Pareto Set of Design Variables corresponding to the Pareto front for mogatest3

Figure 23.14: Pareto Front showing Tradeoffs between Function F1 and Function F2 for mogatest3

The remaining first- and second-order coefficients β_i and $\beta_{i,j}$, respectively, are independently generated from a standard normal distribution (zero mean and unit standard deviation); the remaining third- and fourth-order coefficients are set to zero.

Examination of the test function reveals that one should be able to conclude the following (stated and verified computationally in [97]) for this test problem:

1. the first ten factors are important;
2. of these, the first seven have significant effects involving either interactions or curvatures; and
3. the other three are important mainly because of their first-order effect.

The dakota test input `Dakota/test/dakota_psuade.in` exercises the MOAT algorithm described in Section 5.5 on the Morris problem. The DAKOTA output obtained is shown in Figures 23.15 and 23.16. The MOAT

```
Running MPI executable in serial mode.
DAKOTA version 5.0+ developmental release.
Subversion revision 172 built Dec  8 2010 17:35:13.
Constructing Single Method Strategy...
Writing new restart file dakota.rst
methodName = psuade_moat
gradientType = none
hessianType = none

>>>>> Running Single Method Strategy.

>>>>> Running psuade_moat iterator.

PSUADE DACE method = psuade_moat Samples = 84 Seed (user-specified) = 500
            Partitions = 3 (Levels = 4)
```

Figure 23.15: DAKOTA initialization output for the PSUADE MOAT method on the Morris test problem showing the study parameters.

analysis output reveals that each of the desired observations can be made for the test problem. These are also reflected in Figure 23.17. The modified mean (based on averaging absolute values of elementary effects) shows a clear difference in inputs 1–10 as compared to inputs 11–20. The standard deviation of the (signed) elementary effects indicates correctly that inputs 1–7 have substantial interaction-based or nonlinear effect on the output, while the others have less. While some of inputs 11–20 have nontrivial values of σ, their relatively small modified means μ^* indicate they have little overall influence.

```
>>>>>> PSUADE MOAT output for function 0:

************************************************************
*********************** MOAT Analysis ***********************
------------------------------------------------------------
Input   1 (mod. mean & std) =   9.5329e+01   9.0823e+01
Input   2 (mod. mean & std) =   6.7297e+01   9.5242e+01
Input   3 (mod. mean & std) =   1.0648e+02   1.5479e+02
Input   4 (mod. mean & std) =   6.6231e+01   7.5895e+01
Input   5 (mod. mean & std) =   9.5717e+01   1.2733e+02
Input   6 (mod. mean & std) =   8.0394e+01   9.9959e+01
Input   7 (mod. mean & std) =   3.2722e+01   2.7947e+01
Input   8 (mod. mean & std) =   4.2013e+01   7.6090e+00
Input   9 (mod. mean & std) =   4.1965e+01   7.8535e+00
Input  10 (mod. mean & std) =   3.6809e+01   3.6151e+00
Input  11 (mod. mean & std) =   8.2655e+00   1.0311e+01
Input  12 (mod. mean & std) =   4.9299e+00   7.0591e+00
Input  13 (mod. mean & std) =   3.5455e+00   4.4025e+00
Input  14 (mod. mean & std) =   3.4151e+00   2.4905e+00
Input  15 (mod. mean & std) =   2.5143e+00   5.5168e-01
Input  16 (mod. mean & std) =   9.0344e+00   1.0115e+01
Input  17 (mod. mean & std) =   6.4357e+00   8.3820e+00
Input  18 (mod. mean & std) =   9.1886e+00   2.5373e+00
Input  19 (mod. mean & std) =   2.4105e+00   3.1102e+00
Input  20 (mod. mean & std) =   5.8234e+00   7.2403e+00
<<<<< Function evaluation summary: 84 total (84 new, 0 duplicate)
```

Figure 23.16: DAKOTA analysis output for the PSUADE MOAT method on the Morris problem showing the modified mean and standard deviation of the elementary effect corresponding to each input factor.

Figure 23.17: Standard deviation of elementary effects plotted against modified mean for Morris for each of 20 inputs. Red circles 1–7 correspond to inputs having interactions or nonlinear effects, blue squares 8–10 indicate those with mainly linear effects, and black Xs denote insignificant inputs.

Bibliography

[1] Eclipse.org home. http://www.eclipse.org/. Last accessed: July 6, 2009. 193

[2] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, M. S. Eldred, P. D. Hough, S. Lefantzi, L. P. Swiler, and D. M. Vigil. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.2 theory manual. Technical Report SAND2011-xxxx, Sandia National Laboratories, Albuquerque, NM, 2011. Available online from `http://dakota.sandia.gov/documentation.html`. 53, 103, 107, 111, 112, 139, 146, 216, 217, 220, 221, 305

[3] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, M. S. Eldred, P. D. Hough, S. Lefantzi, L. P. Swiler, and D. M. Vigil. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.2 reference manual. Technical Report SAND2010-2184, Sandia National Laboratories, Albuquerque, NM, Updated Nov. 2011. Available online from `http://dakota.sandia.gov/documentation.html`. 32, 43, 46, 47, 49, 63, 67, 78, 96, 122, 123, 124, 125, 126, 127, 136, 137, 148, 157, 159, 161, 163, 167, 169, 171, 189, 193, 194, 195, 205, 207, 208, 210, 218, 222, 262, 281, 288

[4] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, M. S. Eldred, P. D. Hough, S. Lefantzi, L. P. Swiler, and D. M. Vigil. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.2 developers manual. Technical Report SAND2010-2185, Sandia National Laboratories, Albuquerque, NM, Updated Nov. 2011. Available online from `http://dakota.sandia.gov/documentation.html`. 75, 147, 148, 150, 172, 233, 234

[5] G. Anderson and P. Anderson. *The UNIX C Shell Field Guide*. Prentice-Hall, Englewood Cliffs, NJ, 1986. 30, 175, 177, 225, 287

[6] J. S. Arora. *Introduction to Optimum Design*. McGraw-Hill, New York, 1989. 21

[7] J. M. Aughenbaugh and C. J. J. Paredis. Probability bounds analysis as a general approach to sensitivity analysis in decision making under uncertainty. In *SAE World Congress and Exposition*, number SAE-2007-01-1480, Detroit, MI, 2007. SAE. 213

[8] R. A. Bartlett and L. T. Biegler. rSQP++: An object-oriented framework for successive quadratic programming. *Large-Scale PDE-Constrained Optimization, Springer-Verlag Lecture Notes in Computational Science and Engineering*, 30:316–330, 2003. 256

[9] B. J. Bichon, M. S. Eldred, L. P. Swiler, S. Mahadevan, and J. M. McFarland. Efficient global reliability analysis for nonlinear implicit performance functions. *AIAA Journal*, 46(10):2459–2468, 2008. 105, 106

[10] B. Blinn. *Portable Shell Programming: An Extensive Collection of Bourne Shell Examples*. Prentice Hall PTR, New Jersey, 1996. 225, 287

[11] R. H. Byrd, R. B. Schnabel, and G. A. Schultz. Parallel quasi-newton methods for unconstrained optimization. *Mathematical Programming*, 42:273–306, 1988. 243

[12] K. J. Chang, R. T. Haftka, G. L. Giles, and P.-J. Kao. Sensitivity-based scaling for approximating structural response. *J. Aircraft*, 30:283–288, 1993. 148

[13] C. A. Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic/Plenum Publishers, New York, 2002. 49, 311

[14] P. G. Constantine and M. S. Eldred. Sparse polynomial chaos expansions. *International Journal for Uncertainty Quantification*, in preparation. 112

[15] P. G. Constantine, D. F. Gleich, and G. Iaccarino. Spectral methods for parameterized matrix equations. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2681–2699, 2010. 112

[16] N. Cressie. *Statistics of Spatial Data*. John Wiley and Sons, New York, 1991. 152

[17] J. E. Dennis, D. M. Gay, and R. E. Welsch. ALGORITHM 573: NL2SOL–an adaptive nonlinear least-squares algorithm. *ACM Trans. Math. Software*, 7:369–383, 1981. 41, 70, 137

[18] J. E. Dennis and R. M. Lewis. Problem formulations and other optimization issues in multidisciplinary optimization. In *Proc. AIAA Symposium on Fluid Dynamics*, number AIAA-94-2196, Colorado Springs, Colordao, June 1994. 242

[19] J. E. Dennis and V. J. Torczon. Derivative-free pattern search methods for multidisciplinary design problems. In *Proc. 5th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-94-4349, pages 922–932, Panama City, FL, September 7–9 1994. 126

[20] A. Der Kiureghian and P. L. Liu. Structural reliability under incomplete information. *J. Eng. Mech., ASCE*, 112(EM-1):85–104, 1986. 103

[21] A. Dey and S. Mahadevan. Ductile structural system reliability analysis using adaptive importance sampling. *Structural Safety*, 20:137–154, 1998. 106

[22] Q. Du, V. Faber, and M. Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Review*, 41:637–676, 1999. 91

[23] M. S. Eldred. Optimization strategies for complex engineering applications. Technical Report SAND98-0340, Sandia National Laboratories, Albuquerque, NM, 1998. 19, 173

[24] M. S. Eldred, H. Agarwal, V. M. Perez, Jr. Wojtkiewicz, S. F., and J. E. Renaud. Investigation of reliability method formulations in dakota/uq. *Structure & Infrastructure Engineering: Maintenance, Management, Life-Cycle Design & Performance*, 3(3):199–213, 2007. 95, 217, 307, 310

[25] M. S. Eldred and B. J. Bichon. Second-order reliability formulations in DAKOTA/UQ. In *Proceedings of the 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, number AIAA-2006-1828, Newport, RI, May 1–4 2006. 107, 217, 307, 310

[26] M. S. Eldred, A. A. Giunta, and S. S. Collis. Second-order corrections for surrogate-based optimization with model hierarchies. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY,, Aug. 30–Sept. 1, 2004. AIAA Paper 2004-4457. 142, 148, 149

[27] M. S. Eldred, A. A. Giunta, S. F. Wojtkiewicz Jr., and T. G. Trucano. Formulations for surrogate-based optimization under uncertainty. In *Proc. 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-2002-5585, Atlanta, GA, September 4–6, 2002. 217, 220, 310

[28] M. S. Eldred and W. E. Hart. Design and implementation of multilevel parallel optimization on the Intel TeraFLOPS. In *Proc. 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-98-4707, pages 44–54, St. Louis, MO, September 2–4 1998. 241, 242

[29] M. S. Eldred, W. E. Hart, W. J. Bohnhoff, V. J. Romero, S. A. Hutchinson, and A. G. Salinger. Utilizing object-oriented design to build advanced optimization strategies with generic implementation. In *Proc. 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-96-4164, pages 1568–1582, Bellevue, WA, September 4–6 1996. 173, 242

[30] M. S. Eldred, W. E. Hart, B. D. Schimel, and B. G. van Bloemen Waanders. Multilevel parallelism for optimization on MP computers: Theory and experiment. In *Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-2000-4818, Long Beach, CA, 2000. 254, 256, 258, 261

[31] M. S. Eldred, D. E. Outka, W. J. Bohnhoff, W. R. Witkowski, V. J. Romero, E. R. Ponslet, and K. S. Chen. Optimization of complex mechanics simulations with object-oriented software design. *Computer Modeling and Simulation in Engineering*, 1(3), August 1996. 173

[32] M. S Eldred and L. P. Swiler. Efficient algorithms for mixed aleatory-epistemic uncertainty quantification with application to radiation-hardened electronics. part 1: Algorithms and benchmark results. Technical Report SAND2009-5805, Sandia National Laboratories, Albuquerque, NM, 2009. 74, 222

[33] M. S. Eldred, L. P. Swiler, and G Tang. Mixed aleatory-epistemic uncertainty quantification with stochastic expansions and optimization-based interval estimation. *Reliability Engineering and System Safety*, 2011. Revised paper accepted Oct. 2010, publication in 2011. 74

[34] M. S. Eldred, C. G. Webster, and P. Constantine. Evaluation of non-intrusive approaches for wiener-askey generalized polynomial chaos. In *Proceedings of the 10th AIAA Non-Deterministic Approaches Conference*, number AIAA-2008-1892, Schaumburg, IL, April 7–10 2008. 217

[35] G. M. Fadel, M. F. Riley, and J.-F. M. Barthelemy. Two point exponential approximation method for structural optimization. *Structural Optimization*, 2(2):117–124, 1990. 72, 150

[36] S. Ferson and W. T. Tucker. Sensitivity analysis using probability bounding. *Reliability Engineering and System Safety*, 91:1435–1442, 2006. 213

[37] D. Flaggs. JPrePost user's manual. In preparation. 228

[38] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming, 2nd ed.* Duxbury Press/Brooks/Cole Publishing Co., Pacific Grove, CA, 2003. For small examples, e.g., at most 300 variables, a student version of AMPL suffices; see `http://www.ampl.com/DOWNLOADS`. 169

[39] J. H. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1–141, March 1991. 73, 155

[40] K. Fukuda. cddlib reference manual. Web site, 2005. `ftp://ftp.ifor.math.ethz.ch/pub/fukuda/cdd/cddlibman/cddlibman.html`. 122

[41] J. Gablonsky. Direct version 2.0 userguide technical report. Technical Report CRSC-TR01-08, North Carolina State University, Center for Research in Scientific Computation, Raleigh, NC, 2001. 69, 125

[42] D. M. Gay. Hooking your solver to AMPL. Technical Report Technical Report 97-4-06, Bell Laboratories, Murray Hill, NJ, 1997. Available online as http://www.ampl.com/REFS/HOOKING/index.html and http://www.ampl.com/REFS/hooking2.pdf . 169

[43] R. Ghanem and J. R. Red-Horse. Propagation of probabilistic uncertainty in complex physical systems using a stochastic finite element technique. *Physica D*, 133:137–144, 1999. 67, 95, 111

[44] R. G. Ghanem and P. D. Spanos. *Stochastic Finite Elements: A Spectral Approach*. Springer-Verlag, New York, 1991. 67, 95, 111

[45] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. User's guide for NPSOL (Version 4.0): A Fortran package for nonlinear programming. Technical Report TR SOL-86-2, System Optimization Laboratory, Stanford University, Stanford, CA, 1986. 69, 126

[46] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, San Diego, CA, 1981. 21, 30, 135, 293

[47] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005. 124

[48] D. Gilly. *UNIX in a Nutshell*. O'Reilly and Associates, Inc., Sebastopol, CA, 1992. 204

[49] A. A. Giunta. Use of data sampling, surrogate models, and numerical optimization in engineering design. In *Proc. 40th AIAA Aerospace Science Meeting and Exhibit*, number AIAA-2002-0538, Reno, NV, January 2002. 140

[50] A. A. Giunta and M. S. Eldred. Implementation of a trust region model management strategy in the DAKOTA optimization toolkit. In *Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-2000-4935, Long Beach, CA, September 6–8, 2000. 140, 210

[51] A. A. Giunta, J. M. McFarland, L. P. Swiler, and M. S. Eldred. The promise and peril of uncertainty quantification using respone surface approximations. *Structure and Infrastructure Engineering*, 2(3-4):175–189, September-December 2006. 222, 223

[52] A. A. Giunta and L. T. Watson. A comparison of approximation modeling techniques: Polynomial versus interpolating models. In *Proc. 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-98-4758, pages 392–404, St. Louis, MO, 1998. 73, 153

[53] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wessley Publishing Co., Inc., Reading, MA, 1989. 46

[54] G. H.. Golub and J. H. Welsch. Caclulation of gauss quadrature rules. *Mathematics of Computation*, 23(106):221–230, 1969. 112

[55] I. R. Goodman and H. T. Nguyen. Probability updating using second order probabilities and conditional event algebra. *Information Sciences*, 121:295–347, 1999. 216

[56] G. A. Gray and T. G. Kolda. Algorithm 856: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization. *ACM Transactions on Mathematical Software*, 32(3):485–507, September 2006. 68, 122

[57] J. D Griffin and T. G. Kolda. Nonlinearly-constrained optimization using asynchronous parallel generating set search. Technical Report SAND2007-3257, Sandia National Laboratories, Livermore, CA, 2007. 68, 122

[58] J. D. Griffin, T. G. Kolda, and R. M. Lewis. Asynchronous parallel generating set search for linearly-constrained optimization. *SIAM Journal on Scientific Computing*, 30(4):1892–1924, May 2008. 68, 122

[59] W. Gropp and E. Lusk. User's guide for mpich, a portable implementation of MPI. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, Mathematics and Computer Science Division, 1996. 260

[60] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, Portable Parallel Programing with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1994. 250

[61] R. T. Haftka and Z. Gurdal. *Elements of Structural Optimization*. Kluwer, Boston, 1992. 21, 46

[62] A. Haldar and S. Mahadevan. *Probability, Reliability, and Statistical Methods in Engineering Design*. Wiley, New York, 2000. 21, 67, 161, 305

[63] W. E. Hart. The coliny project. Web site, 2007. `http://software.sandia.gov/Acro/Coliny/`. 43, 46, 47, 122

[64] A. S. Hedayat, N. J. A. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, New York, 1999. 88

[65] J. C. Helton and F. J. Davis. Sampling-based methods for uncertainty and sensitivity analysis. Technical Report SAND99-2240, Sandia National Laboratories, Albuquerque, NM, 2000. 96

[66] J. C. Helton, J. D. Johnson, W. L. Oberkampf, and C. B Storlie. A sampling-based computational strategy for the representation of epistemic uncertainty in model predictions with evidence theory. *Comp. Methods in Applied Mechanics and Engineering*, 196:3980–3998, 2007. 112, 116

[67] H.P. Hong. Simple approximations for improving second-order reliability estimates. *J. Eng. Mech., ASCE*, 125(5):592–595, 1999. 306

[68] D. Huang, T. T. Allen, W. I. Notz, and N. Zeng. Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of Global Optimization*, 34:441–466, 2006. 105, 144

[69] R. L. Iman and M. J Shortencarier. A Fortran 77 program and user's guide for the generation of latin hypercube samples for use with computer models. Technical Report NUREG/CR-3624, SAND83-2365, Sandia National Laboratories, Albuquerque, NM, 1984. 67, 96, 150

[70] D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998. 71, 105, 144, 146

[71] D. R. Karanki, H. S. Kishwaha, A. K. Verma, and S. Ajit. Uncertainty analysis based on probability bounds (p-box) approach in probabilistic safety assessment. *Risk Analysis*, 29:662–675, 2009. 213

[72] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall PTR, Englewood Cliffs, NJ, 2nd edition, 1988. 173, 175, 181, 247

[73] J. R. Koehler and A. B. Owen. Computer experiments. In S. Ghosh and C. R. Rao, editors, *Handbook of Statistics*, volume 13. Elsevier Science, New York, 1996. 87, 152

[74] N. Kuschel and R. Rackwitz. Two basic problems in reliability-based structural optimization. *Math. Method Oper. Res.*, 46:309–333, 1997. 306, 310

[75] L. D. Lathauwer, B. D. Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000. 142

[76] H. Lee, R. Gramacy, C. Linkletter, and G. Gray. Optimization Subject to Hidden Constraints via Statistical Emulation. *Pacific Journal of Optimization*, to appear. 297

[77] R. M. Lewis and S. N. Nash. A multigrid approach to the optimization of systems governed by differential equations. Technical Report AIAA-2000-4890, AIAA, 2000. 148

[78] A. Martelli. *Python in a Nutshell*. O'Reilly and Associates, Cambridge, MA, 2003. 225

[79] J. M. McFarland. *Uncertainty Analysis for Computer Simulations through Validation and Calibration*. PhD thesis, Vanderbilt University, Nashville, Tennesssee, 2008. available for download at http://etd.library.vanderbilt.edu/ETD-db/available/etd-03282008-125137/. 153

[80] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979. 96

[81] J. C. Meza, R. A. Oliva, P. D. Hough, and P. J. Williams. OPT++: an object oriented toolkit for nonlinear optimization. *ACM Transactions on Mathematical Software*, 33(2), 2007. 70, 126, 136

[82] M. D. Morris. Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33(2):161–174, 1991. 91, 314

[83] R. H. Myers and D. C. Montgomery. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. John Wiley & Sons, Inc., New York, 1995. 93, 152

[84] National Research Council of the National Academies. *Evaluation of Quantification of Margins and Uncertainties Methodology for Assessing and Certifying the Reliability of the Nuclear Stockpile*. National Academy Press, Washington D.C., 2008. 216

[85] A. Nealen. A short-as-possible introduction to the least squares, weighted least squares, and moving least squares methods for scattered data approximation and interpolation. Technical report, Discrete Geometric Modeling Group, Technishe Universitaet, Berlin, Germany, 2004. 73, 156

[86] J. Nocedal and Wright S. J. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 1999. 21, 293

[87] W. .L. Oberkampf and J. C. Helton. Evidence theory for engineering applications. Technical Report SAND2003-3559P, Sandia National Laboratories, Albuquerque, NM, 2003. 112, 116

[88] M. J. L. Orr. Introduction to radial basis function networks. Technical report, University of Edinburgh, Edinburgh, Scotland, 1996. 156

[89] A. B. Owen. Orthogonal arrays for computer experiments, integration, and visualization. *Statistica Sinica*, 2(2):439–452, 1992. 90

[90] C. D. Perttunen, D. R. Jones, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *J. Optimization Theory and Application*, 79(1):157–181, 1993. 122

[91] T. D. Plantenga. Hopspack 2.0 user manual. Technical Report SAND2009-6265, Sandia National Laboratories, 2009. 68, 122

[92] E. R. Ponslet and M. S. Eldred. Discrete optimization of isolator locations for vibration isolation systems: an analytical and experimental investigation. In *Proc. 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-96-4178, pages 1703–1716, Bellevue, WA, September 4–6 1996. Also appears as Sandia Technical Report SAND96-1169, May 1996. 123

[93] T. D. Robinson, M. S. Eldred, K. E. Willcox, and R. Haimes. Strategies for multifidelity optimization with variable dimensional hierarchical models. In *Proceedings of the 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (2nd AIAA Multidisciplinary Design Optimization Specialist Conference)*, Newport, RI, May 1–4, 2006. AIAA Paper 2006-1819. 142

[94] T. D. Robinson, K. E. Willcox, M. S. Eldred, and R. Haimes. Multifidelity optimization for variable-complexity design. In *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA, September 6–8, 2006. AIAA Paper 2006-7114. 142

[95] M. Rosenblatt. Remarks on a multivariate transformation. *Annals of Mathematical Statistics*, 23(3):470–472, 1952. 103

[96] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *Computer Journal*, 3:175–184, 1960. 30

[97] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models.* John Wiley & Sons, 2004. 92, 93, 320

[98] K. Schittkowski. *More Test Examples for Nonlinear Programming, Lecture Notes in Economics and Mathematical Systems, Vol. 282*. Springer-Verlag, Berlin, 1987. 293

[99] K. Schittkowski. NLPQLP: A fortran implementation of a sequential quadratic programming algorithm with distributed and non-monotone line search – user's guide. Technical report, Department of Mathematics, University of Bayreuth, Bayreuth, Germany, 2004. 69, 125, 243

[100] P. R. Schunk, P. A. Sackinger, R. R. Rao, K. S. Chen, and R. A. Cairncross. GOMA – a full-newton finite element program for free and moving boundary problems with coupled fluid/solid momentum, energy, mass, and chemical species transport: User's guide. Technical Report SAND95-2937, Sandia National Laboratories, Albuquerque, NM, 1995. 289

[101] G. A. F. Seber and C. J. Wild. *Nonlinear Regression*. Wiley, New Jersey, 2003. 137

[102] I.C. Simpson. Numerical integration over a semi-infinite interval, using the lognormal distibution. *Numerische Mathematik*, 31:71–76, 1978. 112

[103] G. D. Sjaardema. APREPRO: An algebraic preprocessor for parameterizing finite element analyses. Technical Report SAND92-2291, Sandia National Laboratories, Albuquerque, NM, 1992. 164, 165

[104] S.A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl. Akad. Nauk SSSR*, 4:240–243, 1963. 111

[105] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996. 250

[106] A. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice Hall, 1971. 111

[107] R. Sues, M. Aminpour, and Y. Shin. Reliability-based multidisciplinary optimization for aerospace systems. In *Proc. 42rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number AIAA-2001-1521, Seattle, WA, April 16-19 2001. 308

[108] L. P Swiler, R. Slepoy, and A. A. Giunta. Evaluation of sampling methods in constructing response surface approximations. In *Proc. 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number AIAA-2006-1827, Newport, RI, May 1-4 2006. 222

[109] L. P. Swiler and G. D. Wyss. A user's guide to Sandia's latin hypercube sampling software: LHS UNIX library and standalone version. Technical Report SAND04-2439, Sandia National Laboratories, Albuquerque, NM, July 2004. 85, 96, 161

[110] G. Tang, G. Iaccarino, and M. S Eldred. Global sensitivity analysis for stochastic collocation expansion. In *Proceedings of the 51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (12th AIAA Non-Deterministic Approaches conference)*, Orlando, FL, April 12-15, 2010. AIAA Paper 2010-XXXX. 119

[111] G. Tang, L. P. Swiler, and M. S Eldred. Using stochastic expansion methods in evidence theory for mixed aleatory-epistemic uncertainty quantification. In *Proceedings of the 51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (12th AIAA Non-Deterministic Approaches conference)*, Orlando, FL, April 12-15, 2010. AIAA Paper 2010-XXXX. 95

[112] M.A. Tatang. *Direct incorporation of uncertainty in chemical and environmental engineering systems*. PhD thesis, MIT, 1995. 111

[113] C. H. Tong. The psuade software library. Web site, 2005. http://www.llnl.gov/CASC/uncertainty_quantification/#psuade. 66, 85, 91

[114] C. H. Tong and J. C. Meza. DDACE: A distributed object-oriented software with multiple samplings for the design and analysis of computer experiments. Technical Report SAND##-XXXX, Sandia National Laboratories, Livermore, CA. Draft as yet unpublished, see also http://csmr.ca.sandia.gov/projects/ddace/DDACEdoc/html/index.html. 66, 85, 87, 150

[115] L. Tvedt. Distribution of quadratic forms in normal space – applications to structural reliability. *J. Eng. Mech., ASCE*, 116(6):1183–1197, 1990. 306

[116] G. N. Vanderplaats. CONMIN – a FORTRAN program for constrained function minimization. Technical Report TM X-62282, NASA, 1973. See also Addendum to Technical Memorandum, 1978. 36, 69, 123

[117] G. N. Vanderplaats. *Numerical Optimization Techniques for Engineering Design: With Applications*. McGraw-Hill, New York, 1984. 21

[118] Vanderplaats Research and Development, Inc., Colorado Springs, CO. *DOT Users Manual, Version 4.20*, 1995. 69, 124

[119] K. W. Vugrin, L. P. Swiler, R. M. Roberts, N. J. Stuckey-Mack, and S. P Sullivan. Confidence region estimation techniques for nonlinear regression in groundwater flow: Three case studies. *Water Resources Research*, 43, 2007. 137

[120] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Cambridge, 2nd edition, 1996. 225

[121] R. W. Walters. Towards stochastic fluid mechanics via polynomial chaos. In *Proceedings of the 41st AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA-2003-0413, Reno, NV, January 6–9, 2003. 111

[122] B. Walton. BPREPRO preprocessor documentation. Online document http://bwalton.com/bprepro.html. 228

[123] G. Weickum, M. S. Eldred, and K. Maute. Multi-point extended reduced order modeling for design optimization and uncertainty analysis. In *Proceedings of the 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (2nd AIAA Multidisciplinary Design Optimization Specialist Conference)*, Newport, RI, May 1–4, 2006. AIAA Paper 2006-2145. 142

[124] V. G. Weirs, J. R. Kamm, L. P. Swiler, M. Ratto, S. Tarantola, B. M. Adams, W. J. Rider, and M. S Eldred. Sensitivity analysis techniques applied to a system of hyperbolic conservation laws. *Reliability Engineering and System Safety*. submitted Oct 2010. 93

[125] Y.-T. Wu, Y. Shin, R. Sues, and M. Cesare. Safety-factor based approach for probability-based design optimization. In *Proc. 42nd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number AIAA-2001-1522, Seattle, WA, April 16–19 2001. 308

[126] D. Xiu and G. M. Karniadakis. The wiener-askey polynomial chaos for stochastic differential equations. *SIAM J. Sci. Comput.*, 24(2):619–644, 2002. 111

[127] S. Xu and R. V. Grandhi. Effective two-point function approximation for design optimization. *AIAA J.*, 36(12):2269–2275, 1998. 72, 150

[128] D. C. Zimmerman. Genetic algorithms for navigating expensive and complex design spaces, September 1996. Final Report for Sandia National Laboratories contract AO-7736 CA 02. 73, 155

SAND2011-9106
Unlimited Release
Updated December 9, 2011

DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis

Version 5.2 Theory Manual

Brian M. Adams, Keith R. Dalbey, Michael S. Eldred, Laura P. Swiler
Optimization and Uncertainty Quantification Department

William J. Bohnhoff
Radiation Transport Department

John P. Eddy
System Readiness and Sustainment Technologies Department

Dena M. Vigil
Multiphysics Simulation Technologies Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Patricia D. Hough, Sophia Lefantzi
Quantitative Modeling and Analysis Department

Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551

Abstract

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible and extensible interface between simulation codes and iterative analysis methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods; uncertainty quantification with sampling, reliability, and stochastic expansion methods; parameter estimation with nonlinear least squares methods; and sensitivity/variance analysis with design of experiments and parameter study methods. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for design and performance analysis of computational models on high performance computers.

This report serves as a theoretical manual for selected algorithms implemented within the DAKOTA software. It is not intended as a comprehensive theoretical treatment, since a number of existing texts cover general optimization theory, statistical analysis, and other introductory topics. Rather, this manual is intended to summarize a set of DAKOTA-related research publications in the areas of surrogate-based optimization, uncertainty quantification, and optimization under uncertainty that provide the foundation for many of DAKOTA's iterative analysis capabilities.

Contents

1 Reliability Methods **9**

1.1 Local Reliability Methods . . . 9

1.1.1 Mean Value . . . 9

1.1.2 MPP Search Methods . . . 10

1.1.2.1 Limit state approximations . . . 11

1.1.2.2 Probability integrations . . . 13

1.1.2.3 Hessian approximations . . . 14

1.1.2.4 Optimization algorithms . . . 15

1.1.2.5 Warm Starting of MPP Searches . . . 15

1.2 Global Reliability Methods . . . 15

1.2.1 Importance Sampling . . . 15

1.2.2 Efficient Global Optimization . . . 16

1.2.2.1 Gaussian Process Model . . . 17

1.2.2.2 Expected Improvement Function . . . 18

1.2.2.3 Expected Feasibility Function . . . 19

2 Stochastic Expansion Methods **21**

2.1 Orthogonal polynomials . . . 21

2.1.1 Askey scheme . . . 21

2.1.2 Numerically generated orthogonal polynomials . . . 22

2.2 Interpolation polynomials . . . 22

2.2.1 Global value-based . . . 22

2.2.2 Global gradient-enhanced . . . 23

2.2.3 Local value-based . . . 23

2.2.4 Local gradient-enhanced . . . 23

2.3 Generalized Polynomial Chaos . . . 24
2.3.1 Expansion truncation and tailoring . . . 24
2.4 Stochastic Collocation . . . 26
2.4.1 Value-based . . . 26
2.4.2 Gradient-enhanced . . . 27
2.5 Transformations to uncorrelated standard variables . . . 27
2.6 Spectral projection . . . 28
2.6.1 Sampling . . . 29
2.6.2 Tensor product quadrature . . . 29
2.6.3 Smolyak sparse grids . . . 30
2.6.4 Cubature . . . 32
2.7 Linear regression . . . 32
2.8 Analytic moments . . . 33
2.9 Local sensitivity analysis: derivatives with respect to expansion variables . . . 33
2.10 Global sensitivity analysis: variance-based decomposition . . . 34
2.11 Automated Refinement . . . 35
2.11.1 Uniform refinement with unbiased grids . . . 35
2.11.2 Dimension-adaptive refinement with biased grids . . . 36
2.11.3 Goal-oriented dimension-adaptive refinement with greedy adaptation . . . 36
2.12 Multifidelity methods . . . 37

3 Epistemic Methods **41**
3.1 Dempster-Shafer theory of evidence (DSTE) . . . 41

4 Surrogate Models **43**
4.1 Kriging and Gaussian Process Models . . . 43
4.1.1 Kriging & Gaussian Processes: Function Values Only . . . 43
4.1.2 Gradient Enhanced Kriging . . . 48

5 Surrogate-Based Local Minimization **53**
5.1 Iterate acceptance logic . . . 55
5.2 Merit functions . . . 56
5.3 Convergence assessment . . . 57
5.4 Constraint relaxation . . . 57

6 Optimization Under Uncertainty (OUU) 61

6.1 Reliability-Based Design Optimization (RBDO) 61

6.1.1 Bi-level RBDO 61

6.1.2 Sequential/Surrogate-based RBDO 63

6.2 Stochastic Expansion-Based Design Optimization (SEBDO) 63

6.2.1 Stochastic Sensitivity Analysis 63

6.2.1.1 Local sensitivity analysis: first-order probabilistic expansions 64

6.2.1.2 Local sensitivity analysis: zeroth-order combined expansions 65

6.2.1.3 Inputs and outputs 65

6.2.2 Optimization Formulations 66

6.2.2.1 Bi-level SEBDO 66

6.2.2.2 Sequential/Surrogate-Based SEBDO 67

6.2.2.3 Multifidelity SEBDO 67

Chapter 1

Reliability Methods

1.1 Local Reliability Methods

Local reliability methods include the Mean Value method and the family of most probable point (MPP) search methods. Each of these methods is gradient-based, employing local approximations and/or local optimization methods.

1.1.1 Mean Value

The Mean Value method (MV, also known as MVFOSM in [45]) is the simplest, least-expensive reliability method because it estimates the response means, response standard deviations, and all CDF/CCDF response-probability-reliability levels from a single evaluation of response functions and their gradients at the uncertain variable means. This approximation can have acceptable accuracy when the response functions are nearly linear and their distributions are approximately Gaussian, but can have poor accuracy in other situations. The expressions for approximate response mean μ_g, approximate response variance σ_g^2, response target to approximate probability/reliability level mapping ($\bar{z} \rightarrow p, \beta$), and probability/reliability target to approximate response level mapping ($\bar{p}, \bar{\beta} \rightarrow z$) are

$$\mu_g = g(\mu_{\mathbf{x}}) \tag{1.1}$$

$$\sigma_g^2 = \sum_i \sum_j Cov(i,j) \frac{dg}{dx_i}(\mu_{\mathbf{x}}) \frac{dg}{dx_j}(\mu_{\mathbf{x}}) \tag{1.2}$$

$$\bar{z} \rightarrow \beta : \qquad \beta_{\mathrm{CDF}} = \frac{\mu_g - \bar{z}}{\sigma_g}, \quad \beta_{\mathrm{CCDF}} = \frac{\bar{z} - \mu_g}{\sigma_g} \tag{1.3}$$

$$\bar{\beta} \rightarrow z : \qquad z = \mu_g - \sigma_g \bar{\beta}_{\mathrm{CDF}}, \quad z = \mu_g + \sigma_g \bar{\beta}_{\mathrm{CCDF}} \tag{1.4}$$

respectively, where $\mathbf{x}$ are the uncertain values in the space of the original uncertain variables ("x-space"), $g(\mathbf{x})$ is the limit state function (the response function for which probability-response level pairs are needed), and β_{CDF} and β_{CCDF} are the CDF and CCDF reliability indices, respectively.

With the introduction of second-order limit state information, MVSOSM calculates a second-order mean as

$$\mu_g = g(\mu_{\mathbf{x}}) + \frac{1}{2} \sum_i \sum_j Cov(i,j) \frac{d^2 g}{dx_i dx_j}(\mu_{\mathbf{x}}) \tag{1.5}$$

This is commonly combined with a first-order variance (Equation 1.2), since second-order variance involves higher order distribution moments (skewness, kurtosis) [45] which are often unavailable.

The first-order CDF probability $p(g \leq z)$, first-order CCDF probability $p(g > z)$, β_{CDF}, and β_{CCDF} are related to one another through

$$p(g \leq z) = \Phi(-\beta_{\text{CDF}}) \tag{1.6}$$
$$p(g > z) = \Phi(-\beta_{\text{CCDF}}) \tag{1.7}$$
$$\beta_{\text{CDF}} = -\Phi^{-1}(p(g \leq z)) \tag{1.8}$$
$$\beta_{\text{CCDF}} = -\Phi^{-1}(p(g > z)) \tag{1.9}$$
$$\beta_{\text{CDF}} = -\beta_{\text{CCDF}} \tag{1.10}$$
$$p(g \leq z) = 1 - p(g > z) \tag{1.11}$$

where $\Phi()$ is the standard normal cumulative distribution function. A common convention in the literature is to define g in such a way that the CDF probability for a response level z of zero (i.e., $p(g \leq 0)$) is the response metric of interest. DAKOTA is not restricted to this convention and is designed to support CDF or CCDF mappings for general response, probability, and reliability level sequences.

With the Mean Value method, it is possible to obtain importance factors indicating the relative importance of input variables. The importance factors can be viewed as an extension of linear sensitivity analysis combining deterministic gradient information with input uncertainty information, *i.e.* input variable standard deviations. The accuracy of the importance factors is contingent of the validity of the linear approximation used to approximate the true response functions. The importance factors are determined as:

$$ImpFactor_i = (\frac{\sigma_{x_i}}{\sigma_g}\frac{dg}{dx_i}(\mu_{\mathbf{x}}))^2 \tag{1.12}$$

1.1.2 MPP Search Methods

All other local reliability methods solve an equality-constrained nonlinear optimization problem to compute a most probable point (MPP) and then integrate about this point to compute probabilities. The MPP search is performed in uncorrelated standard normal space ("u-space") since it simplifies the probability integration: the distance of the MPP from the origin has the meaning of the number of input standard deviations separating the mean response from a particular response threshold. The transformation from correlated non-normal distributions (x-space) to uncorrelated standard normal distributions (u-space) is denoted as $\mathbf{u} = T(\mathbf{x})$ with the reverse transformation denoted as $\mathbf{x} = T^{-1}(\mathbf{u})$. These transformations are nonlinear in general, and possible approaches include the Rosenblatt [71], Nataf [21], and Box-Cox [10] transformations. The nonlinear transformations may also be linearized, and common approaches for this include the Rackwitz-Fiessler [66] two-parameter equivalent normal and the Chen-Lind [15] and Wu-Wirsching [86] three-parameter equivalent normals. DAKOTA employs the Nataf nonlinear transformation which is suitable for the common case when marginal distributions and a correlation matrix are provided, but full joint distributions are not known[1]. This transformation occurs in the following two steps. To transform between the original correlated x-space variables and correlated standard normals ("z-space"), a CDF matching condition is applied for each of the marginal distributions:

$$\Phi(z_i) = F(x_i) \tag{1.13}$$

where $F()$ is the cumulative distribution function of the original probability distribution. Then, to transform between correlated z-space variables and uncorrelated u-space variables, the Cholesky factor $\mathbf{L}$ of a modified

[1] If joint distributions are known, then the Rosenblatt transformation is preferred.

correlation matrix is used:

$$\mathbf{z} = \mathbf{L}\mathbf{u} \tag{1.14}$$

where the original correlation matrix for non-normals in x-space has been modified to represent the corresponding "warped" correlation in z-space [21].

The forward reliability analysis algorithm of computing CDF/CCDF probability/reliability levels for specified response levels is called the reliability index approach (RIA), and the inverse reliability analysis algorithm of computing response levels for specified CDF/CCDF probability/reliability levels is called the performance measure approach (PMA) [78]. The differences between the RIA and PMA formulations appear in the objective function and equality constraint formulations used in the MPP searches. For RIA, the MPP search for achieving the specified response level $\bar{z}$ is formulated as computing the minimum distance in u-space from the origin to the $\bar{z}$ contour of the limit state response function:

$$\begin{array}{ll} \text{minimize} & \mathbf{u}^T\mathbf{u} \\ \text{subject to} & G(\mathbf{u}) = \bar{z} \end{array} \tag{1.15}$$

and for PMA, the MPP search for achieving the specified reliability/probability level $\bar{\beta}, \bar{p}$ is formulated as computing the minimum/maximum response function value corresponding to a prescribed distance from the origin in u-space:

$$\begin{array}{ll} \text{minimize} & \pm G(\mathbf{u}) \\ \text{subject to} & \mathbf{u}^T\mathbf{u} = \bar{\beta}^2 \end{array} \tag{1.16}$$

where $\mathbf{u}$ is a vector centered at the origin in u-space and $g(\mathbf{x}) \equiv G(\mathbf{u})$ by definition. In the RIA case, the optimal MPP solution $\mathbf{u}^*$ defines the reliability index from $\beta = \pm\|\mathbf{u}^*\|_2$, which in turn defines the CDF/CCDF probabilities (using Equations 1.6-1.7 in the case of first-order integration). The sign of β is defined by

$$G(\mathbf{u}^*) > G(\mathbf{0}) : \beta_{\text{CDF}} < 0, \beta_{\text{CCDF}} > 0 \tag{1.17}$$

$$G(\mathbf{u}^*) < G(\mathbf{0}) : \beta_{\text{CDF}} > 0, \beta_{\text{CCDF}} < 0 \tag{1.18}$$

where $G(\mathbf{0})$ is the median limit state response computed at the origin in u-space[2] (where β_{CDF} = β_{CCDF} = 0 and first-order $p(g \leq z)$ = $p(g > z)$ = 0.5). In the PMA case, the sign applied to $G(\mathbf{u})$ (equivalent to minimizing or maximizing $G(\mathbf{u})$) is similarly defined by $\bar{\beta}$

$$\bar{\beta}_{\text{CDF}} < 0, \bar{\beta}_{\text{CCDF}} > 0 : \text{maximize } G(\mathbf{u}) \tag{1.19}$$

$$\bar{\beta}_{\text{CDF}} > 0, \bar{\beta}_{\text{CCDF}} < 0 : \text{minimize } G(\mathbf{u}) \tag{1.20}$$

and the limit state at the MPP ($G(\mathbf{u}^*)$) defines the desired response level result.

1.1.2.1 Limit state approximations

There are a variety of algorithmic variations that are available for use within RIA/PMA reliability analyses. First, one may select among several different limit state approximations that can be used to reduce computational expense during the MPP searches. Local, multipoint, and global approximations of the limit state are possible. [25] investigated local first-order limit state approximations, and [26] investigated local second-order and multipoint approximations. These techniques include:

[2] It is not necessary to explicitly compute the median response since the sign of the inner product $\langle \mathbf{u}^*, \nabla_{\mathbf{u}} G \rangle$ can be used to determine the orientation of the optimal response with respect to the median response.

1. a single Taylor series per response/reliability/probability level in x-space centered at the uncertain variable means. The first-order approach is commonly known as the Advanced Mean Value (AMV) method:

$$g(\mathbf{x}) \cong g(\mu_{\mathbf{x}}) + \nabla_x g(\mu_{\mathbf{x}})^T (\mathbf{x} - \mu_{\mathbf{x}}) \tag{1.21}$$

and the second-order approach has been named AMV^2:

$$g(\mathbf{x}) \cong g(\mu_{\mathbf{x}}) + \nabla_{\mathbf{x}} g(\mu_{\mathbf{x}})^T (\mathbf{x} - \mu_{\mathbf{x}}) + \frac{1}{2}(\mathbf{x} - \mu_{\mathbf{x}})^T \nabla^2_{\mathbf{x}} g(\mu_{\mathbf{x}})(\mathbf{x} - \mu_{\mathbf{x}}) \tag{1.22}$$

2. same as AMV/AMV^2, except that the Taylor series is expanded in u-space. The first-order option has been termed the u-space AMV method:

$$G(\mathbf{u}) \cong G(\mu_{\mathbf{u}}) + \nabla_u G(\mu_{\mathbf{u}})^T (\mathbf{u} - \mu_{\mathbf{u}}) \tag{1.23}$$

where $\mu_{\mathbf{u}} = T(\mu_{\mathbf{x}})$ and is nonzero in general, and the second-order option has been named the u-space AMV^2 method:

$$G(\mathbf{u}) \cong G(\mu_{\mathbf{u}}) + \nabla_{\mathbf{u}} G(\mu_{\mathbf{u}})^T (\mathbf{u} - \mu_{\mathbf{u}}) + \frac{1}{2}(\mathbf{u} - \mu_{\mathbf{u}})^T \nabla^2_{\mathbf{u}} G(\mu_{\mathbf{u}})(\mathbf{u} - \mu_{\mathbf{u}}) \tag{1.24}$$

3. an initial Taylor series approximation in x-space at the uncertain variable means, with iterative expansion updates at each MPP estimate ($\mathbf{x}^*$) until the MPP converges. The first-order option is commonly known as AMV+:

$$g(\mathbf{x}) \cong g(\mathbf{x}^*) + \nabla_x g(\mathbf{x}^*)^T (\mathbf{x} - \mathbf{x}^*) \tag{1.25}$$

and the second-order option has been named AMV^2+:

$$g(\mathbf{x}) \cong g(\mathbf{x}^*) + \nabla_{\mathbf{x}} g(\mathbf{x}^*)^T (\mathbf{x} - \mathbf{x}^*) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T \nabla^2_{\mathbf{x}} g(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) \tag{1.26}$$

4. same as AMV+/AMV^2+, except that the expansions are performed in u-space. The first-order option has been termed the u-space AMV+ method.

$$G(\mathbf{u}) \cong G(\mathbf{u}^*) + \nabla_u G(\mathbf{u}^*)^T (\mathbf{u} - \mathbf{u}^*) \tag{1.27}$$

and the second-order option has been named the u-space AMV^2+ method:

$$G(\mathbf{u}) \cong G(\mathbf{u}^*) + \nabla_{\mathbf{u}} G(\mathbf{u}^*)^T (\mathbf{u} - \mathbf{u}^*) + \frac{1}{2}(\mathbf{u} - \mathbf{u}^*)^T \nabla^2_{\mathbf{u}} G(\mathbf{u}^*)(\mathbf{u} - \mathbf{u}^*) \tag{1.28}$$

5. a multipoint approximation in x-space. This approach involves a Taylor series approximation in intermediate variables where the powers used for the intermediate variables are selected to match information at the current and previous expansion points. Based on the two-point exponential approximation concept (TPEA, [33]), the two-point adaptive nonlinearity approximation (TANA-3, [91]) approximates the limit state as:

$$g(\mathbf{x}) \cong g(\mathbf{x}_2) + \sum_{i=1}^{n} \frac{\partial g}{\partial x_i}(\mathbf{x}_2) \frac{x_{i,2}^{1-p_i}}{p_i}(x_i^{p_i} - x_{i,2}^{p_i}) + \frac{1}{2}\epsilon(\mathbf{x}) \sum_{i=1}^{n} (x_i^{p_i} - x_{i,2}^{p_i})^2 \tag{1.29}$$

where n is the number of uncertain variables and:

$$p_i = 1 + \ln\left[\frac{\frac{\partial g}{\partial x_i}(\mathbf{x}_1)}{\frac{\partial g}{\partial x_i}(\mathbf{x}_2)}\right] \Big/ \ln\left[\frac{x_{i,1}}{x_{i,2}}\right] \tag{1.30}$$

$$\epsilon(\mathbf{x}) = \frac{H}{\sum_{i=1}^{n}(x_i^{p_i} - x_{i,1}^{p_i})^2 + \sum_{i=1}^{n}(x_i^{p_i} - x_{i,2}^{p_i})^2} \tag{1.31}$$

$$H = 2\left[g(\mathbf{x}_1) - g(\mathbf{x}_2) - \sum_{i=1}^{n} \frac{\partial g}{\partial x_i}(\mathbf{x}_2) \frac{x_{i,2}^{1-p_i}}{p_i}(x_{i,1}^{p_i} - x_{i,2}^{p_i})\right] \tag{1.32}$$

and $\mathbf{x}_2$ and $\mathbf{x}_1$ are the current and previous MPP estimates in x-space, respectively. Prior to the availability of two MPP estimates, x-space AMV+ is used.

6. a multipoint approximation in u-space. The u-space TANA-3 approximates the limit state as:

$$G(\mathbf{u}) \cong G(\mathbf{u}_2) + \sum_{i=1}^{n} \frac{\partial G}{\partial u_i}(\mathbf{u}_2) \frac{u_{i,2}^{1-p_i}}{p_i}(u_i^{p_i} - u_{i,2}^{p_i}) + \frac{1}{2}\epsilon(\mathbf{u}) \sum_{i=1}^{n}(u_i^{p_i} - u_{i,2}^{p_i})^2 \tag{1.33}$$

where:

$$p_i = 1 + \ln\left[\frac{\frac{\partial G}{\partial u_i}(\mathbf{u}_1)}{\frac{\partial G}{\partial u_i}(\mathbf{u}_2)}\right] \Big/ \ln\left[\frac{u_{i,1}}{u_{i,2}}\right] \tag{1.34}$$

$$\epsilon(\mathbf{u}) = \frac{H}{\sum_{i=1}^{n}(u_i^{p_i} - u_{i,1}^{p_i})^2 + \sum_{i=1}^{n}(u_i^{p_i} - u_{i,2}^{p_i})^2} \tag{1.35}$$

$$H = 2\left[G(\mathbf{u}_1) - G(\mathbf{u}_2) - \sum_{i=1}^{n} \frac{\partial G}{\partial u_i}(\mathbf{u}_2) \frac{u_{i,2}^{1-p_i}}{p_i}(u_{i,1}^{p_i} - u_{i,2}^{p_i})\right] \tag{1.36}$$

and $\mathbf{u}_2$ and $\mathbf{u}_1$ are the current and previous MPP estimates in u-space, respectively. Prior to the availability of two MPP estimates, u-space AMV+ is used.

7. the MPP search on the original response functions without the use of any approximations. Combining this option with first-order and second-order integration approaches (see next section) results in the traditional first-order and second-order reliability methods (FORM and SORM).

The Hessian matrices in AMV^2 and AMV^2+ may be available analytically, estimated numerically, or approximated through quasi-Newton updates. The selection between x-space or u-space for performing approximations depends on where the approximation will be more accurate, since this will result in more accurate MPP estimates (AMV, AMV^2) or faster convergence (AMV+, AMV^2+, TANA). Since this relative accuracy depends on the forms of the limit state $g(x)$ and the transformation $T(x)$ and is therefore application dependent in general, DAKOTA supports both options. A concern with approximation-based iterative search methods (i.e., AMV+, AMV^2+ and TANA) is the robustness of their convergence to the MPP. It is possible for the MPP iterates to oscillate or even diverge. However, to date, this occurrence has been relatively rare, and DAKOTA contains checks that monitor for this behavior. Another concern with TANA is numerical safeguarding (e.g., the possibility of raising negative x_i or u_i values to nonintegral p_i exponents in Equations 1.29, 1.31-1.33, and 1.35-1.36). Safeguarding involves offseting negative x_i or u_i and, for potential numerical difficulties with the logarithm ratios in Equations 1.30 and 1.34, reverting to either the linear ($p_i = 1$) or reciprocal ($p_i = -1$) approximation based on which approximation has lower error in $\frac{\partial g}{\partial x_i}(\mathbf{x}_1)$ or $\frac{\partial G}{\partial u_i}(\mathbf{u}_1)$.

1.1.2.2 Probability integrations

The second algorithmic variation involves the integration approach for computing probabilities at the MPP, which can be selected to be first-order (Equations 1.6-1.7) or second-order integration. Second-order integration involves applying a curvature correction [11, 47, 48]. Breitung applies a correction based on asymptotic analysis [11]:

$$p = \Phi(-\beta_p) \prod_{i=1}^{n-1} \frac{1}{\sqrt{1 + \beta_p \kappa_i}} \tag{1.37}$$

where κ_i are the principal curvatures of the limit state function (the eigenvalues of an orthonormal transformation of $\nabla^2_{\mathbf{u}} G$, taken positive for a convex limit state) and $\beta_p \geq 0$ (a CDF or CCDF probability correction is selected to

obtain the correct sign for β_p). An alternate correction in [47] is consistent in the asymptotic regime ($\beta_p \rightarrow \infty$) but does not collapse to first-order integration for $\beta_p = 0$:

$$p = \Phi(-\beta_p) \prod_{i=1}^{n-1} \frac{1}{\sqrt{1 + \psi(-\beta_p)\kappa_i}} \tag{1.38}$$

where $\psi() = \frac{\phi()}{\Phi()}$ and $\phi()$ is the standard normal density function. [48] applies further corrections to Equation 1.38 based on point concentration methods. At this time, all three approaches are available within the code, but the Hohenbichler-Rackwitz correction is used by default (switching the correction is a compile-time option in the source code and has not not currently been exposed in the input specification).

1.1.2.3 Hessian approximations

To use a second-order Taylor series or a second-order integration when second-order information ($\nabla^2_{\mathbf{x}} g$, $\nabla^2_{\mathbf{u}} G$, and/or κ) is not directly available, one can estimate the missing information using finite differences or approximate it through use of quasi-Newton approximations. These procedures will often be needed to make second-order approaches practical for engineering applications.

In the finite difference case, numerical Hessians are commonly computed using either first-order forward differences of gradients using

$$\nabla^2 g(\mathbf{x}) \cong \frac{\nabla g(\mathbf{x} + h\mathbf{e}_i) - \nabla g(\mathbf{x})}{h} \tag{1.39}$$

to estimate the ith Hessian column when gradients are analytically available, or second-order differences of function values using

$$\nabla^2 g(\mathbf{x}) \cong \frac{g(\mathbf{x}+h\mathbf{e}_i+h\mathbf{e}_j) - g(\mathbf{x}+h\mathbf{e}_i-h\mathbf{e}_j) - g(\mathbf{x}-h\mathbf{e}_i+h\mathbf{e}_j) + g(\mathbf{x}-h\mathbf{e}_i-h\mathbf{e}_j)}{4h^2} \tag{1.40}$$

to estimate the ij^{th} Hessian term when gradients are not directly available. This approach has the advantage of locally-accurate Hessians for each point of interest (which can lead to quadratic convergence rates in discrete Newton methods), but has the disadvantage that numerically estimating each of the matrix terms can be expensive.

Quasi-Newton approximations, on the other hand, do not reevaluate all of the second-order information for every point of interest. Rather, they accumulate approximate curvature information over time using secant updates. Since they utilize the existing gradient evaluations, they do not require any additional function evaluations for evaluating the Hessian terms. The quasi-Newton approximations of interest include the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update

$$\mathbf{B}_{k+1} = \mathbf{B}_k - \frac{\mathbf{B}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{B}_k}{\mathbf{s}_k^T \mathbf{B}_k \mathbf{s}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} \tag{1.41}$$

which yields a sequence of symmetric positive definite Hessian approximations, and the Symmetric Rank 1 (SR1) update

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{(\mathbf{y}_k - \mathbf{B}_k \mathbf{s}_k)(\mathbf{y}_k - \mathbf{B}_k \mathbf{s}_k)^T}{(\mathbf{y}_k - \mathbf{B}_k \mathbf{s}_k)^T \mathbf{s}_k} \tag{1.42}$$

which yields a sequence of symmetric, potentially indefinite, Hessian approximations. $\mathbf{B}_k$ is the kth approximation to the Hessian $\nabla^2 g$, $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ is the step and $\mathbf{y}_k = \nabla g_{k+1} - \nabla g_k$ is the corresponding yield in the gradients. The selection of BFGS versus SR1 involves the importance of retaining positive definiteness in the Hessian approximations; if the procedure does not require it, then the SR1 update can be more accurate if the true Hessian is not positive definite. Initial scalings for $\mathbf{B}_0$ and numerical safeguarding techniques (damped BFGS, update skipping) are described in [26].

1.1.2.4 Optimization algorithms

The next algorithmic variation involves the optimization algorithm selection for solving Eqs. 1.15 and 1.16. The Hasofer-Lind Rackwitz-Fissler (HL-RF) algorithm [45] is a classical approach that has been broadly applied. It is a Newton-based approach lacking line search/trust region globalization, and is generally regarded as computationally efficient but occasionally unreliable. DAKOTA takes the approach of employing robust, general-purpose optimization algorithms with provable convergence properties. In particular, we employ the sequential quadratic programming (SQP) and nonlinear interior-point (NIP) optimization algorithms from the NPSOL [40] and OPT++ [57] libraries, respectively.

1.1.2.5 Warm Starting of MPP Searches

The final algorithmic variation for local reliability methods involves the use of warm starting approaches for improving computational efficiency. [25] describes the acceleration of MPP searches through warm starting with approximate iteration increment, with $z/p/\beta$ level increment, and with design variable increment. Warm started data includes the expansion point and associated response values and the MPP optimizer initial guess. Projections are used when an increment in $z/p/\beta$ level or design variables occurs. Warm starts were consistently effective in [25], with greater effectiveness for smaller parameter changes, and are used by default in DAKOTA.

1.2 Global Reliability Methods

Local reliability methods, while computationally efficient, have well-known failure mechanisms. When confronted with a limit state function that is nonsmooth, local gradient-based optimizers may stall due to gradient inaccuracy and fail to converge to an MPP. Moreover, if the limit state is multimodal (multiple MPPs), then a gradient-based local method can, at best, locate only one local MPP solution. Finally, a linear (Eqs. 1.6–1.7) or parabolic (Eqs. 1.37–1.38) approximation to the limit state at this MPP may fail to adequately capture the contour of a highly nonlinear limit state.

A reliability analysis method that is both efficient when applied to expensive response functions and accurate for a response function of any arbitrary shape is needed. This section develops such a method based on efficient global optimization [51] (EGO) to the search for multiple points on or near the limit state throughout the random variable space. By locating multiple points on the limit state, more complex limit states can be accurately modeled, resulting in a more accurate assessment of the reliability. It should be emphasized here that these multiple points exist on a single limit state. Because of its roots in efficient global optimization, this method of reliability analysis is called efficient global reliability analysis (EGRA) [9]. The following two subsections describe two capabilities that are incorporated into the EGRA algorithm: importance sampling and EGO.

1.2.1 Importance Sampling

An alternative to MPP search methods is to directly perform the probability integration numerically by sampling the response function. Sampling methods do not rely on a simplifying approximation to the shape of the limit state, so they can be more accurate than FORM and SORM, but they can also be prohibitively expensive because they generally require a large number of response function evaluations. Importance sampling methods reduce this expense by focusing the samples in the important regions of the uncertain space. They do this by centering the sampling density function at the MPP rather than at the mean. This ensures the samples will lie the region of interest, thus increasing the efficiency of the sampling method. Adaptive importance sampling (AIS) further improves the efficiency by adaptively updating the sampling density function. Multimodal adaptive importance

sampling [22, 93] is a variation of AIS that allows for the use of multiple sampling densities making it better suited for cases where multiple sections of the limit state are highly probable.

Note that importance sampling methods require that the location of at least one MPP be known because it is used to center the initial sampling density. However, current gradient-based, local search methods used in MPP search may fail to converge or may converge to poor solutions for highly nonlinear problems, possibly making these methods inapplicable. As the next section describes, EGO is a global optimization method that does not depend on the availability of accurate gradient information, making convergence more reliable for nonsmooth response functions. Moreover, EGO has the ability to locate multiple failure points, which would provide multiple starting points and thus a good multimodal sampling density for the initial steps of multimodal AIS. The resulting Gaussian process model is accurate in the vicinity of the limit state, thereby providing an inexpensive surrogate that can be used to provide response function samples. As will be seen, using EGO to locate multiple points along the limit state, and then using the resulting Gaussian process model to provide function evaluations in multimodal AIS for the probability integration, results in an accurate and efficient reliability analysis tool.

1.2.2 Efficient Global Optimization

Efficient Global Optimization (EGO) was developed to facilitate the unconstrained minimization of expensive implicit response functions. The method builds an initial Gaussian process model as a global surrogate for the response function, then intelligently selects additional samples to be added for inclusion in a new Gaussian process model in subsequent iterations. The new samples are selected based on how much they are expected to improve the current best solution to the optimization problem. When this expected improvement is acceptably small, the globally optimal solution has been found. The application of this methodology to equality-constrained reliability analysis is the primary contribution of EGRA.

Efficient global optimization was originally proposed by Jones et al. [51] and has been adapted into similar methods such as sequential kriging optimization (SKO) [50]. The main difference between SKO and EGO lies within the specific formulation of what is known as the expected improvement function (EIF), which is the feature that sets all EGO/SKO-type methods apart from other global optimization methods. The EIF is used to select the location at which a new training point should be added to the Gaussian process model by maximizing the amount of improvement in the objective function that can be expected by adding that point. A point could be expected to produce an improvement in the objective function if its predicted value is better than the current best solution, or if the uncertainty in its prediction is such that the probability of it producing a better solution is high. Because the uncertainty is higher in regions of the design space with fewer observations, this provides a balance between exploiting areas of the design space that predict good solutions, and exploring areas where more information is needed.

The general procedure of these EGO-type methods is:

1. Build an initial Gaussian process model of the objective function.

2. Find the point that maximizes the EIF. If the EIF value at this point is sufficiently small, stop.

3. Evaluate the objective function at the point where the EIF is maximized. Update the Gaussian process model using this new point. Go to Step 2.

The following sections discuss the construction of the Gaussian process model used, the form of the EIF, and then a description of how that EIF is modified for application to reliability analysis.

1.2.2.1 Gaussian Process Model

Gaussian process (GP) models are set apart from other surrogate models because they provide not just a predicted value at an unsampled point, but also and estimate of the prediction variance. This variance gives an indication of the uncertainty in the GP model, which results from the construction of the covariance function. This function is based on the idea that when input points are near one another, the correlation between their corresponding outputs will be high. As a result, the uncertainty associated with the model's predictions will be small for input points which are near the points used to train the model, and will increase as one moves further from the training points.

It is assumed that the true response function being modeled $G(\mathbf{u})$ can be described by: [19]

$$G(\mathbf{u}) = \mathbf{h}(\mathbf{u})^T \boldsymbol{\beta} + Z(\mathbf{u}) \tag{1.43}$$

where $\mathbf{h}()$ is the trend of the model, $\boldsymbol{\beta}$ is the vector of trend coefficients, and $Z()$ is a stationary Gaussian process with zero mean (and covariance defined below) that describes the departure of the model from its underlying trend. The trend of the model can be assumed to be any function, but taking it to be a constant value has been reported to be generally sufficient. [72] For the work presented here, the trend is assumed constant and $\boldsymbol{\beta}$ is taken as simply the mean of the responses at the training points. The covariance between outputs of the Gaussian process $Z()$ at points $\mathbf{a}$ and $\mathbf{b}$ is defined as:

$$Cov\left[Z(\mathbf{a}), Z(\mathbf{b})\right] = \sigma_Z^2 R(\mathbf{a}, \mathbf{b}) \tag{1.44}$$

where σ_Z^2 is the process variance and $R()$ is the correlation function. There are several options for the correlation function, but the squared-exponential function is common [72], and is used here for $R()$:

$$R(\mathbf{a}, \mathbf{b}) = \exp\left[-\sum_{i=1}^{d} \theta_i (a_i - b_i)^2\right] \tag{1.45}$$

where d represents the dimensionality of the problem (the number of random variables), and θ_i is a scale parameter that indicates the correlation between the points within dimension i. A large θ_i is representative of a short correlation length.

The expected value $\mu_G()$ and variance $\sigma_G^2()$ of the GP model prediction at point $\mathbf{u}$ are:

$$\mu_G(\mathbf{u}) = \mathbf{h}(\mathbf{u})^T \boldsymbol{\beta} + \mathbf{r}(\mathbf{u})^T \mathbf{R}^{-1} (\mathbf{g} - \mathbf{F}\boldsymbol{\beta}) \tag{1.46}$$

$$\sigma_G^2(\mathbf{u}) = \sigma_Z^2 - \begin{bmatrix} \mathbf{h}(\mathbf{u})^T & \mathbf{r}(\mathbf{u})^T \end{bmatrix} \begin{bmatrix} \mathbf{0} & \mathbf{F}^T \\ \mathbf{F} & \mathbf{R} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{h}(\mathbf{u}) \\ \mathbf{r}(\mathbf{u}) \end{bmatrix} \tag{1.47}$$

where $\mathbf{r}(\mathbf{u})$ is a vector containing the covariance between $\mathbf{u}$ and each of the n training points (defined by Eq. 1.44), $\mathbf{R}$ is an $n \times n$ matrix containing the correlation between each pair of training points, $\mathbf{g}$ is the vector of response outputs at each of the training points, and $\mathbf{F}$ is an $n \times q$ matrix with rows $\mathbf{h}(\mathbf{u}_i)^T$ (the trend function for training point i containing q terms; for a constant trend $q = 1$). This form of the variance accounts for the uncertainty in the trend coefficients $\boldsymbol{\beta}$, but assumes that the parameters governing the covariance function (σ_Z^2 and $\boldsymbol{\theta}$) have known values.

The parameters σ_Z^2 and $\boldsymbol{\theta}$ are determined through maximum likelihood estimation. This involves taking the log of the probability of observing the response values $\mathbf{g}$ given the covariance matrix $\mathbf{R}$, which can be written as: [72]

$$\log\left[p(\mathbf{g}|\mathbf{R})\right] = -\frac{1}{n}\log|\mathbf{R}| - \log(\hat{\sigma}_Z^2) \tag{1.48}$$

where $|\mathbf{R}|$ indicates the determinant of $\mathbf{R}$, and $\hat{\sigma}_Z^2$ is the optimal value of the variance given an estimate of $\boldsymbol{\theta}$ and is defined by:

$$\hat{\sigma}_Z^2 = \frac{1}{n}(\mathbf{g} - \mathbf{F}\boldsymbol{\beta})^T \mathbf{R}^{-1} (\mathbf{g} - \mathbf{F}\boldsymbol{\beta}) \tag{1.49}$$

Maximizing Eq. 1.48 gives the maximum likelihood estimate of $\boldsymbol{\theta}$, which in turn defines σ_Z^2.

1.2.2.2 Expected Improvement Function

The expected improvement function is used to select the location at which a new training point should be added. The EIF is defined as the expectation that any point in the search space will provide a better solution than the current best solution based on the expected values and variances predicted by the GP model. An important feature of the EIF is that it provides a balance between exploiting areas of the design space where good solutions have been found, and exploring areas of the design space where the uncertainty is high. First, recognize that at any point in the design space, the GP prediction $\hat{G}()$ is a Gaussian distribution:

$$\hat{G}(\mathbf{u}) \sim N\left[\mu_G(\mathbf{u}), \sigma_G(\mathbf{u})\right] \tag{1.50}$$

where the mean $\mu_G()$ and the variance $\sigma_G^2()$ were defined in Eqs. 1.46 and 1.47, respectively. The EIF is defined as: [51]

$$EI\big(\hat{G}(\mathbf{u})\big) \equiv E\left[\max\Big(G(\mathbf{u}^*) - \hat{G}(\mathbf{u}), 0\Big)\right] \tag{1.51}$$

where $G(\mathbf{u}^*)$ is the current best solution chosen from among the true function values at the training points (henceforth referred to as simply G^*). This expectation can then be computed by integrating over the distribution $\hat{G}(\mathbf{u})$ with G^* held constant:

$$EI\big(\hat{G}(\mathbf{u})\big) = \int_{-\infty}^{G^*} (G^* - G)\ \hat{G}(\mathbf{u})\ dG \tag{1.52}$$

where G is a realization of $\hat{G}$. This integral can be expressed analytically as: [51]

$$EI\big(\hat{G}(\mathbf{u})\big) = (G^* - \mu_G)\ \Phi\left(\frac{G^* - \mu_G}{\sigma_G}\right) + \sigma_G\, \phi\left(\frac{G^* - \mu_G}{\sigma_G}\right) \tag{1.53}$$

where it is understood that μ_G and σ_G are functions of $\mathbf{u}$.

The point at which the EIF is maximized is selected as an additional training point. With the new training point added, a new GP model is built and then used to construct another EIF, which is then used to choose another new training point, and so on, until the value of the EIF at its maximized point is below some specified tolerance. In Ref. [50] this maximization is performed using a Nelder-Mead simplex approach, which is a local optimization method. Because the EIF is often highly multimodal [51] it is expected that Nelder-Mead may fail to converge to the true global optimum. In Ref. [51], a branch-and-bound technique for maximizing the EIF is used, but was found to often be too expensive to run to convergence. In DAKOTA, an implementation of the DIRECT global optimization algorithm is used [36].

It is important to understand how the use of this EIF leads to optimal solutions. Eq. 1.53 indicates how much the objective function value at $\mathbf{x}$ is expected to be less than the predicted value at the current best solution. Because the GP model provides a Gaussian distribution at each predicted point, expectations can be calculated. Points with good expected values and even a small variance will have a significant expectation of producing a better solution (exploitation), but so will points that have relatively poor expected values and greater variance (exploration).

The application of EGO to reliability analysis, however, is made more complicated due to the inclusion of equality constraints (see Eqs. 1.15-1.16). For inverse reliability analysis, this extra complication is small. The response being modeled by the GP is the objective function of the optimization problem (see Eq. 1.16) and the deterministic constraint might be handled through the use of a merit function, thereby allowing EGO to solve this equality-constrained optimization problem. Here the problem lies in the interpretation of the constraint for multimodal problems as mentioned previously. In the forward reliability case, the response function appears in the constraint rather than the objective. Here, the maximization of the EIF is inappropriate because feasibility is the main concern. This application is therefore a significant departure from the original objective of EGO and requires a new formulation. For this problem, the expected feasibility function is introduced.

1.2.2.3 Expected Feasibility Function

The expected improvement function provides an indication of how much the true value of the response at a point can be expected to be less than the current best solution. It therefore makes little sense to apply this to the forward reliability problem where the goal is not to minimize the response, but rather to find where it is equal to a specified threshold value. The expected feasibility function (EFF) is introduced here to provide an indication of how well the true value of the response is expected to satisfy the equality constraint $G(\mathbf{u}) = \bar{z}$. Inspired by the contour estimation work in [67], this expectation can be calculated in a similar fashion as Eq. 1.52 by integrating over a region in the immediate vicinity of the threshold value $\bar{z} \pm \epsilon$:

$$EF\big(\hat{G}(\mathbf{u})\big) = \int_{z-\epsilon}^{z+\epsilon} \big[\epsilon - |\bar{z} - G|\big]\, \hat{G}(\mathbf{u})\, dG \tag{1.54}$$

where G denotes a realization of the distribution $\hat{G}$, as before. Allowing z^+ and z^- to denote $\bar{z} \pm \epsilon$, respectively, this integral can be expressed analytically as:

$$\begin{aligned} EF\big(\hat{G}(\mathbf{u})\big) = (\mu_G - \bar{z}) & \left[2\,\Phi\left(\frac{\bar{z} - \mu_G}{\sigma_G}\right) - \Phi\left(\frac{z^- - \mu_G}{\sigma_G}\right) - \Phi\left(\frac{z^+ - \mu_G}{\sigma_G}\right)\right] \\ & - \sigma_G \left[2\,\phi\left(\frac{\bar{z} - \mu_G}{\sigma_G}\right) - \phi\left(\frac{z^- - \mu_G}{\sigma_G}\right) - \phi\left(\frac{z^+ - \mu_G}{\sigma_G}\right)\right] \\ & + \quad \epsilon \left[\Phi\left(\frac{z^+ - \mu_G}{\sigma_G}\right) - \Phi\left(\frac{z^- - \mu_G}{\sigma_G}\right)\right] \end{aligned} \tag{1.55}$$

where ϵ is proportional to the standard deviation of the GP predictor ($\epsilon \propto \sigma_G$). In this case, z^-, z^+, μ_G, σ_G, and ϵ are all functions of the location $\mathbf{u}$, while $\bar{z}$ is a constant. Note that the EFF provides the same balance between exploration and exploitation as is captured in the EIF. Points where the expected value is close to the threshold ($\mu_G \approx \bar{z}$) and points with a large uncertainty in the prediction will have large expected feasibility values.

Chapter 2

Stochastic Expansion Methods

This chapter explores two approaches to forming stochastic expansions, the polynomial chaos expansion (PCE), which employs bases of multivariate orthogonal polynomials, and stochastic collocation (SC), which employs bases of multivariate interpolation polynomials. Both approaches capture the functional relationship between a set of output response metrics and a set of input random variables.

2.1 Orthogonal polynomials

2.1.1 Askey scheme

Table 2.1 shows the set of classical orthogonal polynomials which provide an optimal basis for different continuous probability distribution types. It is derived from the family of hypergeometric orthogonal polynomials known as the Askey scheme [6], for which the Hermite polynomials originally employed by Wiener [83] are a subset. The optimality of these basis selections derives from their orthogonality with respect to weighting functions that correspond to the probability density functions (PDFs) of the continuous distributions when placed in a standard form. The density and weighting functions differ by a constant factor due to the requirement that the integral of the PDF over the support range is one.

Table 2.1: Linkage between standard forms of continuous probability distributions and Askey scheme of continuous hyper-geometric polynomials.

Distribution	Density function	Polynomial	Weight function	Support range
Normal	$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$	Hermite $He_n(x)$	$e^{\frac{-x^2}{2}}$	$[-\infty,\infty]$
Uniform	$\frac{1}{2}$	Legendre $P_n(x)$	1	$[-1,1]$
Beta	$\frac{(1-x)^\alpha(1+x)^\beta}{2^{\alpha+\beta+1}B(\alpha+1,\beta+1)}$	Jacobi $P_n^{(\alpha,\beta)}(x)$	$(1-x)^\alpha(1+x)^\beta$	$[-1,1]$
Exponential	e^{-x}	Laguerre $L_n(x)$	e^{-x}	$[0,\infty]$
Gamma	$\frac{x^\alpha e^{-x}}{\Gamma(\alpha+1)}$	Generalized Laguerre $L_n^{(\alpha)}(x)$	$x^\alpha e^{-x}$	$[0,\infty]$

Note that Legendre is a special case of Jacobi for $\alpha=\beta=0$, Laguerre is a special case of generalized Laguerre for $\alpha=0$, $\Gamma(a)$ is the Gamma function which extends the factorial function to continuous values, and $B(a,b)$ is the Beta function defined as $B(a,b)=\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$. Some care is necessary when specifying the α and β parameters

for the Jacobi and generalized Laguerre polynomials since the orthogonal polynomial conventions [1] differ from the common statistical PDF conventions. The former conventions are used in Table 2.1.

2.1.2 Numerically generated orthogonal polynomials

If all random inputs can be described using independent normal, uniform, exponential, beta, and gamma distributions, then Askey polynomials can be directly applied. If correlation or other distribution types are present, then additional techniques are required. One solution is to employ nonlinear variable transformations as described in Section 2.5 such that an Askey basis can be applied in the transformed space. This can be effective as shown in [31], but convergence rates are typically degraded. In addition, correlation coefficients are warped by the nonlinear transformation [21], and simple expressions for these transformed correlation values are not always readily available. An alternative is to numerically generate the orthogonal polynomials (using Gauss-Wigert [73], discretized Stieltjes [37], Chebyshev [37], or Gramm-Schmidt [84] approaches) and then compute their Gauss points and weights (using the Golub-Welsch [44] tridiagonal eigensolution). These solutions are optimal for given random variable sets having arbitrary probability density functions and eliminate the need to induce additional nonlinearity through variable transformations, but performing this process for general joint density functions with correlation is a topic of ongoing research (refer to Section 2.5 for additional details).

2.2 Interpolation polynomials

Interpolation polynomials may be local or global, value-based or gradient-enhanced, and nodal or hierarchical, with a total of six combinations currently implemented: Lagrange (global value-based), Hermite (global gradient-enhanced), piecewise linear spline (local value-based) in nodal and hierarchical formulations, and piecewise cubic spline (local gradient-enhanced) in nodal and hierarchical formulations[1]. The subsections that follow describe the one-dimensional interpolation polynomials for these cases and Section 2.4 describes their use for multivariate interpolation within the stochastic collocation algorithm.

2.2.1 Global value-based

Lagrange polynomials interpolate a set of points in a single dimension using the functional form

$$L_j = \prod_{\substack{k=1 \\ k \neq j}}^{m} \frac{\xi - \xi_k}{\xi_j - \xi_k} \tag{2.1}$$

where it is evident that L_j is 1 at $\xi = \xi_j$, is 0 for each of the points $\xi = \xi_k$, and has order $m - 1$.

For interpolation of a response function R in one dimension over m points, the expression

$$R(\xi) \cong \sum_{j=1}^{m} r(\xi_j)\, L_j(\xi) \tag{2.2}$$

reproduces the response values $r(\xi_j)$ at the interpolation points and smoothly interpolates between these values at other points.

[1] hierarchical formulations, while implemented, are not yet active in release 5.2

2.2.2 Global gradient-enhanced

Hermite interpolation polynomials (not to be confused with Hermite orthogonal polynomials shown in Table 2.1) interpolate both values and derivatives. In our case, we are interested in interpolating values and first derivatives, i.e, gradients. In the gradient-enhanced case, interpolation of a one-dimensional function involves both type 1 and type 2 interpolation polynomials,

$$R(\xi) \cong \sum_{j=1}^{m} \left[r(\xi_j) H_j^{(1)}(\xi) + \frac{dr}{d\xi}(\xi_j) H_j^{(2)}(\xi) \right] \tag{2.3}$$

where the former interpolate a particular value while producing a zero gradient (i^{th} type 1 interpolant produces a value of 1 for the ith collocation point, zero values for all other points, and zero gradients for all points) and the latter interpolate a particular gradient while producing a zero value (i^{th} type 2 interpolant produces a gradient of 1 for the ith collocation point, zero gradients for all other points, and zero values for all points). One-dimensional polynomials satisfying these constraints for general point sets are generated using divided differences as described in [13].

2.2.3 Local value-based

Linear spline basis polynomials define a "hat function," which produces the value of one at its collocation point and decays linearly to zero at its nearest neighbors. In the case where its collocation point corresponds to a domain boundary, then the half interval that extends beyond the boundary is truncated.

For the case of non-equidistant closed points (e.g., Clenshaw-Curtis), the linear spline polynomials are defined as

$$L_j(\xi) = \begin{cases} 1 - \frac{\xi - \xi_j}{\xi_{j-1} - \xi_j} & \text{if } \xi_{j-1} \leq \xi \leq \xi_j \text{ (left half interval)} \\ 1 - \frac{\xi - \xi_j}{\xi_{j+1} - \xi_j} & \text{if } \xi_j < \xi \leq \xi_{j+1} \text{ (right half interval)} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

For the case of equidistant closed points (i.e., Newton-Cotes), this can be simplified to

$$L_j(\xi) = \begin{cases} 1 - \frac{|\xi - \xi_j|}{h} & \text{if } |\xi - \xi_j| \leq h \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

for h defining the half-interval $\frac{b-a}{m-1}$ of the hat function L_j over the range $\xi \in [a, b]$. For the special case of $m = 1$ point, $L_1(\xi) = 1$ for $\xi_1 = \frac{b+a}{2}$ in both cases above.

2.2.4 Local gradient-enhanced

Type 1 cubic spline interpolants are formulated as follows:

$$H_j^{(1)}(\xi) = \begin{cases} t^2(3 - 2t) \;\text{ for }\; t = \frac{\xi - \xi_{j-1}}{\xi_j - \xi_{j-1}} & \text{if } \xi_{j-1} \leq \xi \leq \xi_j \text{ (left half interval)} \\ (t-1)^2(1 + 2t) \;\text{ for }\; t = \frac{\xi - \xi_j}{\xi_{j+1} - \xi_j} & \text{if } \xi_j < \xi \leq \xi_{j+1} \text{ (right half interval)} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

which produce the desired zero-one-zero property for left-center-right values and zero-zero-zero property for left-center-right gradients. Type 2 cubic spline interpolants are formulated as follows:

$$H_j^{(2)}(\xi) = \begin{cases} ht^2(t-1) \text{ for } h = \xi_j - \xi_{j-1}, \ t = \frac{\xi - \xi_{j-1}}{h} & \text{if } \xi_{j-1} \leq \xi \leq \xi_j \text{ (left half interval)} \\ ht(t-1)^2 \text{ for } h = \xi_{j+1} - \xi_j, \ t = \frac{\xi - \xi_j}{h} & \text{if } \xi_j < \xi \leq \xi_{j+1} \text{ (right half interval)} \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

which produce the desired zero-zero-zero property for left-center-right values and zero-one-zero property for left-center-right gradients. For the special case of $m = 1$ point over the range $\xi \in [a, b]$, $H_1^{(1)}(\xi) = 1$ and $H_1^{(2)}(\xi) = \xi$ for $\xi_1 = \frac{b+a}{2}$.

2.3 Generalized Polynomial Chaos

The set of polynomials from 2.1.1 and 2.1.2 are used as an orthogonal basis to approximate the functional form between the stochastic response output and each of its random inputs. The chaos expansion for a response R takes the form

$$R = a_0 B_0 + \sum_{i_1=1}^{\infty} a_{i_1} B_1(\xi_{i_1}) + \sum_{i_1=1}^{\infty} \sum_{i_2=1}^{i_1} a_{i_1 i_2} B_2(\xi_{i_1}, \xi_{i_2}) + \sum_{i_1=1}^{\infty} \sum_{i_2=1}^{i_1} \sum_{i_3=1}^{i_2} a_{i_1 i_2 i_3} B_3(\xi_{i_1}, \xi_{i_2}, \xi_{i_3}) + ... \tag{2.8}$$

where the random vector dimension is unbounded and each additional set of nested summations indicates an additional order of polynomials in the expansion. This expression can be simplified by replacing the order-based indexing with a term-based indexing

$$R = \sum_{j=0}^{\infty} \alpha_j \Psi_j(\boldsymbol{\xi}) \tag{2.9}$$

where there is a one-to-one correspondence between $a_{i_1 i_2 ... i_n}$ and α_j and between $B_n(\xi_{i_1}, \xi_{i_2}, ..., \xi_{i_n})$ and $\Psi_j(\boldsymbol{\xi})$. Each of the $\Psi_j(\boldsymbol{\xi})$ are multivariate polynomials which involve products of the one-dimensional polynomials. For example, a multivariate Hermite polynomial $B(\boldsymbol{\xi})$ of order n is defined from

$$B_n(\xi_{i_1}, ..., \xi_{i_n}) = e^{\frac{1}{2}\boldsymbol{\xi}^T \boldsymbol{\xi}} (-1)^n \frac{\partial^n}{\partial \xi_{i_1} ... \partial \xi_{i_n}} e^{-\frac{1}{2}\boldsymbol{\xi}^T \boldsymbol{\xi}} \tag{2.10}$$

which can be shown to be a product of one-dimensional Hermite polynomials involving an expansion term multi-index t_i^j:

$$B_n(\xi_{i_1}, ..., \xi_{i_n}) = \Psi_j(\boldsymbol{\xi}) = \prod_{i=1}^{n} \psi_{t_i^j}(\xi_i) \tag{2.11}$$

In the case of a mixed basis, the same multi-index definition is employed although the one-dimensional polynomials $\psi_{t_i^j}$ are heterogeneous in type.

2.3.1 Expansion truncation and tailoring

In practice, one truncates the infinite expansion at a finite number of random variables and a finite expansion order

$$R \cong \sum_{j=0}^{P} \alpha_j \Psi_j(\boldsymbol{\xi}) \tag{2.12}$$

Traditionally, the polynomial chaos expansion includes a complete basis of polynomials up to a fixed total-order specification. That is, for an expansion of total order p involving n random variables, the expansion term multi-index defining the set of Ψ_j is constrained by

$$\sum_{i=1}^{n} t_i^j \leq p \tag{2.13}$$

For example, the multidimensional basis polynomials for a second-order expansion over two random dimensions are

$$\begin{aligned}
\Psi_0(\boldsymbol{\xi}) &= \psi_0(\xi_1)\,\psi_0(\xi_2) &= 1 \\
\Psi_1(\boldsymbol{\xi}) &= \psi_1(\xi_1)\,\psi_0(\xi_2) &= \xi_1 \\
\Psi_2(\boldsymbol{\xi}) &= \psi_0(\xi_1)\,\psi_1(\xi_2) &= \xi_2 \\
\Psi_3(\boldsymbol{\xi}) &= \psi_2(\xi_1)\,\psi_0(\xi_2) &= \xi_1^2 - 1 \\
\Psi_4(\boldsymbol{\xi}) &= \psi_1(\xi_1)\,\psi_1(\xi_2) &= \xi_1\xi_2 \\
\Psi_5(\boldsymbol{\xi}) &= \psi_0(\xi_1)\,\psi_2(\xi_2) &= \xi_2^2 - 1
\end{aligned}$$

The total number of terms N_t in an expansion of total order p involving n random variables is given by

$$N_t \;=\; 1 + P \;=\; 1 + \sum_{s=1}^{p} \frac{1}{s!} \prod_{r=0}^{s-1} (n+r) \;=\; \frac{(n+p)!}{n!p!} \tag{2.14}$$

This traditional approach will be referred to as a "total-order expansion."

An important alternative approach is to employ a "tensor-product expansion," in which polynomial order bounds are applied on a per-dimension basis (no total-order bound is enforced) and all combinations of the one-dimensional polynomials are included. That is, the expansion term multi-index defining the set of Ψ_j is constrained by

$$t_i^j \leq p_i \tag{2.15}$$

where p_i is the polynomial order bound for the ith dimension. In this case, the example basis for $p = 2, n = 2$ is

$$\begin{aligned}
\Psi_0(\boldsymbol{\xi}) &= \psi_0(\xi_1)\,\psi_0(\xi_2) &= 1 \\
\Psi_1(\boldsymbol{\xi}) &= \psi_1(\xi_1)\,\psi_0(\xi_2) &= \xi_1 \\
\Psi_2(\boldsymbol{\xi}) &= \psi_2(\xi_1)\,\psi_0(\xi_2) &= \xi_1^2 - 1 \\
\Psi_3(\boldsymbol{\xi}) &= \psi_0(\xi_1)\,\psi_1(\xi_2) &= \xi_2 \\
\Psi_4(\boldsymbol{\xi}) &= \psi_1(\xi_1)\,\psi_1(\xi_2) &= \xi_1\xi_2 \\
\Psi_5(\boldsymbol{\xi}) &= \psi_2(\xi_1)\,\psi_1(\xi_2) &= (\xi_1^2 - 1)\xi_2 \\
\Psi_6(\boldsymbol{\xi}) &= \psi_0(\xi_1)\,\psi_2(\xi_2) &= \xi_2^2 - 1 \\
\Psi_7(\boldsymbol{\xi}) &= \psi_1(\xi_1)\,\psi_2(\xi_2) &= \xi_1(\xi_2^2 - 1) \\
\Psi_8(\boldsymbol{\xi}) &= \psi_2(\xi_1)\,\psi_2(\xi_2) &= (\xi_1^2 - 1)(\xi_2^2 - 1)
\end{aligned}$$

and the total number of terms N_t is

$$N_t \;=\; 1 + P \;=\; \prod_{i=1}^{n} (p_i + 1) \tag{2.16}$$

It is apparent from Eq. 2.16 that the tensor-product expansion readily supports anisotropy in polynomial order for each dimension, since the polynomial order bounds for each dimension can be specified independently. It is also feasible to support anisotropy with total-order expansions, through pruning polynomials that satisfy the

total-order bound but violate individual per-dimension bounds (the number of these pruned polynomials would then be subtracted from Eq. 2.14). Finally, custom tailoring of the expansion form can also be explored, e.g. to closely synchronize with monomial coverage in sparse grids through use of a summation of tensor expansions (see Section 2.6.3). In all cases, the specifics of the expansion are codified in the term multi-index, and subsequent machinery for estimating response values and statistics from the expansion can be performed in a manner that is agnostic to the specific expansion form.

2.4 Stochastic Collocation

The SC expansion is formed as a sum of a set of multidimensional interpolation polynomials, one polynomial per interpolated response quantity (one response value and potentially multiple response gradient components) per unique collocation point.

2.4.1 Value-based

For value-based interpolation in multiple dimensions, a tensor-product of the one-dimensional polynomials described in Section 2.2.1 or Section 2.2.3 is used:

$$R(\boldsymbol{\xi}) \cong \sum_{j_1=1}^{m_{i_1}} \cdots \sum_{j_n=1}^{m_{i_n}} r\left(\xi_{j_1}^{i_1}, \ldots, \xi_{j_n}^{i_n}\right) \left(L_{j_1}^{i_1} \otimes \cdots \otimes L_{j_n}^{i_n}\right) \tag{2.17}$$

where $\boldsymbol{i} = (m_1, m_2, \cdots, m_n)$ are the number of nodes used in the n-dimensional interpolation and $\xi_{j_k}^{i_k}$ indicates the jth point out of i possible collocation points in the kth dimension. This can be simplified to

$$R(\boldsymbol{\xi}) \cong \sum_{j=1}^{N_p} r_j \boldsymbol{L}_j(\boldsymbol{\xi}) \tag{2.18}$$

where N_p is the number of unique collocation points in the multidimensional grid. The multidimensional interpolation polynomials are defined as

$$\boldsymbol{L}_j(\boldsymbol{\xi}) = \prod_{k=1}^{n} L_{c_k^j}(\xi_k) \tag{2.19}$$

where c_k^j is a collocation multi-index (similar to the expansion term multi-index in Eq. 2.11) that maps from the jth unique collocation point to the corresponding multidimensional indices within the tensor grid, and we have dropped the superscript notation indicating the number of nodes in each dimension for simplicity. The tensor-product structure preserves the desired interpolation properties where the jth multivariate interpolation polynomial assumes the value of 1 at the jth point and assumes the value of 0 at all other points, thereby reproducing the response values at each of the collocation points and smoothly interpolating between these values at other unsampled points.

Multivariate interpolation on Smolyak sparse grids involves a weighted sum of the tensor products in Eq. 2.17 with varying $\boldsymbol{i}$ levels. For sparse interpolants based on nested quadrature rules (e.g., Clenshaw-Curtis, Gauss-Patterson, Genz-Keister), the inteprolation property is preserved, but sparse interpolants based on non-nested rules may exhibit some interpolation error at the collocation points.

2.4.2 Gradient-enhanced

For gradient-enhanced interpolation in multiple dimensions, we extend the formulation in Eq 2.18 to use a tensor-product of the one-dimensional type 1 and type 2 polynomials described in Section 2.2.2 or Section 2.2.4:

$$R(\boldsymbol{\xi}) \cong \sum_{j=1}^{N_p} \left[r_j \boldsymbol{H}_j^{(1)}(\boldsymbol{\xi}) + \sum_{k=1}^{n} \frac{dr_j}{d\xi_k} \boldsymbol{H}_{jk}^{(2)}(\boldsymbol{\xi}) \right] \tag{2.20}$$

The multidimensional type 1 basis polynomials are

$$\boldsymbol{H}_j^{(1)}(\boldsymbol{\xi}) = \prod_{k=1}^{n} H_{c_k^j}^{(1)}(\xi_k) \tag{2.21}$$

where c_k^j is the same collocation multi-index described for Eq. 2.19 and the superscript notation indicating the number of nodes in each dimension has again been omitted. The multidimensional type 2 basis polynomials for the kth gradient component are the same as the type 1 polynomials for each dimension except k:

$$\boldsymbol{H}_{jk}^{(2)}(\boldsymbol{\xi}) = H_{c_k^j}^{(2)}(\xi_k) \prod_{\substack{l=1 \\ l \neq k}}^{n} H_{c_l^j}^{(1)}(\xi_l) \tag{2.22}$$

As for the value-based case, multivariate interpolation on Smolyak sparse grids involves a weighted sum of the tensor products in Eq. 2.20 with varying $\boldsymbol{i}$ levels.

2.5 Transformations to uncorrelated standard variables

Polynomial chaos and stochastic collocation are expanded using polynomials that are functions of independent standard random variables $\boldsymbol{\xi}$. Thus, a key component of either approach is performing a transformation of variables from the original random variables $\boldsymbol{x}$ to independent standard random variables $\boldsymbol{\xi}$ and then applying the stochastic expansion in the transformed space. This notion of independent standard space is extended over the notion of "u-space" used in reliability methods (see Section 1.1.2) in that it extends the standardized set beyond standard normals. For distributions that are already independent, three different approaches are of interest:

1. *Extended basis:* For each Askey distribution type, employ the corresponding Askey basis (Table 2.1). For non-Askey types, numerically generate an optimal polynomial basis for each independent distribution as described in Section 2.1.2. With usage of the optimal basis corresponding to each of the random variable types, we can exploit basis orthogonality under expectation (e.g., Eq. 2.25) without requiring a transformation of variables, thereby avoiding inducing additional nonlinearity that could slow convergence.

2. *Askey basis:* For non-Askey types, perform a nonlinear variable transformation from a given input distribution to the most similar Askey basis. For example, lognormal distributions might employ a Hermite basis in a transformed standard normal space and loguniform, triangular, and histogram distributions might employ a Legendre basis in a transformed standard uniform space. All distributions then employ the Askey orthogonal polynomials and their associated Gauss points/weights.

3. *Wiener basis:* For non-normal distributions, employ a nonlinear variable transformation to standard normal distributions. All distributions then employ the Hermite orthogonal polynomials and their associated Gauss points/weights.

For dependent distributions, we must first perform a nonlinear variable transformation to uncorrelated standard normal distributions, due to the independence of decorrelated standard normals. This involves the Nataf transformation, described in the following paragraph. We then have the following choices:

1. *Single transformation:* Following the Nataf transformation to independent standard normal distributions, employ the Wiener basis in the transformed space.
2. *Double transformation:* From independent standard normal space, transform back to either the original marginal distributions or the desired Askey marginal distributions and employ an extended or Askey basis, respectively, in the transformed space. Independence is maintained, but the nonlinearity of the Nataf transformation is at least partially mitigated.

DAKOTA currently supports single transformations for dependent variables in combination with an Askey basis for independent variables.

The transformation from correlated non-normal distributions to uncorrelated standard normal distributions is denoted as $\boldsymbol{\xi} = T(\mathbf{x})$ with the reverse transformation denoted as $\mathbf{x} = T^{-1}(\boldsymbol{\xi})$. These transformations are nonlinear in general, and possible approaches include the Rosenblatt [71], Nataf [21], and Box-Cox [10] transformations. The results in this paper employ the Nataf transformation, which is suitable for the common case when marginal distributions and a correlation matrix are provided, but full joint distributions are not known[2]. The Nataf transformation occurs in the following two steps. To transform between the original correlated x-space variables and correlated standard normals ("z-space"), a CDF matching condition is applied for each of the marginal distributions:

$$\Phi(z_i) = F(x_i) \tag{2.23}$$

where $\Phi()$ is the standard normal cumulative distribution function and $F()$ is the cumulative distribution function of the original probability distribution. Then, to transform between correlated z-space variables and uncorrelated ξ-space variables, the Cholesky factor $\mathbf{L}$ of a modified correlation matrix is used:

$$\mathbf{z} = \mathbf{L}\boldsymbol{\xi} \tag{2.24}$$

where the original correlation matrix for non-normals in x-space has been modified to represent the corresponding "warped" correlation in z-space [21].

2.6 Spectral projection

The major practical difference between PCE and SC is that, in PCE, one must estimate the coefficients for known basis functions, whereas in SC, one must form the interpolants for known coefficients. PCE estimates its coefficients using either spectral projection or linear regression, where the former approach involves numerical integration based on random sampling, tensor-product quadrature, Smolyak sparse grids, or cubature methods. In SC, the multidimensional interpolants need to be formed over structured data sets, such as point sets from quadrature or sparse grids; approaches based on random sampling may not be used.

The spectral projection approach projects the response against each basis function using inner products and employs the polynomial orthogonality properties to extract each coefficient. Similar to a Galerkin projection, the residual error from the approximation is rendered orthogonal to the selected basis. From Eq. 2.12, taking the inner product of both sides with respect to Ψ_j and enforcing orthogonality yields:

$$\alpha_j \;=\; \frac{\langle R, \Psi_j \rangle}{\langle \Psi_j^2 \rangle} \;=\; \frac{1}{\langle \Psi_j^2 \rangle} \int_\Omega R\, \Psi_j\, \varrho(\boldsymbol{\xi})\, d\boldsymbol{\xi}, \tag{2.25}$$

[2] If joint distributions are known, then the Rosenblatt transformation is preferred.

where each inner product involves a multidimensional integral over the support range of the weighting function. In particular, $\Omega = \Omega_1 \otimes \cdots \otimes \Omega_n$, with possibly unbounded intervals $\Omega_j \subset \mathbb{R}$ and the tensor product form $\varrho(\boldsymbol{\xi}) = \prod_{i=1}^{n} \varrho_i(\xi_i)$ of the joint probability density (weight) function. The denominator in Eq. 2.25 is the norm squared of the multivariate orthogonal polynomial, which can be computed analytically using the product of univariate norms squared

$$\langle \Psi_j^2 \rangle \;=\; \prod_{i=1}^{n} \langle \psi_{t_i^j}^2 \rangle \tag{2.26}$$

where the univariate inner products have simple closed form expressions for each polynomial in the Askey scheme [1] and are readily computed as part of the numerically-generated solution procedures described in Section 2.1.2. Thus, the primary computational effort resides in evaluating the numerator, which is evaluated numerically using sampling, quadrature, cubature, or sparse grid approaches (and this numerical approximation leads to use of the term "pseudo-spectral" by some investigators).

2.6.1 Sampling

In the sampling approach, the integral evaluation is equivalent to computing the expectation (mean) of the response-basis function product (the numerator in Eq. 2.25) for each term in the expansion when sampling within the density of the weighting function. This approach is only valid for PCE and since sampling does not provide any particular monomial coverage guarantee, it is common to combine this coefficient estimation approach with a total-order chaos expansion.

In computational practice, coefficient estimations based on sampling benefit from first estimating the response mean (the first PCE coefficient) and then removing the mean from the expectation evaluations for all subsequent coefficients. While this has no effect for quadrature/sparse grid methods (see following two sections) and little effect for fully-resolved sampling, it does have a small but noticeable beneficial effect for under-resolved sampling.

2.6.2 Tensor product quadrature

In quadrature-based approaches, the simplest general technique for approximating multidimensional integrals, as in Eq. 2.25, is to employ a tensor product of one-dimensional quadrature rules. Since there is little benefit to the use of nested quadrature rules in the tensor-product case[3], we choose Gaussian abscissas, i.e. the zeros of polynomials that are orthogonal with respect to a density function weighting, e.g. Gauss-Hermite, Gauss-Legendre, Gauss-Laguerre, generalized Gauss-Laguerre, Gauss-Jacobi, or numerically-generated Gauss rules.

We first introduce an index $i \in \mathbb{N}_+$, $i \geq 1$. Then, for each value of i, let $\{\xi_1^i, \ldots, \xi_{m_i}^i\} \subset \Omega_i$ be a sequence of abscissas for quadrature on Ω_i. For $f \in C^0(\Omega_i)$ and $n = 1$ we introduce a sequence of one-dimensional quadrature operators

$$\mathscr{U}^i(f)(\xi) = \sum_{j=1}^{m_i} f(\xi_j^i)\, w_j^i, \tag{2.27}$$

with $m_i \in \mathbb{N}$ given. When utilizing Gaussian quadrature, Eq. 2.27 integrates exactly all polynomials of degree less than $2m_i - 1$, for each $i = 1, \ldots, n$. Given an expansion order p, the highest order coefficient evaluations (Eq. 2.25) can be assumed to involve integrands of at least polynomial order $2p$ (Ψ of order p and R modeled to order p) in each dimension such that a minimal Gaussian quadrature order of $p+1$ will be required to obtain good accuracy in these coefficients.

[3] Unless a refinement procedure is in use.

Now, in the multivariate case $n > 1$, for each $f \in C^0(\Omega)$ and the multi-index $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbb{N}_+^n$ we define the full tensor product quadrature formulas

$$\mathcal{Q}_{\mathbf{i}}^n f(\xi) = \left(\mathscr{U}^{i_1} \otimes \cdots \otimes \mathscr{U}^{i_n}\right)(f)(\boldsymbol{\xi}) = \sum_{j_1=1}^{m_{i_1}} \cdots \sum_{j_n=1}^{m_{i_n}} f\left(\xi_{j_1}^{i_1}, \ldots, \xi_{j_n}^{i_n}\right) \left(w_{j_1}^{i_1} \otimes \cdots \otimes w_{j_n}^{i_n}\right). \tag{2.28}$$

Clearly, the above product needs $\prod_{j=1}^n m_{i_j}$ function evaluations. Therefore, when the number of input random variables is small, full tensor product quadrature is a very effective numerical tool. On the other hand, approximations based on tensor product grids suffer from the *curse of dimensionality* since the number of collocation points in a tensor grid grows exponentially fast in the number of input random variables. For example, if Eq. 2.28 employs the same order for all random dimensions, $m_{i_j} = m$, then Eq. 2.28 requires m^n function evaluations.

In [27], it is demonstrated that close synchronization of expansion form with the monomial resolution of a particular numerical integration technique can result in significant performance improvements. In particular, the traditional approach of exploying a total-order PCE (Eqs. 2.13–2.14) neglects a significant portion of the monomial coverage for a tensor-product quadrature approach, and one should rather employ a tensor-product PCE (Eqs. 2.15–2.16) to provide improved synchronization and more effective usage of the Gauss point evaluations. When the quadrature points are standard Gauss rules (i.e., no Clenshaw-Curtis, Gauss-Patterson, or Genz-Keister nested rules), it has been shown that tensor-product PCE and SC result in identical polynomial forms [18], completely eliminating a performance gap that exists between total-order PCE and SC [27].

2.6.3 Smolyak sparse grids

If the number of random variables is moderately large, one should rather consider sparse tensor product spaces as first proposed by Smolyak [74] and further investigated by Refs. [38, 7, 35, 90, 59, 60] that reduce dramatically the number of collocation points, while preserving a high level of accuracy.

Here we follow the notation and extend the description in Ref. [59] to describe the Smolyak *isotropic* formulas $\mathscr{A}(\mathrm{w}, n)$, where w is a level that is independent of dimension[4]. The Smolyak formulas are just linear combinations of the product formulas in Eq. 2.28 with the following key property: only products with a relatively small number of points are used. With $\mathscr{U}^0 = 0$ and for $i \geq 1$ define

$$\Delta^i = \mathscr{U}^i - \mathscr{U}^{i-1}. \tag{2.29}$$

and we set $|\mathbf{i}| = i_1 + \cdots + i_n$. Then the isotropic Smolyak quadrature formula is given by

$$\mathscr{A}(\mathrm{w}, n) = \sum_{|\mathbf{i}| \leq \mathrm{w}+n} \left(\Delta^{i_1} \otimes \cdots \otimes \Delta^{i_n}\right). \tag{2.30}$$

Equivalently, formula Eq. 2.30 can be written as [82]

$$\mathscr{A}(\mathrm{w}, n) = \sum_{\mathrm{w}+1 \leq |\mathbf{i}| \leq \mathrm{w}+n} (-1)^{\mathrm{w}+n-|\mathbf{i}|} \binom{n-1}{\mathrm{w}+n-|\mathbf{i}|} \cdot \left(\mathscr{U}^{i_1} \otimes \cdots \otimes \mathscr{U}^{i_n}\right). \tag{2.31}$$

For each index set $\mathbf{i}$ of levels, linear or nonlinear growth rules are used to define the corresponding one-dimensional quadrature orders. The following growth rules are employed for indices $i \geq 1$, where closed and open refer to the

[4] Other common formulations use a dimension-dependent level q where $q \geq n$. We use $w = q - n$, where $w \geq 0$ for all n.

inclusion and exclusion of the bounds within an interval, respectively:

$$\text{closed nonlinear}: \; m = \begin{cases} 1 & i = 1 \\ 2^{i-1} + 1 & i > 1 \end{cases} \tag{2.32}$$

$$\text{open nonlinear}: \; m = 2^i - 1 \tag{2.33}$$

$$\text{open linear}: \; m = 2i - 1 \tag{2.34}$$

Nonlinear growth rules are used for fully nested rules (e.g., Clenshaw-Curtis is closed fully nested and Gauss-Patterson is open fully nested), and linear growth rules are best for standard Gauss rules that take advantage of, at most, "weak" nesting (e.g., reuse of the center point).

Examples of isotropic sparse grids, constructed from the fully nested Clenshaw-Curtis abscissas and the weakly-nested Gaussian abscissas are shown in Figure 2.1, where $\Omega = [-1, 1]^2$ and both Clenshaw-Curtis and Gauss-Legendre employ nonlinear growth[5] from Eqs. 2.32 and 2.33, respectively. There, we consider a two-dimensional parameter space and a maximum level $\mathrm{w} = 5$ (sparse grid $\mathscr{A}(5, 2)$). To see the reduction in function evaluations with respect to full tensor product grids, we also include a plot of the corresponding Clenshaw-Curtis isotropic full tensor grid having the same maximum number of points in each direction, namely $2^{\mathrm{w}} + 1 = 33$.

Figure 2.1: Two-dimensional grid comparison with a tensor product grid using Clenshaw-Curtis points (left) and sparse grids $\mathscr{A}(5, 2)$ utilizing Clenshaw-Curtis (middle) and Gauss-Legendre (right) points with nonlinear growth.

In [27], it is demonstrated that the synchronization of total-order PCE with the monomial resolution of a sparse grid is imperfect, and that sparse grid SC consistently outperforms sparse grid PCE when employing the sparse grid to directly evaluate the integrals in Eq. 2.25. In our DAKOTA implementation, we depart from the use of sparse integration of total-order expansions, and instead employ a linear combination of tensor expansions [17]. That is, we compute separate tensor polynomial chaos expansions for each of the underlying tensor quadrature grids (for which there is no synchronization issue) and then sum them using the Smolyak combinatorial coefficient (from Eq. 2.31 in the isotropic case). This improves accuracy, preserves the PCE/SC consistency property described in Section 2.6.2, and also simplifies PCE for the case of anisotropic sparse grids described next.

For anisotropic Smolyak sparse grids, a dimension preference vector is used to emphasize important stochastic dimensions. Given a mechanism for defining anisotropy, we can extend the definition of the sparse grid from that of Eq. 2.31 to weight the contributions of different index set components. First, the sparse grid index set constraint

[5] We prefer linear growth for Gauss-Legendre, but employ nonlinear growth here for purposes of comparison.

becomes

$$\mathrm{w}\underline{\gamma} < \mathbf{i} \cdot \gamma \leq \mathrm{w}\underline{\gamma} + |\gamma| \tag{2.35}$$

where $\underline{\gamma}$ is the minimum of the dimension weights γ_k, k = 1 to n. The dimension weighting vector γ amplifies the contribution of a particular dimension index within the constraint, and is therefore inversely related to the dimension preference (higher weighting produces lower index set levels). For the isotropic case of all $\gamma_k = 1$, it is evident that you reproduce the isotropic index constraint $\mathrm{w} + 1 \leq |\mathbf{i}| \leq \mathrm{w} + n$ (note the change from $<$ to $\leq$). Second, the combinatorial coefficient for adding the contribution from each of these index sets is modified as described in [12].

2.6.4 Cubature

Cubature rules [75, 89] are specifically optimized for multidimensional integration and are distinct from tensor-products and sparse grids in that they are not based on combinations of one-dimensional Gauss quadrature rules. They have the advantage of improved scalability to large numbers of random variables, but are restricted in integrand order and require homogeneous random variable sets (achieved via transformation). For example, optimal rules for integrands of 2, 3, and 5 and either Gaussian or uniform densities allow low-order polynomial chaos expansions ($p = 1$ or 2) that are useful for global sensitivity analysis including main effects and, for $p = 2$, all two-way interactions.

2.7 Linear regression

The linear regression approach uses a single linear least squares solution of the form:

$$\boldsymbol{\Psi}\boldsymbol{\alpha} = \boldsymbol{R} \tag{2.36}$$

to solve for the complete set of PCE coefficients $\boldsymbol{\alpha}$ that best match a set of response values $\boldsymbol{R}$. The set of response values is obtained either by performing a design of computer experiments within the density function of $\boldsymbol{\xi}$ (point collocation [81, 49]) or from a subset of tensor quadrature points with highest product weight (probabilistic collocation [77]). In either case, each row of the matrix $\boldsymbol{\Psi}$ contains the N_t multivariate polynomial terms Ψ_j evaluated at a particular $\boldsymbol{\xi}$ sample. An over-sampling is recommended in the case of random samples ([49] recommends $2N_t$ samples), resulting in a least squares solution for the over-determined system. As for sampling-based coefficient estimation, this approach is only valid for PCE and does not require synchronization with monomial coverage; thus it is common to combine this coefficient estimation approach with a traditional total-order chaos expansion in order to keep sampling requirements low. In this case, simulation requirements for this approach scale as $\frac{r(n+p)!}{n!p!}$ (r is an over-sampling factor with typical values $1 \leq r \leq 2$), which can be significantly more affordable than isotropic tensor-product quadrature (scales as $(p+1)^n$ for standard Gauss rules) for larger problems. Finally, additional regression equations can be obtained through the use of derivative information (gradients and Hessians) from each collocation point, which can aid in scaling with respect to the number of random variables, particularly for adjoint-based derivative approaches.

2.8 Analytic moments

Mean and covariance of polynomial chaos expansions are available in simple closed form:

$$\mu_i = \langle R_i \rangle \cong \sum_{k=0}^{P} \alpha_{ik} \langle \Psi_k(\boldsymbol{\xi}) \rangle = \alpha_{i0} \tag{2.37}$$

$$\Sigma_{ij} = \langle (R_i - \mu_i)(R_j - \mu_j) \rangle \cong \sum_{k=1}^{P} \sum_{l=1}^{P} \alpha_{ik} \alpha_{jl} \langle \Psi_k(\boldsymbol{\xi}) \Psi_l(\boldsymbol{\xi}) \rangle = \sum_{k=1}^{P} \alpha_{ik} \alpha_{jk} \langle \Psi_k^2 \rangle \tag{2.38}$$

where the norm squared of each multivariate polynomial is computed from Eq. 2.26. These expressions provide exact moments of the expansions, which converge under refinement to moments of the true response functions.

Similar expressions can be derived for stochastic collocation:

$$\mu_i = \langle R_i \rangle \cong \sum_{k=1}^{N_p} r_{ik} \langle \boldsymbol{L}_k(\boldsymbol{\xi}) \rangle = \sum_{k=1}^{N_p} r_{ik} w_k \tag{2.39}$$

$$\Sigma_{ij} = \langle R_i R_j \rangle - \mu_i \mu_j \cong \sum_{k=1}^{N_p} \sum_{l=1}^{N_p} r_{ik} r_{jl} \langle \boldsymbol{L}_k(\boldsymbol{\xi}) \boldsymbol{L}_l(\boldsymbol{\xi}) \rangle - \mu_i \mu_j = \sum_{k=1}^{N_p} r_{ik} r_{jk} w_k - \mu_i \mu_j \tag{2.40}$$

where we have simplified the expectation of Lagrange polynomials constructed at Gauss points and then integrated at these same Gauss points. For tensor grids and sparse grids with fully nested rules, these expectations leave only the weight corresponding to the point for which the interpolation value is one, such that the final equalities in Eqs. 2.39–2.40 hold precisely. For sparse grids with non-nested rules, however, interpolation error exists at the collocation points, such that these final equalities hold only approximately. In this case, we have the choice of computing the moments based on sparse numerical integration or based on the moments of the (imperfect) sparse interpolant, where small differences may exist prior to numerical convergence. In DAKOTA, we employ the former approach; i.e., the right-most expressions in Eqs. 2.39–2.40 are employed for all tensor and sparse cases irregardless of nesting. Skewness and kurtosis calculations as well as sensitivity derivations in the following sections are also based on this choice. The expressions for skewness and (excess) kurtosis from direct numerical integration of the response function are as follows:

$$\gamma_{1_i} = \left\langle \left(\frac{R_i - \mu_i}{\sigma_i} \right)^3 \right\rangle \cong \frac{1}{\sigma_i^3} \left[\sum_{k=1}^{N_p} (r_{ik} - \mu_i)^3 w_k \right] \tag{2.41}$$

$$\gamma_{2_i} = \left\langle \left(\frac{R_i - \mu_i}{\sigma_i} \right)^4 \right\rangle - 3 \cong \frac{1}{\sigma_i^4} \left[\sum_{k=1}^{N_p} (r_{ik} - \mu_i)^4 w_k \right] - 3 \tag{2.42}$$

2.9 Local sensitivity analysis: derivatives with respect to expansion variables

Polynomial chaos expansions are easily differentiated with respect to the random variables [68]. First, using Eq. 2.12,

$$\frac{dR}{d\xi_i} = \sum_{j=0}^{P} \alpha_j \frac{d\Psi_j}{d\xi_i}(\boldsymbol{\xi}) \tag{2.43}$$

and then using Eq. 2.11,

$$\frac{d\Psi_j}{d\xi_i}(\boldsymbol{\xi}) = \frac{d\psi_{t_i^j}}{d\xi_i}(\xi_i) \prod_{\substack{k=1 \\ k\neq i}}^{n} \psi_{t_k^j}(\xi_k) \tag{2.44}$$

where the univariate polynomial derivatives $\frac{d\psi}{d\xi}$ have simple closed form expressions for each polynomial in the Askey scheme [1]. Finally, using the Jacobian of the (extended) Nataf variable transformation,

$$\frac{dR}{dx_i} = \frac{dR}{d\boldsymbol{\xi}} \frac{d\boldsymbol{\xi}}{dx_i} \tag{2.45}$$

which simplifies to $\frac{dR}{d\xi_i}\frac{d\xi_i}{dx_i}$ in the case of uncorrelated x_i.

Similar expressions may be derived for stochastic collocation, starting from Eq. 2.18:

$$\frac{dR}{d\xi_i} = \sum_{j=1}^{N_p} r_j \frac{d\boldsymbol{L}_j}{d\xi_i}(\boldsymbol{\xi}) \tag{2.46}$$

where the multidimensional interpolant $\boldsymbol{L}_j$ is formed over either tensor-product quadrature points or a Smolyak sparse grid. For the former case, the derivative of the multidimensional interpolant $\boldsymbol{L}_j$ involves differentiation of Eq. 2.19:

$$\frac{d\boldsymbol{L}_j}{d\xi_i}(\boldsymbol{\xi}) = \frac{dL_{c_i^j}}{d\xi_i}(\xi_i) \prod_{\substack{k=1 \\ k\neq i}}^{n} L_{c_k^j}(\xi_k) \tag{2.47}$$

and for the latter case, the derivative involves a linear combination of these product rules, as dictated by the Smolyak recursion shown in Eq. 2.31. Finally, calculation of $\frac{dR}{dx_i}$ involves the same Jacobian application shown in Eq. 2.45.

2.10 Global sensitivity analysis: variance-based decomposition

In addition to obtaining derivatives of stochastic expansions with respect to the random variables, it is possible to obtain variance-based sensitivity indices from the stochastic expansions. Variance-based sensitivity indices are explained in the Design of Experiments Chapter of the User's Manual [2]. The concepts are summarized here as well. Variance-based decomposition is a global sensitivity method that summarizes how the uncertainty in model output can be apportioned to uncertainty in individual input variables. VBD uses two primary measures, the main effect sensitivity index S_i and the total effect index T_i. These indices are also called the Sobol' indices. The main effect sensitivity index corresponds to the fraction of the uncertainty in the output, Y, that can be attributed to input x_i alone. The total effects index corresponds to the fraction of the uncertainty in the output, Y, that can be attributed to input x_i and its interactions with other variables. The main effect sensitivity index compares the variance of the conditional expectation $Var_{x_i}[E(Y|x_i)]$ against the total variance $Var(Y)$. Formulas for the indices are:

$$S_i = \frac{Var_{x_i}[E(Y|x_i)]}{Var(Y)} \tag{2.48}$$

and

$$T_i = \frac{E(Var(Y|x_{-i}))}{Var(Y)} = \frac{Var(Y) - Var(E[Y|x_{-i}])}{Var(Y)} \tag{2.49}$$

where $Y = f(\mathbf{x})$ and $x_{-i} = (x_1, ..., x_{i-1}, x_{i+1}, ..., x_m)$.

The calculation of S_i and T_i requires the evaluation of m-dimensional integrals which are typically approximated by Monte-Carlo sampling. However, in stochastic expansion methods, it is possible to obtain the sensitivity indices as analytic functions of the coefficients in the stochastic expansion. The derivation of these results is presented in [76]. The sensitivity indices are printed as a default when running either polynomial chaos or stochastic collocation in DAKOTA. Note that in addition to the first-order main effects, S_i, we are able to calculate the sensitivity indices for higher order interactions such as the two-way interaction $S_{i,j}$.

2.11 Automated Refinement

Several approaches for refinement of stochastic expansions are currently supported, involving uniform or dimension-adaptive approaches to p- or h-refinement using structured (isotropic, anisotropic, or generalized) or unstructured grids. Specific combinations include:

- uniform refinement with unbiased grids
  - p-refinement: isotropic global tensor/sparse grids (PCE, SC) and regression (PCE only) with global basis polynomials
  - h-refinement: isotropic global tensor/sparse grids with local basis polynomials (SC only)
- dimension-adaptive refinement with biased grids
  - p-refinement: anisotropic global tensor/sparse grids with global basis polynomials using global sensitivity analysis (PCE, SC) or spectral decay rate estimation (PCE only)
  - h-refinement: anisotropic global tensor/sparse grids with local basis polynomials (SC only) using global sensitivity analysis
- goal-oriented dimension-adaptive refinement with greedy adaptation
  - p-refinement: generalized sparse grids with global basis polynomials (PCE, SC)
  - h-refinement: generalized sparse grids with local basis polynomials (SC only)

Each involves incrementing the global grid using differing grid refinement criteria, synchronizing the stochastic expansion specifics for the updated grid, and then updating the statistics and computing convergence criteria. Future releases will support local h-refinement approaches that can replace or augment the global grids currently supported. The sub-sections that follow enumerate each of the first level bullets above.

2.11.1 Uniform refinement with unbiased grids

Uniform refinement involves ramping the resolution of a global structured or unstructured grid in an unbiased manner and then refining an expansion to synchronize with this increased grid resolution. In the case of increasing the order of an isotropic tensor-product quadrature grid or the level of an isotropic Smolyak sparse grid, a p-refinement approach increases the order of the global basis polynomials (Sections 2.1, 2.2.1, and 2.2.2) in a synchronized manner and an h-refinement approach reduces the approximation range of fixed order local basis polynomials (Sections 2.2.3 and 2.2.4). And in the case of uniform p-refinement with PCE regression, the collocation oversampling ratio (refer to Methods specification within DAKOTA Reference Manual [3]) is held fixed, such that an increment in isotropic expansion order is matched with a corresponding increment in the number

of structured (probabilistic collocation) or unstructured samples (point collocation) employed in the linear least squares solve.

For uniform refinement, anisotropic dimension preferences are not computed, and the only algorithmic requirements are:

- With the usage of nested integration rules with restricted exponential growth, DAKOTA must ensure that a change in level results in a sufficient change in the grid; otherwise premature convergence could occur within the refinement process. If no change is initially detected, DAKOTA continues incrementing the order/level (without grid evaluation) until the number of grid points increases.
- A convergence criterion is required. For uniform refinement, DAKOTA employs the L^2 norm of the change in the response covariance matrix as a general-purpose convergence metric.

2.11.2 Dimension-adaptive refinement with biased grids

Dimension-adaptive refinement involves ramping the order of a tensor-product quadrature grid or the level of a Smolyak sparse grid anisotropically, that is, using a defined dimension preference. This dimension preference may be computed from local sensitivity analysis, global sensitivity analysis, a posteriori error estimation, or decay rate estimation. In the current release, we focus on global sensitivity analysis and decay rate estimation. In the former case, dimension preference is defined from total Sobol' indices (Eq. 2.49) and is updated on every iteration of the adaptive refinement procedure, where higher variance attribution for a dimension indicates higher preference for that dimension. In the latter case, the spectral decay rates for the polynomial chaos coefficients are estimated using the available sets of univariate expansion terms (interaction terms are ignored). Given a set of scaled univariate coefficients (scaled to correspond to a normalized basis), the decay rate can be inferred using a technique analogous to Richardson extrapolation. The dimension preference is then defined from the inverse of the rate: slowly converging dimensions need greater refinement pressure. For both of these cases, the dimension preference vector supports anisotropic sparse grids based on a linear index-set constraint (Eq. 2.35) or anisotropic tensor grids (Eq. 2.28) with dimension order scaled proportionately to preference; for both grids, dimension refinement lower bound constraints are enforced to ensure that all previously evaluated points remain in new refined grids.

Given an anisotropic global grid, the expansion refinement proceeds as for the uniform case, in that the p-refinement approach increases the order of the global basis polynomials (Sections 2.1, 2.2.1, and 2.2.2) in a synchronized manner and an h-refinement approach reduces the approximation range of fixed order local basis polynomials (Sections 2.2.3 and 2.2.4). Also, the same grid change requirements and convergence criteria described for uniform refinement (Section 2.11.1) are applied in this case.

2.11.3 Goal-oriented dimension-adaptive refinement with greedy adaptation

Relative to the uniform and dimension-adaptive refinement capabilities described previously, the generalized sparse grid algorithm [39] supports greater flexibility in the definition of sparse grid index sets and supports refinement controls based on general statistical quantities of interest (QOI). This algorithm was originally intended for adaptive numerical integration on a hypercube, but can be readily extended to the adaptive refinement of stochastic expansions using the following customizations:

- In addition to hierarchical interpolants in SC, we employ independent polynomial chaos expansions for each active and accepted index set. Pushing and popping index sets then involves increments of tensor chaos expansions (as described in Section 2.6.3) along with corresponding increments to the Smolyak combinatorial coefficients.

- Since we support bases for more than uniform distributions on a hypercube, we exploit rule nesting when possible (i.e., Gauss-Patterson for uniform or transformed uniform variables, and Genz-Keister for normal or transformed normal variables), but we do not require it. This implies a loss of some algorithmic simplifications in [39] that occur when grids are strictly hierarchical.

- In the evaluation of the effect of a trial index set, the goal in [39] is numerical integration and the metric is the size of the increment induced by the trial set on the expectation of the function of interest. It is straightforward to instead measure the effect of a trial index set on response covariance, numerical probability, or other statistical QOI computed by post-processing the resulting PCE or SC expansion. By tying the refinement process more closely to the statistical QOI, the refinement process can become more efficient in achieving the desired analysis objectives.

Given these customizations, the algorithmic steps can be summarized as:

1. *Initialization:* Starting from an initial isotropic or anisotropic reference grid (often the $w = 0$ grid corresponding to a single collocation point), accept the reference index sets as the old set and define active index sets using the admissible forward neighbors of all old index sets.

2. *Trial set evaluation:* Evaluate the tensor grid corresponding to each trial active index set, form the tensor polynomial chaos expansion or tensor interpolant corresponding to it, update the Smolyak combinatorial coefficients, and combine the trial expansion with the reference expansion. Perform necessary bookkeeping to allow efficient restoration of previously evaluated tensor expansions.

3. *Trial set selection:* Select the trial index set that induces the largest change in the statistical QOI, normalized by the cost of evaluating the trial index set (as indicated by the number of new collocation points in the trial grid). In our implementation, the statistical QOI is defined using an L^2 norm of change in CDF/CCDF probability/reliability/response level mappings, when level mappings are present, or L^2 norm of change in response covariance, when level mappings are not present.

4. *Update sets:* If the largest change induced by the trial sets exceeds a specified convergence tolerance, then promote the selected trial set from the active set to the old set and update the active sets with new admissible forward neighbors; return to step 2 and evaluate all trial sets with respect to the new reference point. If the convergence tolerance is satisfied, advance to step 5.

5. *Finalization:* Promote all remaining active sets to the old set, update the Smolyak combinatorial coefficients, and perform a final combination of tensor expansions to arrive at the final result for the statistical QOI.

2.12 Multifidelity methods

In a multifidelity uncertainty quantification approach employing stochastic expansions, we seek to utilize a predictive low-fidelity model to our advantage in reducing the number of high-fidelity model evaluations required to compute high-fidelity statistics to a particular precision. When a low-fidelity model captures useful trends of the high-fidelity model, then the model discrepancy may have either lower complexity, lower variance, or both, requiring less computational effort to resolve its functional form than that required for the original high-fidelity model [58].

To accomplish this goal, an expansion will first be formed for the model discrepancy (the difference between response results if additive correction or the ratio of results if multiplicative correction). These discrepancy

functions are the same functions approximated in surrogate-based minimization (see Surrogate Models section within the Models chapter of the User's Manual [2]). The exact discrepancy functions are

$$A(\boldsymbol{\xi}) = R_{hi}(\boldsymbol{\xi}) - R_{lo}(\boldsymbol{\xi}) \tag{2.50}$$

$$B(\boldsymbol{\xi}) = \frac{R_{hi}(\boldsymbol{\xi})}{R_{lo}(\boldsymbol{\xi})} \tag{2.51}$$

Approximating the high-fidelity response functions using approximations of these discrepancy functions then involves

$$\hat{R}_{hi_A}(\boldsymbol{\xi}) = R_{lo}(\boldsymbol{\xi}) + \hat{A}(\boldsymbol{\xi}) \tag{2.52}$$

$$\hat{R}_{hi_B}(\boldsymbol{\xi}) = R_{lo}(\boldsymbol{\xi})\hat{B}(\boldsymbol{\xi}) \tag{2.53}$$

where $\hat{A}(\boldsymbol{\xi})$ and $\hat{B}(\boldsymbol{\xi})$ are stochastic expansion approximations to the exact correction functions:

$$\hat{A}(\boldsymbol{\xi}) = \sum_{j=0}^{P_{hi}} \alpha_j \Psi_j(\boldsymbol{\xi}) \quad \text{or} \quad \sum_{j=1}^{N_{hi}} a_j \boldsymbol{L}_j(\boldsymbol{\xi}) \tag{2.54}$$

$$\hat{B}(\boldsymbol{\xi}) = \sum_{j=0}^{P_{hi}} \beta_j \Psi_j(\boldsymbol{\xi}) \quad \text{or} \quad \sum_{j=1}^{N_{hi}} b_j \boldsymbol{L}_j(\boldsymbol{\xi}) \tag{2.55}$$

where α_j and β_j are the spectral coefficients for a polynomial chaos expansion (evaluated via Eq. 2.25, for example) and a_j and b_j are the interpolation coefficients for stochastic collocation (values of the exact discrepancy evaluated at the collocation points).

Second, an expansion will be formed for the low fidelity surrogate model, where the intent is for the level of resolution to be higher than that required to resolve the discrepancy ($P_{lo} \gg P_{hi}$ or $N_{lo} \gg N_{hi}$; either enforced statically through order/level selections or automatically through adaptive refinement):

$$R_{lo}(\boldsymbol{\xi}) \cong \sum_{j=0}^{P_{lo}} \gamma_j \Psi_j(\boldsymbol{\xi}) \quad \text{or} \quad \sum_{j=1}^{N_{lo}} r_{lo_j} \boldsymbol{L}_j(\boldsymbol{\xi}) \tag{2.56}$$

Then the two expansions are combined (added or multiplied) into a new expansion that approximates the high fidelity model, from which the final set of statistics are generated. For polynomial chaos expansions, this combination entails:

- in the additive case, the high-fidelity expansion is formed by simply overlaying the expansion forms and adding the spectral coefficients that correspond to the same basis polynomials.
- in the multiplicative case, the form of the high-fidelity expansion must first be defined to include all polynomial orders indicated by the products of each of the basis polynomials in the low fidelity and discrepancy expansions (most easily estimated from total-order, tensor, or sum of tensor expansions which involve simple order additions). Then the coefficients of this product expansion are computed as follows (shown generically for $z = xy$ where x, y, and z are each expansions of arbitrary form):

$$\sum_{k=0}^{P_z} z_k \Psi_k(\boldsymbol{\xi}) = \sum_{i=0}^{P_x} \sum_{j=0}^{P_y} x_i y_j \Psi_i(\boldsymbol{\xi}) \Psi_j(\boldsymbol{\xi}) \tag{2.57}$$

$$z_k = \frac{\sum_{i=0}^{P_x} \sum_{j=0}^{P_y} x_i y_j \langle \Psi_i \Psi_j \Psi_k \rangle}{\langle \Psi_k^2 \rangle} \tag{2.58}$$

where tensors of one-dimensional basis triple products $\langle \psi_i \psi_j \psi_k \rangle$ are typically sparse and can be efficiently precomputed using one dimensional quadrature for fast lookup within the multidimensional triple products.

For stochastic collocation, the high-fidelity expansion generated from combining the low fidelity and discrepancy expansions retains the polynomial form of the low fidelity expansion, for which only the coefficients are updated in order to interpolate the sum or product values (and potentially their derivatives). Since we will typically need to produce values of a less resolved discrepancy expansion on a more resolved low fidelity grid to perform this combination, we utilize the discrepancy expansion rather than the original discrepancy function values for both interpolated and non-interpolated point values (and derivatives), in order to ensure consistency.

Chapter 3

Epistemic Methods

This chapter covers theoretical aspects of methods for propagating epistemic uncertainty.

3.1 Dempster-Shafer theory of evidence (DSTE)

In Dempster-Shafer theory, the event space is defined by a triple $(\mathcal{S}, \mathbb{S}, m)$ which defines $\mathcal{S}$ the universal set, $\mathbb{S}$ a countable collection of subsets of $\mathcal{S}$, and a notional measure m. $\mathcal{S}$ and $\mathbb{S}$ have a similar meaning to that in classical probability theory; the main difference is that $\mathbb{S}$, also known as the focal elements, does not have to be a σ-algebra over $\mathcal{S}$. The operator m is defined to be

$$m(\mathcal{U}) = \begin{cases} > 0 & \text{if } \mathcal{U} \in \mathbb{S} \\ 0 & \text{if } \mathcal{U} \subset \mathcal{S} \text{ and } \mathcal{U} \notin \mathbb{S} \end{cases} \tag{3.1}$$

$$\sum_{\mathcal{U} \in \mathbb{S}} m(\mathcal{U}) = 1 \tag{3.2}$$

where $m(\mathcal{U})$ is known as the basic probability assignment (BPA) of the set $\mathcal{U}$. In the DSTE framework, belief and plausibility are defined as:

$$\mathrm{Bel}(\mathcal{E}) = \sum_{\{\mathcal{U} \mid \mathcal{U} \subset \mathcal{E},\, \mathcal{U} \in \mathbb{S}\}} m(\mathcal{U}) \tag{3.3}$$

$$\mathrm{Pl}(\mathcal{E}) = \sum_{\{\mathcal{U} \mid \mathcal{U} \cap \mathcal{E} \neq \emptyset,\, \mathcal{U} \in \mathbb{S}\}} m(\mathcal{U}) \tag{3.4}$$

The belief Bel($\mathcal{E}$) is interpreted to be the minimum likelihood that is associated with the event $\mathcal{E}$. Similarly, the plausibility Pl($\mathcal{E}$) is the maximum amount of likelihood that could be associated with $\mathcal{E}$. This particular structure allows us to handle unconventional inputs, such as conflicting pieces of evidence (e.g. dissenting expert opinions), that would be otherwise discarded in an interval analysis or probabilistic framework. The ability to make use of this information results in a commensurately more informed output.

The procedure to compute belief structures involves four major steps:

1. Determine the set of d-dimensional hypercubes that have a nonzero evidential measure
2. Compute the composite evidential measure (BPA) of each hypercube
3. Propagate each hypercube through the model and obtain the response bounds within each hypercube
4. Aggregate the minimum and maximum values of the response per hypercube with the BPAs to obtain cumulative belief and plausibility functions on the response (e.g. calculate a belief structure on the response).

The first step involves identifying combinations of focal elements defined on the inputs that define a hypercube. The second step involves defining an aggregate BPA for that hypercube, which is the product of the BPAs of the individual focal elements defining the hypercube. The third step involves finding the maximum and minimum values of the response value in each hypercube, and this part can be very computationally expensive. Finally, the results over all hypercubes are aggregated to form belief structures on the response.

Chapter 4

Surrogate Models

This chapter deals with the theory behind DAKOTA's surrogate models, which are also known as response surfaces and meta-models.

4.1 Kriging and Gaussian Process Models

In this discussion of Kriging and Gaussian Process (GP) models, vectors are indicated by a single underline and matrices are indicated by a double underline. Capital letters indicate data, or functions of data, that is used to construct an emulator. Lower case letters indicate arbitrary points, i.e. points where the simulator may or may not have been evaluated, and functions of arbitrary points. Estimates, approximations, and models are indicated by hats. For instance, $\hat{f}(\underline{x})$ is a model/emulator of the function $f(\underline{x})$ and $\hat{y}$ is the emulator's prediction or estimate of the true response $y = f(\underline{x})$ evaluated at the point $\underline{x}$. A tilde indicates a reordering of points/equations, with the possible omission of some but not all points. N is the number of points in the sample design and M is the number of input dimensions.

4.1.1 Kriging & Gaussian Processes: Function Values Only

The set of interpolation techniques known as Kriging, also referred to as Gaussian Processes, were originally developed in the geostatistics and spatial statistics communities to produce maps of underground geologic deposits based on a set of widely and irregularly spaced borehole sites[19]. Building a Kriging model typically involves the

1. Choice of a trend function,
2. Choice of a correlation function, and
3. Estimation of correlation parameters.

A Kriging emulator, $\hat{f}(\underline{x})$, consists of a trend function (frequently a least squares fit to the data, $\underline{g}(\underline{x})^T \underline{\beta}$) plus a Gaussian process error model, $\epsilon(\underline{x})$, that is used to correct the trend function.

$$\hat{f}(\underline{x}) = \underline{g}(\underline{x})^T \underline{\beta} + \epsilon(\underline{x})$$

This represents an estimated distribution for the unknown true surface, $f(\underline{x})$. The error model, $\epsilon(\underline{x})$, makes an adjustment to the trend function so that the emulator will interpolate, and have zero uncertainty at, the data points it was built from. The covariance between the error at two arbitrary points, $\underline{x}$ and $\underline{x}'$, is modeled as

$$\mathrm{Cov}\left(y\left(\underline{x}\right), y\left(\underline{x}'\right)\right) = \mathrm{Cov}\left(\epsilon\left(\underline{x}\right), \epsilon\left(\underline{x}'\right)\right) = \sigma^2\, r\left(\underline{x}, \underline{x}'\right).$$

Here σ^2 is known as the unadjusted variance and $r\left(\underline{x}, \underline{x}'\right)$ is a correlation function. Measurement error can be modeled explicitly by modifying this to

$$\mathrm{Cov}\left(\epsilon\left(\underline{x}\right), \epsilon\left(\underline{x}'\right)\right) = \sigma^2\, r\left(\underline{x}, \underline{x}'\right) + \Delta^2 \delta\left(\underline{x} - \underline{x}'\right)$$

where

$$\delta\left(\underline{x} - \underline{x}'\right) = \begin{cases} 1 & \text{if } \underline{x} - \underline{x}' = \underline{0} \\ 0 & \text{otherwise} \end{cases}$$

and Δ^2 is the variance of the measurement error. In this work, the term "nugget" refers to the ratio $\eta = \frac{\Delta^2}{\sigma^2}$.

By convention, the terms simple Kriging, ordinary Kriging, and universal Kriging are used to indicate the three most common choices for the trend function. In simple Kriging, the trend is treated as a known constant, usually zero, $g\left(\underline{x}\right)\beta \equiv 0$. Universal Kriging [56] uses a general polynomial trend model $\underline{g}\left(\underline{x}\right)^T \underline{\beta}$ whose coefficients are determined by least squares regression. Ordinary Kriging is essentially universal Kriging with a trend order of zero, i.e. the trend function is treated as an unknown constant and $g\left(\underline{x}\right) = 1$. N_β is the number of basis functions in $\underline{g}\left(\underline{x}\right)$ and therefore number of elements in the vector $\underline{\beta}$.

For a finite number of sample points, N, there will be uncertainty about the most appropriate value of the vector, $\underline{\beta}$, which can therefore be described as having a distribution of possible values. If one assumes zero prior knowledge about this distribution, which is referred to as the "vague prior" assumption, then the maximum likelihood value of $\underline{\beta}$ can be computed via least squares generalized by the inverse of the error model's correlation matrix, $\underline{\underline{R}}$

$$\underline{\hat{\beta}} = \left(\underline{\underline{G}}^T \underline{\underline{R}}^{-1} \underline{\underline{G}}\right)^{-1} \left(\underline{\underline{G}}^T \underline{\underline{R}}^{-1} \underline{Y}\right).$$

Here $\underline{\underline{G}}$ is a N by N_β matrix that contains the evaluation of the least squares basis functions at all points in $\underline{\underline{X}}$, such that $G_{i,l} = g_l\left(\underline{X_i}\right)$. The real, symmetric, positive-definite correlation matrix, $\underline{\underline{R}}$, of the error model contains evaluations of the correlation function, r, at all pairwise combination of points (rows) in the sample design, $\underline{\underline{X}}$.

$$R_{i,j} = R_{j,i} = r\left(\underline{X_i}, \underline{X_j}\right) = r\left(\underline{X_j}, \underline{X_i}\right)$$

There are many options for r, among them are the following families of correlation functions:

- **Powered-Exponential**

$$r\left(\underline{X_i}, \underline{X_j}\right) = \exp\left(-\sum_{k=1}^{M} \theta_k \left|X_{i,k} - X_{j,k}\right|^\gamma\right) \tag{4.1}$$

  where $0 < \gamma \leq 2$ and $0 < \theta_k$.

- **Matern**

$$r\left(\underline{X_i}, \underline{X_j}\right) = \prod_{k=1}^{M} \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\theta_k \left|X_{i,k} - X_{j,k}\right|\right)^\nu \mathcal{K}_\nu\left(\theta_k \left|X_{i,k} - X_{j,k}\right|\right)$$

  where $0 < \nu$, $0 < \theta_k$, and $\mathcal{K}_\nu(\cdot)$ is the modified Bessel function of order ν; $\nu = s + \frac{1}{2}$ is the smallest value of ν which results in a Kriging model that is s times differentiable.

- **Cauchy**

$$r\left(\underline{X_i}, \underline{X_j}\right) = \prod_{k=1}^{M} \left(1 + \theta_k \left|X_{i,k} - X_{j,k}\right|^{\gamma}\right)^{-\nu}$$

where $0 < \gamma \leq 2$, $0 < \nu$, and $0 < \theta_k$.

Gneiting et al. [43] provide a more thorough discussion of the properties of and relationships between these three families. Some additional correlation functions include the Dagum family [8] and cubic splines.

The squared exponential or Gaussian correlation function (Equation 4.1 with $\gamma = 2$) was selected to be the first correlation function implemented in DAKOTA on the basis that its infinite smoothness or differentiability should aid in leveraging the anticipated and hopefully sparse data. For the Gaussian correlation function, the correlation parameters, $\underline{\theta}$, are related to the correlation lengths, $\underline{L}$, by

$$\theta_k = \frac{1}{2\, L_k^2}. \tag{4.2}$$

Here, the correlation lengths, $\underline{L}$, are analogous to standard deviations in the Gaussian or normal distribution and often have physical meaning. The adjusted (by data) mean of the emulator is a best linear unbiased estimator of the unknown true function,

$$\hat{y} = \mathrm{E}\left(\hat{f}\left(\underline{x}\right) | \underline{f}\left(\underline{\underline{X}}\right)\right) = \underline{g}\left(\underline{x}\right)^T \underline{\hat{\beta}} + \underline{r}\left(\underline{x}\right)^T \underline{\underline{R}}^{-1} \underline{\epsilon}. \tag{4.3}$$

Here, $\underline{\epsilon} = \left(\underline{Y} - \underline{\underline{G}}\, \underline{\hat{\beta}}\right)$ is the known vector of differences between the true outputs and trend function at all points in $\underline{\underline{X}}$ and the vector $\underline{r}\left(\underline{x}\right)$ is defined such that $r_i\left(\underline{x}\right) = r\left(\underline{x}, \underline{X_i}\right)$. This correction can be interpreted as the projection of prior belief (the least squares fit) into the span of the data. The adjusted mean of the emulator will interpolate the data that the Kriging model was built from as long as its correlation matrix, $\underline{\underline{R}}$, is numerically non-singular.

Ill-conditioning of $\underline{\underline{R}}$ and other matrices is a recognized as a significant challenge for Kriging. Davis and Morris [20] gave a thorough review of six factors affecting the condition number of matrices associated with Kriging (from the perspective of semivariograms rather than correlation functions). They concluded that "Perhaps the best advice we can give is to be mindful of the condition number when building and solving kriging systems."

In the context of estimating the optimal $\underline{\theta}$, Martin [55] stated that Kriging's "three most prevalent issues are (1) ill-conditioned correlation matrices,(2) multiple local optimum, and (3) long ridges of near optimal values." Because of the second issue, global optimization methods are more robust than local methods. Martin used constrained optimization to address ill-conditioning of $\underline{\underline{R}}$.

Rennen [69] advocated that ill-conditioning be handled by building Kriging models from a uniform subset of available sample points. That option has been available in DAKOTA's "Gaussian process" model (a separate implementation from DAKOTA's "Kriging" model) since version 4.1 [32]. Note that Kriging/Gaussian-Process models will not exactly interpolate the discarded points. The implicit justification for this type of approach is that the row or columns of an ill-conditioned matrix contain a significant amount of duplicate information, and that when discarded, duplicate information should be easy to predict.

As of version 5.2, DAKOTA's `kriging` model has a similar "discard near duplicate points" capability. However, it explicitly addresses the issue of unique information content. Points are **not** discarded prior to the construction

of the Kriging model. Instead, for each vector $\underline{\theta}$ examined that results in an ill-conditioned correlation matrix, $\underline{\underline{R}}$, a pivoted Cholesky factorization of $\underline{\underline{R}}$ is performed. This ranks the points according to how much unique information they contain. Note that the definition of information content depends on $\underline{\theta}$. Low information points are then discarded until $\underline{\underline{R}}$ is no longer ill-conditioned, i.e. until it tightly meets a constraint on condition number. This can be done efficiently using a bisection search that calls LAPACK's fast estimate of the (reciprocal of the) condition number. The possibly, and often, improper subset of points used to construct the Kriging model is the one associated with the chosen $\underline{\theta}$. Methods for selecting $\underline{\theta}$ are discussed below. Since the points that are discarded are the ones that contain the least unique information, they are the ones that are easiest to predict and provide maximum improvement to the condition number.

Adding a nugget, η, to the diagonal entries of $\underline{\underline{R}}$ is a popular approach for both accounting for measurement error in the data and alleviating ill-conditioning. However, doing so will cause the Kriging model to smooth or approximate rather than interpolate the data. Methods for choosing a nugget include:

- Choosing a nugget based on the variance of measurement error (if any); this will be an iterative process if σ^2 is not known in advance.
- Iteratively adding a successively larger nugget until $\underline{\underline{R}} + \eta \underline{\underline{I}}$ is no longer ill-conditioned.
- Exactly calculating the minimum nugget needed for a target condition number from $\underline{\underline{R}}$'s maximum λ_{max} and minimum λ_{min} eigenvalues. The condition number of $\underline{\underline{R}} + \eta \underline{\underline{I}}$ is $\frac{\lambda_{max}+\eta}{\lambda_{min}+\eta}$. However, calculating eigenvalues is computationally expensive. Since Kriging's $\underline{\underline{R}}$ matrix has all ones on the diagonal, its trace and therefore sum of eigenvalues is N. Consequently, a nugget value of $\eta = \frac{N}{\text{target condition number}-1}$ will always alleviate ill-conditioning. A smaller nugget that is also guaranteed to alleviate ill-conditioning can be calculated from LAPACK's fast estimate of the reciprocal of $\underline{\underline{R}}$'s condition number, $\mathrm{rcond}\left(\underline{\underline{R}}\right)$.
- Treating η as another parameter to be selected by the same process used to choose $\underline{\theta}$. Two such approaches are discussed below.

The Kriging model's adjusted variance is commonly used as a spatially varying measure of uncertainty. Knowing where, and by how much, the model "doubts" its own predictions helps build user confidence in the predictions and can be utilized to guide the selection of new sample points during optimization or to otherwise improve the surrogate. The adjusted variance is

$$\begin{aligned}
\mathrm{Var}\left(\hat{y}\right) &= \mathrm{Var}\left(\hat{f}\left(\underline{x}\right) | \underline{f}\left(\underline{\underline{X}}\right)\right) \\
&= \hat{\sigma}^2 \left(1 - \underline{r}\left(\underline{x}\right)^T \underline{\underline{R}}^{-1} \underline{r}\left(\underline{x}\right) + \ldots \right. \\
&\quad \left. \left(\underline{g}\left(\underline{x}\right)^T - \underline{r}\left(\underline{x}\right)^T \underline{\underline{R}}^{-1} \underline{\underline{G}}\right) \left(\underline{\underline{G}}^T \underline{\underline{R}}^{-1} \underline{\underline{G}}\right)^{-1} \left(\underline{g}\left(\underline{x}\right)^T - \underline{r}\left(\underline{x}\right)^T \underline{\underline{R}}^{-1} \underline{\underline{G}}\right)^T\right)
\end{aligned}$$

where the maximum likelihood estimate of the unadjusted variance is

$$\hat{\sigma}^2 = \frac{\underline{\epsilon}^T \underline{\underline{R}}^{-1} \underline{\epsilon}}{N - N_\beta}.$$

There are two types of numerical approaches to choosing $\underline{\theta}$. One of these is to use Bayesian techniques such as Markov Chain Monte Carlo to obtain a distribution represented by an ensemble of vectors $\underline{\theta}$. In this case, evaluating the emulator's mean involves taking a weighted average of Equation 4.3 over the ensemble of $\underline{\theta}$ vectors.

The other, more common, approach to constructing a Kriging model involves using optimization to find the set of correlation parameters $\underline{\theta}$ that maximizes the likelihood of the model given the data. DAKOTA's `gaussian_process` and `kriging` models use the maximum likelihood approach. It is equivalent, and more convenient to maximize the natural logarithm of the likelihood, which assuming a vague prior is,

$$\begin{aligned}\log\left(\mathrm{lik}\left(\underline{\theta}\right)\right) &= -\frac{1}{2}\Bigg(\left(N - N_\beta\right)\left(\frac{\hat{\sigma}^2}{\sigma^2} + \log\left(\sigma^2\right) + \log(2\pi)\right) + ... \\ &\qquad \log\left(\det\left(\underline{\underline{R}}\right)\right) + \log\left(\det\left(\underline{\underline{G}}^T\underline{\underline{R}}^{-1}\underline{\underline{G}}\right)\right)\Bigg).\end{aligned}$$

And, if one substitutes the maximum likelihood estimate $\hat{\sigma}^2$ in for σ^2, then it is equivalent to minimize the following objective function

$$\mathrm{obj}\left(\underline{\theta}\right) = \log\left(\hat{\sigma}^2\right) + \frac{\log\left(\det\left(\underline{\underline{R}}\right)\right) + \log\left(\det\left(\underline{\underline{G}}^T\underline{\underline{R}}^{-1}\underline{\underline{G}}\right)\right)}{N - N_\beta}.$$

Because of the division by $N - N_\beta$, this "per-equation" objective function is mostly independent of the number of sample points, N. It is therefore useful for comparing the (estimated) "goodness" of Kriging models that have different numbers of sample points, e.g. when an arbitrary number of points can be discarded by the pivoted Cholesky approach described above.

Note that the determinant of $\underline{\underline{R}}$ (and $\left(\underline{\underline{G}}^T\underline{\underline{R}}^{-1}\underline{\underline{G}}\right)$) can be efficiently calculated as the square of the product of the diagonals of its Cholesky factorization. However, this will often underflow, i.e. go to zero, making its log incorrectly go to $-\infty$. A more accurate and robust calculation of $\log\left(\det\left(\underline{\underline{R}}\right)\right)$ can be achieved by taking twice the sum of the log of the diagonals of $\underline{\underline{R}}$'s Cholesky factorization.

Also note, that in the absence of constraints, maximizing the likelihood would result in singular $\underline{\underline{R}}$ which makes the emulator incapable of reproducing the data from which it was built. This is because a singular $\underline{\underline{R}}$ makes $\log\left(\det\left(\underline{\underline{R}}\right)\right) = -\infty$ and the *estimate* of likelihood infinite. Constraints are therefore required. Two types of constraints are used in DAKOTA's `kriging` models.

The first of these is an explicit constraint on LAPACK's fast estimate of the (reciprocal of the) condition number, $2^{-40} < \mathrm{rcond}\left(\underline{\underline{R}}\right)$. The value 2^{-40} was chosen based on the assumption that double precision arithmetic is used. Double precision numbers have 52 bits of precision. Therefore the $2^{-40} < \mathrm{rcond}\left(\det\left(\underline{\underline{R}}\right)\right)$ implies that at least the leading three significant figures should be uncorrupted by round off error. In DAKOTA 5.2, this constraint is used to determine how many points can be retained in the pivoted Cholesky approach to subset selection described above.

The second, is a box constraint defining a small "feasible" region in correlation length space to search during the maximum likelihood optimization. Many global optimizers, such as the DIRECT (DIvision of RECTangles) used by DAKOTA's Gaussian Process (as the only option) and Kriging (as the default option) models, require a box constraint definition for the range of acceptable parameters. By default, DAKOTA's `kriging` model defines the input space to be the smallest hyper-rectangle that contains the sample design. The user has the option to define a larger input space that includes a region where they wish to extrapolate. Note that the emulator can be evaluated at points outside the defined input space, but this definition helps to determine the extent of the "feasible" region of correlation lengths. Let the input space be normalized to the unit hypercube centered at the origin. The average

distance between nearest neighboring points is then

$$d = \left(\frac{1}{N}\right)^{1/M}.$$

DAKOTA's "feasible" range of correlation lengths, $\underline{L}$, for the Gaussian correlation function is

$$\frac{d}{4} \leq L_k \leq 8d.$$

This range was chosen based on correlation lengths being analogous to the standard deviation in the Gaussian or Normal distribution. If the correlation lengths are set to $L_k = d/4$, then nearest neighboring points "should be" roughly four "standard deviations" away making them almost completely uncorrelated with each other. $\underline{\underline{R}}$ would then be a good approximation of the identity matrix and have a condition number close to one. In the absence of a pathological spacing of points, this range of $\underline{L}$ should contain some non-singular $\underline{\underline{R}}$. $L_k = 8d$ implies approximately 32% trust in what points 8 neighbors away have to say and 5% trust in what points 16 neighbors away have to say. It is possible that the optimal correlation lengths are larger than $8d$; but if so, then either almost all of the same information will be contained in more nearby neighbors, or it was not appropriate to use the squared-exponential/Gaussian correlation function. When other correlation functions are added to the DAKOTA Kriging implementation, each will be associated with its own range of appropriate correlation lengths chosen by similar reasoning. A different definition of d could be used for non-hypercube input domains.

4.1.2 Gradient Enhanced Kriging

This section focuses on the incorporation of derivative information into Kriging models and challenges in their implementation. Special attention is paid to conditioning issues.

There are at least three basic approaches for incorporating derivative information into Kriging. These are

1. **Indirect**: The sample design is augmented with fictitious points nearby actual sample points which are predicted from derivative information and then a Kriging model is built from the augmented design.

2. **Co-Kriging**: The derivatives with respect to each input variables are treated as separate but correlated output variables and a Co-Kriging model is built for the set of output variables. This would use $\begin{pmatrix} M+2 \\ 2 \end{pmatrix}$ $\underline{\theta}$ vectors.

3. **Direct**: The relationship between the response value and its derivatives is leveraged to use a single $\underline{\theta}$ by assuming

$$\mathrm{Cov}\left(y\left(\underline{x}^1\right), \frac{\partial y\left(\underline{x}^2\right)}{\partial x_k^2}\right) = \frac{\partial}{\partial x_k^2}\left(\mathrm{Cov}\left(y\left(\underline{x}^1\right), y\left(\underline{x}^2\right)\right)\right). \tag{4.4}$$

DAKOTA 5.2 and later includes an implementation of the direct approach, herein referred to simply as Gradient Enhanced (universal) Kriging (GEK). The equations for GEK can be derived by assuming Equation 4.4 and then taking the same steps used to derive function value only Kriging. The superscript on $\underline{x}$ in Equation 4.4 and below indicates whether it's the 1st or 2nd input to $r\left(\underline{x}^1, \underline{x}^2\right)$. Note that when the first and second arguments are the same, the derivative of $r\left(\ ,\ \right)$ with respect to the first argument is equal in magnitude but opposite in sign compared to the derivative with respect to the second argument. The GEK equations can also be obtained by

starting from a Kriging model and making the following substitutions $\underline{Y} \rightarrow \underline{Y_\nabla}$, $\underline{\underline{G}} \rightarrow \underline{\underline{G_\nabla}}$, $\underline{r} \rightarrow \underline{r_\nabla}$, $\underline{\underline{R}} \rightarrow \underline{\underline{R_\nabla}}$, and $N \rightarrow N_\nabla = N\,(1+M)$, where N_∇ is the number of equations rather than the number of points,

$$\underline{Y_\nabla} = \begin{bmatrix} \underline{Y} \\ \frac{\partial \underline{Y}}{\partial X_{:,1}} \\ \frac{\partial \underline{Y}}{\partial X_{:,2}} \\ \vdots \\ \frac{\partial \underline{Y}}{\partial X_{:,M}} \end{bmatrix}, \qquad \underline{\underline{G_\nabla}} = \begin{bmatrix} \underline{\underline{G}} \\ \frac{\partial \underline{\underline{G}}}{\partial X_{:,1}} \\ \frac{\partial \underline{\underline{G}}}{\partial X_{:,2}} \\ \vdots \\ \frac{\partial \underline{\underline{G}}}{\partial X_{:,M}} \end{bmatrix}, \qquad \underline{r_\nabla} = \begin{bmatrix} \underline{r} \\ \frac{\partial \underline{r}}{\partial X_{:,1}} \\ \frac{\partial \underline{r}}{\partial X_{:,2}} \\ \vdots \\ \frac{\partial \underline{r}}{\partial X_{:,M}} \end{bmatrix}$$

$$\underline{\underline{R_\nabla}} = \begin{bmatrix} \underline{\underline{R}} & \frac{\partial \underline{\underline{R}}}{\partial X^2_{:,1}} & \frac{\partial \underline{\underline{R}}}{\partial X^2_{:,2}} & \cdots & \frac{\partial \underline{\underline{R}}}{\partial X^2_{:,M}} \\ \frac{\partial \underline{\underline{R}}}{\partial X^1_{:,1}} & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,1}\partial X^2_{:,1}} & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,1}\partial X^2_{:,2}} & \cdots & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,1}\partial X^2_{:,M}} \\ \frac{\partial \underline{\underline{R}}}{\partial X^1_{:,2}} & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,2}\partial X^2_{:,1}} & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,2}\partial X^2_{:,2}} & \cdots & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,2}\partial X^2_{:,M}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \underline{\underline{R}}}{\partial X^1_{:,M}} & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,M}\partial X^2_{:,1}} & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,M}\partial X^2_{:,2}} & \cdots & \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,M}\partial X^2_{:,M}} \end{bmatrix}$$

$$\frac{\partial \underline{\underline{R}}}{\partial X^1_{:,I}} = -\left(\frac{\partial \underline{\underline{R}}}{\partial X^1_{:,I}}\right)^T = -\frac{\partial \underline{\underline{R}}}{\partial X^2_{:,I}} = \left(\frac{\partial \underline{\underline{R}}}{\partial X^2_{:,I}}\right)^T$$

$$\frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,I}\partial X^2_{:,J}} = \left(\frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,I}\partial X^2_{:,J}}\right)^T = \frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,J}\partial X^2_{:,I}} = \left(\frac{\partial^2 \underline{\underline{R}}}{\partial X^1_{:,J}\partial X^2_{:,I}}\right)^T$$

Here capital I and J are scalar indices for the input dimension (column) of the sample design, $\underline{\underline{X}}$. Note that for the Gaussian correlation function

$$\frac{\partial^2 R_{j,j}}{\partial X^1_{j,I}\partial X^2_{j,I}} = 2\theta_I$$

and has units of length^{-2}. Two of the conditions necessary for a matrix to qualify as a correlation matrix are that all of its elements must be dimensionless and all of its diagonal elements must identically equal one. Since $\underline{\underline{R_\nabla}}$ does not satisfy these requirements, it technically does not qualify as a "correlation matrix." However, referring to $\underline{\underline{R_\nabla}}$ as such is a small abuse of terminology and allows GEK to use the same naming conventions as Kriging.

A straight-forward implementation of GEK tends be significantly more accurate than Kriging given the same sample design provided that the

- Derivatives are accurate
- Derivatives are not infinite (or nearly so)

- Function is sufficiently smooth, and
- $\underline{\underline{R_\nabla}}$ is not ill-conditioned (this can be problematic).

If gradients can be obtained cheaply (e.g. by automatic differentiation or adjoint techniques) and the previous conditions are met, GEK also tends to outperform Kriging for the same computational budget. Previous works, such as Dwight[24], state that the direct approach to GEK is significantly better conditioned than the indirect approach. While this is true, (direct) GEK's $\underline{\underline{R_\nabla}}$ matrix can still be, and often is, horribly ill-conditioned compared to Kriging's $\underline{\underline{R}}$ for the same $\underline{\theta}$ and $\underline{\underline{X}}$

In the literature, ill-conditioning is often attributed to the choice of the correlation function. Although a different correlation function may alleviate the ill-conditioning for some problems, the root cause of the ill-conditioning is a poorly spaced sample design. Furthermore, a sufficiently bad sample design could make any interpolatory Kriging model, gradient enhanced or otherwise, ill-conditioned, regardless of the choice of correlation function. This root cause can be addressed directly by discarding points/equations.

Discarding points/equations is conceptually similar to using a Moore-Penrose pseudo inverse of $\underline{\underline{R_\nabla}}$. However, there are important differences. A pseudo inverse handles ill-conditioning by discarding small singular values, which can be interpreted as throwing away the information that is least present while keeping all of what is most frequently duplicated. This causes a Kriging model to not interpolate any of the data points used to construct it while using some information from all rows.

An alternate strategy is to discard additional copies of the information that is most duplicated and keep more of the barely present information. In the context of eigenvalues, this can be described as decreasing the maximum eigenvalue and increasing the minimum eigenvalue by a smaller amount than a pseudo inverse. The result is that the GEK model will exactly fit all of the retained information. This can be achieved using a pivoted Cholesky factorization, such as the one developed by Lucas [54] to determine a reordering $\underline{\underline{\tilde{R}_\nabla}}$ and dropping equations off its end until it tightly meets the constraint on rcond. However, a straight-forward implementation of this is neither efficient nor robust.

In benchmarking tests, Lucas' level 3 pivoted Cholesky implementation was not competitive with the level 3 LAPACK non-pivoted Cholesky in terms of computational efficiency. In some cases, it was an order of magnitude slower. Note that Lucas' level 3 implementation can default to his level 2 implementation and this may explain some of the loss in performance.

More importantly, applying pivoted Cholesky to $\underline{\underline{R_\nabla}}$ tends to sort derivative equations to the top/front and function value equations to the end. This occurred even when $\underline{\underline{R_\nabla}}$ was equilibrated to have ones for all of its diagonal elements. The result was that for many points at least some of the derivative equations were retained while the function values at the same points were discarded. This can (and often did) significantly degrade the accuracy of the GEK predictions. The implication is that derivative equations contain more information than, but are not as reliable as, function value equations.

To address computational efficiency and robustness, DAKOTA's pivoted Cholesky approach for GEK was modified to:

- Equilibrate $\underline{\underline{R_\nabla}}$ to improve the accuracy of the Cholesky factorization; this is beneficial because $\underline{\underline{R_\nabla}}$ can be poorly scaled. Theorem 4.1 of van der Sluis [79] states that if $\underline{\underline{a}}$ is a real, symmetric, positive definite n by

Figure 4.1: A diagram with pseudo code for the pivoted Cholesky algorithm used to select the subset of equations to retain when $\underline{\underline{R}}_\nabla$ is ill-conditioned. Although it is part of the algorithm, the equilibration of $\underline{\underline{R}}_\nabla$ is not shown in this figure. The pseudo code uses MATLAB notation.

n matrix and the diagonal matrix $\underline{\underline{\alpha}}$ contains the square roots of the diagonals of $\underline{\underline{a}}$, then the equilibration

$$\breve{\underline{\underline{a}}} = \underline{\underline{\alpha}}^{-1} \underline{\underline{a}}\, \underline{\underline{\alpha}}^{-1},$$

minimizes the 2-norm condition number of $\breve{\underline{\underline{a}}}$ (with respect to solving linear systems) over all such symmetric scalings, to within a factor of n. The equilibrated matrix $\breve{\underline{\underline{a}}}$ will have all ones on the diagonal.

- Perform pivoted Cholesky on $\underline{\underline{R}}$, instead of $\underline{\underline{R_\nabla}}$, to rank points according to how much new information they contain. This ranking was reflected by the ordering of points in $\underline{\underline{\tilde{R}}}$.
- Apply the ordering of points in $\underline{\underline{\tilde{R}}}$ to whole points in $\underline{\underline{R_\nabla}}$ to produce $\underline{\underline{\tilde{R}_\nabla}}$. Here a whole point means the function value at a point immediately followed by the derivatives at the same point.
- Perform a LAPACK non-pivoted Cholesky on the equilibrated $\underline{\underline{\tilde{R}_\nabla}}$ and drop equations off the end until it satisfies the constraint on rcond. LAPACK's rcond estimate requires the 1-norm of the original (reordered) matrix as input so the 1-norms for all possible sizes of $\underline{\underline{\tilde{R}_\nabla}}$ are precomputed (using a rank one update approach) and stored prior to the Cholesky factorization. A bisection search is used to efficiently determine the number of equations that need to be retained/discarded. This requires $\mathrm{ceil}\left(\log 2\left(N_\nabla\right)\right)$ or fewer evaluations of rcond. These rcond calls are all based off the same Cholesky factorization of $\underline{\underline{\tilde{R}_\nabla}}$ but use different numbers of rows/columns, $\tilde{N}_\nabla$.

This algorithm is visually depicted in Figure 4.1. Because inverting/factorizing a matrix with n rows and columns requires $\mathcal{O}\left(n^3\right)$ flops, the cost to perform pivoted Cholesky on $\underline{\underline{R}}$ will be much less than, i.e. $\mathcal{O}\left((1+M)^{-3}\right)$, that of $\underline{\underline{R_\nabla}}$ when the number of dimensions M is large. It will also likely be negligible compared to the cost of performing LAPACK's non-pivoted Cholesky on $\underline{\underline{\tilde{R}_\nabla}}$.

Chapter 5

Surrogate-Based Local Minimization

A generally-constrained nonlinear programming problem takes the form

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}_l \leq \mathbf{g}(\mathbf{x}) \leq \mathbf{g}_u \\ & \mathbf{h}(\mathbf{x}) = \mathbf{h}_t \\ & \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u \end{aligned} \tag{5.1}$$

where $\mathbf{x} \in \Re^n$ is the vector of design variables, and f, $\mathbf{g}$, and $\mathbf{h}$ are the objective function, nonlinear inequality constraints, and nonlinear equality constraints, respectively[1]. Individual nonlinear inequality and equality constraints are enumerated using i and j, respectively (e.g., g_i and h_j). The corresponding surrogate-based optimization (SBO) algorithm may be formulated in several ways and applied to either optimization or least-squares calibration problems. In all cases, SBO solves a sequence of k approximate optimization subproblems subject to a trust region constraint Δ^k; however, many different forms of the surrogate objectives and constraints in the approximate subproblem can be explored. In particular, the subproblem objective may be a surrogate of the original objective or a surrogate of a merit function (most commonly, the Lagrangian or augmented Lagrangian), and the subproblem constraints may be surrogates of the original constraints, linearized approximations of the surrogate constraints, or may be omitted entirely. Each of these combinations is shown in Table 5.1, where black indicates an inappropriate combination, gray indicates an acceptable combination, and blue indicates a common combination.

Initial approaches to nonlinearly-constrained SBO optimized an approximate merit function which incorporated the nonlinear constraints [70, 4]:

$$\begin{aligned} \text{minimize} \quad & \hat{\Phi}^k(\mathbf{x}) \\ \text{subject to} \quad & \| \mathbf{x} - \mathbf{x}_c^k \|_\infty \leq \Delta^k \end{aligned} \tag{5.2}$$

[1] Any linear constraints are not approximated and may be added without modification to all formulations

Table 5.1: SBO approximate subproblem formulations.

	Original Objective	Lagrangian	Augmented Lagrangian
No constraints			TRAL
Linearized constraints		SQP-like	
Original constraints	Direct surrogate		IPTRSAO

where the surrogate merit function is denoted as $\hat{\Phi}(\mathbf{x})$, $\mathbf{x}_c$ is the center point of the trust region, and the trust region is truncated at the global variable bounds as needed. The merit function to approximate was typically chosen to be a standard implementation [80, 61, 41] of the augmented Lagrangian merit function (see Eqs. 5.11–5.12), where the surrogate augmented Lagrangian is constructed from individual surrogate models of the objective and constraints (approximate and assemble, rather than assemble and approximate). In Table 5.1, this corresponds to row 1, column 3, and is known as the trust-region augmented Lagrangian (TRAL) approach. While this approach was provably convergent, convergence rates to constrained minima have been observed to be slowed by the required updating of Lagrange multipliers and penalty parameters [64]. Prior to converging these parameters, SBO iterates did not strictly respect constraint boundaries and were often infeasible. A subsequent approach (IPTRSAO [64]) that sought to directly address this shortcoming added explicit surrogate constraints (row 3, column 3 in Table 5.1):

$$\begin{aligned} \text{minimize} \quad & \hat{\Phi}^k(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}_l \leq \hat{\mathbf{g}}^k(\mathbf{x}) \leq \mathbf{g}_u \\ & \hat{\mathbf{h}}^k(\mathbf{x}) = \mathbf{h}_t \\ & \| \mathbf{x} - \mathbf{x}_c^k \|_\infty \leq \Delta^k \ . \end{aligned} \tag{5.3}$$

While this approach does address infeasible iterates, it still shares the feature that the surrogate merit function may reflect inaccurate relative weightings of the objective and constraints prior to convergence of the Lagrange multipliers and penalty parameters. That is, one may benefit from more feasible intermediate iterates, but the process may still be slow to converge to optimality. The concept of this approach is similar to that of SQP-like SBO approaches [4] which use linearized constraints:

$$\begin{aligned} \text{minimize} \quad & \hat{\Phi}^k(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}_l \leq \hat{\mathbf{g}}^k(\mathbf{x}_c^k) + \nabla \hat{\mathbf{g}}^k(\mathbf{x}_c^k)^T(\mathbf{x} - \mathbf{x}_c^k) \leq \mathbf{g}_u \\ & \hat{\mathbf{h}}^k(\mathbf{x}_c^k) + \nabla \hat{\mathbf{h}}^k(\mathbf{x}_c^k)^T(\mathbf{x} - \mathbf{x}_c^k) = \mathbf{h}_t \\ & \| \mathbf{x} - \mathbf{x}_c^k \|_\infty \leq \Delta^k \ . \end{aligned} \tag{5.4}$$

in that the primary concern is minimizing a composite merit function of the objective and constraints, but under the restriction that the original problem constraints may not be wildly violated prior to convergence of Lagrange multiplier estimates. Here, the merit function selection of the Lagrangian function (row 2, column 2 in Table 5.1; see also Eq. 5.10) is most closely related to SQP, which includes the use of first-order Lagrange multiplier updates (Eq. 5.16) that should converge more rapidly near a constrained minimizer than the zeroth-order updates (Eqs. 5.13-5.14) used for the augmented Lagrangian.

All of these previous constrained SBO approaches involve a recasting of the approximate subproblem objective and constraints as a function of the original objective and constraint surrogates. A more direct approach is to use a formulation of:

$$\begin{aligned} \text{minimize} \quad & \hat{f}^k(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}_l \leq \hat{\mathbf{g}}^k(\mathbf{x}) \leq \mathbf{g}_u \\ & \hat{\mathbf{h}}^k(\mathbf{x}) = \mathbf{h}_t \\ & \| \mathbf{x} - \mathbf{x}_c^k \|_\infty \leq \Delta^k \end{aligned} \tag{5.5}$$

This approach has been termed the direct surrogate approach since it optimizes surrogates of the original objective and constraints (row 3, column 1 in Table 5.1) without any recasting. It is attractive both from its simplicity and potential for improved performance, and is the default approach taken in DAKOTA. Other DAKOTA defaults include the use of a filter method for iterate acceptance (see Section 5.1), an augmented Lagrangian merit function (see Section 5.2), Lagrangian hard convergence assessment (see Section 5.3), and no constraint relaxation (see Section 5.4).

Table 5.2: Sample trust region ratio logic.

Ratio Value	Surrogate Accuracy	Iterate Acceptance	Trust Region Sizing
$\rho^k \leq 0$	poor	reject step	shrink
$0 < \rho^k \leq 0.25$	marginal	accept step	shrink
$0.25 < \rho^k < 0.75$ or $\rho^k > 1.25$	moderate	accept step	retain
$0.75 \leq \rho^k \leq 1.25$	good	accept step	expand[2]

While the formulation of Eq. 5.2 (and others from row 1 in Table 5.1) can suffer from infeasible intermediate iterates and slow convergence to constrained minima, each of the approximate subproblem formulations with explicit constraints (Eqs. 5.3-5.5, and others from rows 2-3 in Table 5.1) can suffer from the lack of a feasible solution within the current trust region. Techniques for dealing with this latter challenge involve some form of constraint relaxation. Homotopy approaches [64, 63] or composite step approaches such as Byrd-Omojokun [62], Celis-Dennis-Tapia [14], or MAESTRO [4] may be used for this purpose (see Section 5.4).

After each of the k iterations in the SBO method, the predicted step is validated by computing $f(\mathbf{x}_*^k)$, $\mathbf{g}(\mathbf{x}_*^k)$, and $\mathbf{h}(\mathbf{x}_*^k)$. One approach forms the trust region ratio ρ^k which measures the ratio of the actual improvement to the improvement predicted by optimization on the surrogate model. When optimizing on an approximate merit function (Eqs. 5.2–5.4), the following ratio is natural to compute

$$\rho^k = \frac{\Phi(\mathbf{x}_c^k) - \Phi(\mathbf{x}_*^k)}{\hat{\Phi}(\mathbf{x}_c^k) - \hat{\Phi}(\mathbf{x}_*^k)} \ . \tag{5.6}$$

The formulation in Eq. 5.5 may also form a merit function for computing the trust region ratio; however, the omission of this merit function from explicit use in the approximate optimization cycles can lead to synchronization problems with the optimizer.

Once computed, the value for ρ^k can be used to define the step acceptance and the next trust region size Δ^{k+1} using logic similar to that shown in Table 5.2. Typical factors for shrinking and expanding are 0.5 and 2.0, respectively, but these as well as the threshold ratio values are tunable parameters in the algorithm (see Surrogate-Based Method controls in the DAKOTA Reference Manual [3]). In addition, the use of discrete thresholds is not required, and continuous relationships using adaptive logic can also be explored [87, 88]. Iterate acceptance or rejection completes an SBO cycle, and the cycles are continued until either soft or hard convergence criteria (see Section 5.3) are satisfied.

5.1 Iterate acceptance logic

When a surrogate optimization is completed and the approximate solution has been validated, then the decision must be made to either accept or reject the step. The traditional approach is to base this decision on the value of the trust region ratio, as outlined previously in Table 5.2. An alternate approach is to utilize a filter method [34], which does not require penalty parameters or Lagrange multiplier estimates. The basic idea in a filter method is to apply the concept of Pareto optimality to the objective function and constraint violations and only accept an iterate if it is not dominated by any previous iterate. Mathematically, a new iterate is not dominated if at least one of the following:

$$\text{either} \quad f < f^{(i)} \quad \text{or} \quad c < c^{(i)} \tag{5.7}$$

Figure 5.1: Depiction of filter method.

[2]Exception: retain if $\mathbf{x}_*^k$ in trust region interior for design of experiments-based surrogates (global data fits, S-ROM, global E-ROM)

is true for all i in the filter, where c is a selected norm of the constraint violation. This basic description can be augmented with mild requirements to prevent point accumulation and assure convergence, known as a slanting filter [34]. Figure 5.1 illustrates the filter concept, where objective values are plotted against constraint violation for accepted iterates (blue circles) to define the dominated region (denoted by the gray lines). A filter method relaxes the common enforcement of monotonicity in constraint violation reduction and, by allowing more flexibility in acceptable step generation, often allows the algorithm to be more efficient.

The use of a filter method is compatible with any of the SBO formulations in Eqs. 5.2–5.5.

5.2 Merit functions

The merit function $\Phi(\mathbf{x})$ used in Eqs. 5.2-5.4,5.6 may be selected to be a penalty function, an adaptive penalty function, a Lagrangian function, or an augmented Lagrangian function. In each of these cases, the more flexible inequality and equality constraint formulations with two-sided bounds and targets (Eqs. 5.1,5.3-5.5), have been converted to a standard form of $\mathbf{g}(\mathbf{x}) \leq 0$ and $\mathbf{h}(\mathbf{x}) = 0$ (in Eqs. 5.8,5.10-5.16). The active set of inequality constraints is denoted as $\mathbf{g}^+$.

The penalty function employed in this paper uses a quadratic penalty with the penalty schedule linked to SBO iteration number

$$\Phi(\mathbf{x}, r_p) = f(\mathbf{x}) + r_p \mathbf{g}^+(\mathbf{x})^T \mathbf{g}^+(\mathbf{x}) + r_p \mathbf{h}(\mathbf{x})^T \mathbf{h}(\mathbf{x}) \tag{5.8}$$

$$r_p = e^{(k+\text{offset})/10} \tag{5.9}$$

The adaptive penalty function is identical in form to Eq. 5.8, but adapts r_p using monotonic increases in the iteration offset value in order to accept any iterate that reduces the constraint violation.

The Lagrangian merit function is

$$\Phi(\mathbf{x}, \boldsymbol{\lambda}_g, \boldsymbol{\lambda}_h) = f(\mathbf{x}) + \boldsymbol{\lambda}_g^T \mathbf{g}^+(\mathbf{x}) + \boldsymbol{\lambda}_h^T \mathbf{h}(\mathbf{x}) \tag{5.10}$$

for which the Lagrange multiplier estimation is discussed in Section 5.3. Away from the optimum, it is possible for the least squares estimates of the Lagrange multipliers for active constraints to be zero, which equates to omitting the contribution of an active constraint from the merit function. This is undesirable for tracking SBO progress, so usage of the Lagrangian merit function is normally restricted to approximate subproblems and hard convergence assessments.

The augmented Lagrangian employed in this paper follows the sign conventions described in [80]

$$\Phi(\mathbf{x}, \boldsymbol{\lambda}_\psi, \boldsymbol{\lambda}_h, r_p) = f(\mathbf{x}) + \boldsymbol{\lambda}_\psi^T \boldsymbol{\psi}(\mathbf{x}) + r_p \boldsymbol{\psi}(\mathbf{x})^T \boldsymbol{\psi}(\mathbf{x}) + \boldsymbol{\lambda}_h^T \mathbf{h}(\mathbf{x}) + r_p \mathbf{h}(\mathbf{x})^T \mathbf{h}(\mathbf{x}) \tag{5.11}$$

$$\psi_i = \max\left\{ g_i, -\frac{\lambda_{\psi_i}}{2r_p} \right\} \tag{5.12}$$

where $\boldsymbol{\psi}(\mathbf{x})$ is derived from the elimination of slack variables for the inequality constraints. In this case, simple zeroth-order Lagrange multiplier updates may be used:

$$\boldsymbol{\lambda}_\psi^{k+1} = \boldsymbol{\lambda}_\psi^k + 2r_p \boldsymbol{\psi}(\mathbf{x}) \tag{5.13}$$

$$\boldsymbol{\lambda}_h^{k+1} = \boldsymbol{\lambda}_h^k + 2r_p \mathbf{h}(\mathbf{x}) \tag{5.14}$$

The updating of multipliers and penalties is carefully orchestrated [16] to drive reduction in constraint violation of the iterates. The penalty updates can be more conservative than in Eq. 5.9, often using an infrequent application of a constant multiplier rather than a fixed exponential progression.

5.3 Convergence assessment

To terminate the SBO process, hard and soft convergence metrics are monitored. It is preferable for SBO studies to satisfy hard convergence metrics, but this is not always practical (e.g., when gradients are unavailable or unreliable). Therefore, simple soft convergence criteria are also employed which monitor for diminishing returns (relative improvement in the merit function less than a tolerance for some number of consecutive iterations).

To assess hard convergence, one calculates the norm of the projected gradient of a merit function whenever the feasibility tolerance is satisfied. The best merit function for this purpose is the Lagrangian merit function from Eq. 5.10. This requires a least squares estimation for the Lagrange multipliers that best minimize the projected gradient:

$$\nabla_x \Phi(\mathbf{x}, \boldsymbol{\lambda}_g, \boldsymbol{\lambda}_h) = \nabla_x f(\mathbf{x}) + \boldsymbol{\lambda}_g^T \nabla_x \mathbf{g}^+(\mathbf{x}) + \boldsymbol{\lambda}_h^T \nabla_x \mathbf{h}(\mathbf{x}) \tag{5.15}$$

where gradient portions directed into active global variable bounds have been removed. This can be posed as a linear least squares problem for the multipliers:

$$\mathbf{A}\boldsymbol{\lambda} = -\nabla_x f \tag{5.16}$$

where $\mathbf{A}$ is the matrix of active constraint gradients, $\boldsymbol{\lambda}_g$ is constrained to be non-negative, and $\boldsymbol{\lambda}_h$ is unrestricted in sign. To estimate the multipliers using non-negative and bound-constrained linear least squares, the NNLS and BVLS routines [53] from NETLIB are used, respectively.

5.4 Constraint relaxation

The goal of constraint relaxation is to achieve efficiency through the balance of feasibility and optimality when the trust region restrictions prevent the location of feasible solutions to constrained approximate subproblems (Eqs. 5.3-5.5, and other formulations from rows 2-3 in Table 5.1). The SBO algorithm starting from infeasible points will commonly generate iterates which seek to satisfy feasibility conditions without regard to objective reduction [63].

One approach for achieving this balance is to use *relaxed constraints* when iterates are infeasible with respect to the surrogate constraints. We follow Perez, Renaud, and Watson [64], and use a *global homotopy* mapping the relaxed constraints and the surrogate constraints. For formulations in Eqs. 5.3 and 5.5 (and others from row 3 in Table 5.1), the relaxed constraints are defined from

$$\tilde{\mathbf{g}}^k(\mathbf{x}, \tau) = \hat{\mathbf{g}}^k(\mathbf{x}) + (1-\tau)\mathbf{b}_g \tag{5.17}$$

$$\tilde{\mathbf{h}}^k(\mathbf{x}, \tau) = \hat{\mathbf{h}}^k(\mathbf{x}) + (1-\tau)\mathbf{b}_h \tag{5.18}$$

For Eq. 5.4 (and others from row 2 in Table 5.1), the original surrogate constraints $\hat{\mathbf{g}}^k(\mathbf{x})$ and $\hat{\mathbf{h}}^k(\mathbf{x})$ in Eqs. 5.17-5.18 are replaced with their linearized forms ($\hat{\mathbf{g}}^k(\mathbf{x}_c^k) + \nabla\hat{\mathbf{g}}^k(\mathbf{x}_c^k)^T(\mathbf{x} - \mathbf{x}_c^k)$ and $\hat{\mathbf{h}}^k(\mathbf{x}_c^k) + \nabla\hat{\mathbf{h}}^k(\mathbf{x}_c^k)^T(\mathbf{x} - \mathbf{x}_c^k)$, respectively). The approximate subproblem is then reposed using the relaxed constraints as

$$\begin{array}{ll} \text{minimize} & \hat{f}^k(\mathbf{x}) \ \text{ or } \ \hat{\Phi}^k(\mathbf{x}) \\ \text{subject to} & \mathbf{g}_l \leq \tilde{\mathbf{g}}^k(\mathbf{x}, \tau^k) \leq \mathbf{g}_u \\ & \tilde{\mathbf{h}}^k(\mathbf{x}, \tau^k) = \mathbf{h}_t \\ & \| \mathbf{x} - \mathbf{x}_c^k \|_\infty \leq \Delta^k \end{array} \tag{5.19}$$

in place of the corresponding subproblems in Eqs. 5.3-5.5. Alternatively, since the relaxation terms are constants for the kth iteration, it may be more convenient for the implementation to constrain $\hat{\mathbf{g}}^k(\mathbf{x})$ and $\hat{\mathbf{h}}^k(\mathbf{x})$ (or their

linearized forms) subject to relaxed bounds and targets ($\tilde{\mathbf{g}}_l^k$, $\tilde{\mathbf{g}}_u^k$, $\tilde{\mathbf{h}}_t^k$). The parameter τ is the homotopy parameter controlling the extent of the relaxation: when $\tau = 0$, the constraints are fully relaxed, and when $\tau = 1$, the surrogate constraints are recovered. The vectors $\mathbf{b}_g, \mathbf{b}_h$ are chosen so that the starting point, $\mathbf{x}^0$, is feasible with respect to the fully relaxed constraints:

$$\mathbf{g}_l \leq \tilde{\mathbf{g}}^0(\mathbf{x}^0, 0) \leq \mathbf{g}_u \tag{5.20}$$

$$\tilde{\mathbf{h}}^0(\mathbf{x}^0, 0) = \mathbf{h}_t \tag{5.21}$$

At the start of the SBO algorithm, $\tau^0 = 0$ if $\mathbf{x}^0$ is infeasible with respect to the unrelaxed surrogate constraints; otherwise $\tau^0 = 1$ (i.e., no constraint relaxation is used). At the start of the kth SBO iteration where $\tau^{k-1} < 1$, τ^k is determined by solving the subproblem

$$\begin{aligned} \text{maximize} \quad & \tau^k \\ \text{subject to} \quad & \mathbf{g}_l \leq \tilde{\mathbf{g}}^k(\mathbf{x}, \tau^k) \leq \mathbf{g}_u \\ & \tilde{\mathbf{h}}^k(\mathbf{x}, \tau^k) = \mathbf{h}_t \\ & \| \mathbf{x} - \mathbf{x}_c^k \|_\infty \leq \Delta^k \\ & \tau^k \geq 0 \end{aligned} \tag{5.22}$$

starting at $(\mathbf{x}_*^{k-1}, \tau^{k-1})$, and then adjusted as follows:

$$\tau^k = \min\left\{1, \tau^{k-1} + \alpha\left(\tau_{\max}^k - \tau^{k-1}\right)\right\} \tag{5.23}$$

The adjustment parameter $0 < \alpha < 1$ is chosen so that that the feasible region with respect to the relaxed constraints has positive volume within the trust region. Determining the optimal value for α remains an open question and will be explored in future work.

After τ^k is determined using this procedure, the problem in Eq. 5.19 is solved for $\mathbf{x}_*^k$. If the step is accepted, then the value of τ^k is updated using the current iterate $\mathbf{x}_*^k$ and the validated constraints $\mathbf{g}(\mathbf{x}_*^k)$ and $\mathbf{h}(\mathbf{x}_*^k)$:

$$\tau^k = \min\{1, \min_i \tau_i, \min_j \tau_j\} \tag{5.24}$$

$$\text{where } \tau_i = 1 + \frac{\min\left\{g_i(\mathbf{x}_*^k) - g_{l_i}, g_{u_i} - g_i(\mathbf{x}_*^k)\right\}}{b_{g_i}} \tag{5.25}$$

$$\tau_j = 1 - \frac{|h_j(\mathbf{x}_*^k) - h_{t_j}|}{b_{h_j}} \tag{5.26}$$

Figure 5.2 illustrates the SBO algorithm on a two-dimensional problem with one inequality constraint starting from an infeasible point, $\mathbf{x}^0$. The minimizer of the problem is denoted as $\mathbf{x}^*$. Iterates generated using the surrogate constraints are shown in red, where feasibility is achieved first, and then progress is made toward the optimal point. The iterates generated using the relaxed constraints are shown in blue, where a balance of satisfying feasibility and optimality has been achieved, leading to fewer overall SBO iterations.

Figure 5.2: Illustration of SBO iterates using surrogate (red) and relaxed (blue) constraints.

The behavior illustrated in Fig. 5.2 is an example where using the relaxed constraints over the surrogate constraints may improve the overall performance of the SBO algorithm by reducing the number of iterations performed. This improvement comes at the cost of solving the minimization subproblem in Eq. 5.22, which can be significant in some cases (i.e., when the cost of evaluating $\hat{\mathbf{g}}^k(\mathbf{x})$ and $\hat{\mathbf{h}}^k(\mathbf{x})$ is not negligible, such as with multifidelity or ROM surrogates). As shown in the numerical experiments involving the Barnes

problem presented in [64], the directions toward constraint violation reduction and objective function reduction may be in opposing directions. In such cases, the use of the relaxed constraints may result in an *increase* in the overall number of SBO iterations since feasibility must ultimately take precedence.

Chapter 6

Optimization Under Uncertainty (OUU)

6.1 Reliability-Based Design Optimization (RBDO)

Reliability-based design optimization (RBDO) methods are used to perform design optimization accounting for reliability metrics. The reliability analysis capabilities described in Section 1.1 provide a substantial foundation for exploring a variety of gradient-based RBDO formulations. [25] investigated bi-level, fully-analytic bi-level, and first-order sequential RBDO approaches employing underlying first-order reliability assessments. [26] investigated fully-analytic bi-level and second-order sequential RBDO approaches employing underlying second-order reliability assessments. These methods are overviewed in the following sections.

6.1.1 Bi-level RBDO

The simplest and most direct RBDO approach is the bi-level approach in which a full reliability analysis is performed for every optimization function evaluation. This involves a nesting of two distinct levels of optimization within each other, one at the design level and one at the MPP search level.

Since an RBDO problem will typically specify both the $\bar{z}$ level and the $\bar{p}/\bar{\beta}$ level, one can use either the RIA or the PMA formulation for the UQ portion and then constrain the result in the design optimization portion. In particular, RIA reliability analysis maps $\bar{z}$ to p/β, so RIA RBDO constrains p/β:

$$\begin{aligned} \text{minimize} \quad & f \\ \text{subject to} \quad & \beta \geq \bar{\beta} \\ \text{or} \quad & p \leq \bar{p} \end{aligned} \tag{6.1}$$

And PMA reliability analysis maps $\bar{p}/\bar{\beta}$ to z, so PMA RBDO constrains z:

$$\begin{aligned} \text{minimize} \quad & f \\ \text{subject to} \quad & z \geq \bar{z} \end{aligned} \tag{6.2}$$

where $z \geq \bar{z}$ is used as the RBDO constraint for a cumulative failure probability (failure defined as $z \leq \bar{z}$) but $z \leq \bar{z}$ would be used as the RBDO constraint for a complementary cumulative failure probability (failure defined as $z \geq \bar{z}$). It is worth noting that DAKOTA is not limited to these types of inequality-constrained RBDO formulations; rather, they are convenient examples. DAKOTA supports general optimization under uncertainty

mappings [30] which allow flexible use of statistics within multiple objectives, inequality constraints, and equality constraints.

An important performance enhancement for bi-level methods is the use of sensitivity analysis to analytically compute the design gradients of probability, reliability, and response levels. When design variables are separate from the uncertain variables (i.e., they are not distribution parameters), then the following first-order expressions may be used [46, 52, 5]:

$$\nabla_{\mathbf{d}} z = \nabla_{\mathbf{d}} g \tag{6.3}$$

$$\nabla_{\mathbf{d}} \beta_{cdf} = \frac{1}{\| \nabla_{\mathbf{u}} G \|} \nabla_{\mathbf{d}} g \tag{6.4}$$

$$\nabla_{\mathbf{d}} p_{cdf} = -\phi(-\beta_{cdf}) \nabla_{\mathbf{d}} \beta_{cdf} \tag{6.5}$$

where it is evident from Eqs. 1.10-1.11 that $\nabla_{\mathbf{d}} \beta_{ccdf} = -\nabla_{\mathbf{d}} \beta_{cdf}$ and $\nabla_{\mathbf{d}} p_{ccdf} = -\nabla_{\mathbf{d}} p_{cdf}$. In the case of second-order integrations, Eq. 6.5 must be expanded to include the curvature correction. For Breitung's correction (Eq. 1.37),

$$\nabla_{\mathbf{d}} p_{cdf} = \left[\Phi(-\beta_p) \sum_{i=1}^{n-1} \left(\frac{-\kappa_i}{2(1+\beta_p \kappa_i)^{\frac{3}{2}}} \prod_{\substack{j=1 \\ j \neq i}}^{n-1} \frac{1}{\sqrt{1+\beta_p \kappa_j}} \right) - \phi(-\beta_p) \prod_{i=1}^{n-1} \frac{1}{\sqrt{1+\beta_p \kappa_i}} \right] \nabla_{\mathbf{d}} \beta_{cdf} \tag{6.6}$$

where $\nabla_{\mathbf{d}} \kappa_i$ has been neglected and $\beta_p \geq 0$ (see Section 1.1.2.2). Other approaches assume the curvature correction is nearly independent of the design variables [65], which is equivalent to neglecting the first term in Eq. 6.6.

To capture second-order probability estimates within an RIA RBDO formulation using well-behaved β constraints, a generalized reliability index can be introduced where, similar to Eq. 1.8,

$$\beta^*_{cdf} = -\Phi^{-1}(p_{cdf}) \tag{6.7}$$

for second-order p_{cdf}. This reliability index is no longer equivalent to the magnitude of $\mathbf{u}$, but rather is a convenience metric for capturing the effect of more accurate probability estimates. The corresponding generalized reliability index sensitivity, similar to Eq. 6.5, is

$$\nabla_{\mathbf{d}} \beta^*_{cdf} = -\frac{1}{\phi(-\beta^*_{cdf})} \nabla_{\mathbf{d}} p_{cdf} \tag{6.8}$$

where $\nabla_{\mathbf{d}} p_{cdf}$ is defined from Eq. 6.6. Even when $\nabla_{\mathbf{d}} g$ is estimated numerically, Eqs. 6.3-6.8 can be used to avoid numerical differencing across full reliability analyses.

When the design variables are distribution parameters of the uncertain variables, $\nabla_{\mathbf{d}} g$ is expanded with the chain rule and Eqs. 6.3 and 6.4 become

$$\nabla_{\mathbf{d}} z = \nabla_{\mathbf{d}} \mathbf{x} \nabla_{\mathbf{x}} g \tag{6.9}$$

$$\nabla_{\mathbf{d}} \beta_{cdf} = \frac{1}{\| \nabla_{\mathbf{u}} G \|} \nabla_{\mathbf{d}} \mathbf{x} \nabla_{\mathbf{x}} g \tag{6.10}$$

where the design Jacobian of the transformation ($\nabla_{\mathbf{d}} \mathbf{x}$) may be obtained analytically for uncorrelated $\mathbf{x}$ or semi-analytically for correlated $\mathbf{x}$ ($\nabla_{\mathbf{d}} \mathbf{L}$ is evaluated numerically) by differentiating Eqs. 1.13 and 1.14 with respect to the distribution parameters. Eqs. 6.5-6.8 remain the same as before. For this design variable case, all required information for the sensitivities is available from the MPP search.

Since Eqs. 6.3-6.10 are derived using the Karush-Kuhn-Tucker optimality conditions for a converged MPP, they are appropriate for RBDO using AMV+, AMV^2+, TANA, FORM, and SORM, but not for RBDO using MV-FOSM, MVSOSM, AMV, or AMV^2.

6.1.2 Sequential/Surrogate-based RBDO

An alternative RBDO approach is the sequential approach, in which additional efficiency is sought through breaking the nested relationship of the MPP and design searches. The general concept is to iterate between optimization and uncertainty quantification, updating the optimization goals based on the most recent probabilistic assessment results. This update may be based on safety factors [85] or other approximations [23].

A particularly effective approach for updating the optimization goals is to use the $p/\beta/z$ sensitivity analysis of Eqs. 6.3-6.10 in combination with local surrogate models [92]. In [25] and [26], first-order and second-order Taylor series approximations were employed within a trust-region model management framework [42] in order to adaptively manage the extent of the approximations and ensure convergence of the RBDO process. Surrogate models were used for both the objective function and the constraints, although the use of constraint surrogates alone is sufficient to remove the nesting.

In particular, RIA trust-region surrogate-based RBDO employs surrogate models of f and p/β within a trust region Δ^k centered at $\mathbf{d}_c$. For first-order surrogates:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{d}_c) + \nabla_d f(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) \\ \text{subject to} \quad & \beta(\mathbf{d}_c) + \nabla_d \beta(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) \geq \bar{\beta} \\ \text{or} \quad & p(\mathbf{d}_c) + \nabla_d p(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) \leq \bar{p} \\ & \| \mathbf{d}-\mathbf{d}_c \|_\infty \leq \Delta^k \end{aligned} \tag{6.11}$$

and for second-order surrogates:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{d}_c) + \nabla_{\mathbf{d}} f(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) + \tfrac{1}{2}(\mathbf{d}-\mathbf{d}_c)^T\nabla^2_{\mathbf{d}} f(\mathbf{d}_c)(\mathbf{d}-\mathbf{d}_c) \\ \text{subject to} \quad & \beta(\mathbf{d}_c) + \nabla_{\mathbf{d}} \beta(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) + \tfrac{1}{2}(\mathbf{d}-\mathbf{d}_c)^T\nabla^2_{\mathbf{d}} \beta(\mathbf{d}_c)(\mathbf{d}-\mathbf{d}_c) \geq \bar{\beta} \\ \text{or} \quad & p(\mathbf{d}_c) + \nabla_{\mathbf{d}} p(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) + \tfrac{1}{2}(\mathbf{d}-\mathbf{d}_c)^T\nabla^2_{\mathbf{d}} p(\mathbf{d}_c)(\mathbf{d}-\mathbf{d}_c) \leq \bar{p} \\ & \| \mathbf{d}-\mathbf{d}_c \|_\infty \leq \Delta^k \end{aligned} \tag{6.12}$$

For PMA trust-region surrogate-based RBDO, surrogate models of f and z are employed within a trust region Δ^k centered at $\mathbf{d}_c$. For first-order surrogates:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{d}_c) + \nabla_d f(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) \\ \text{subject to} \quad & z(\mathbf{d}_c) + \nabla_d z(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) \geq \bar{z} \\ & \| \mathbf{d}-\mathbf{d}_c \|_\infty \leq \Delta^k \end{aligned} \tag{6.13}$$

and for second-order surrogates:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{d}_c) + \nabla_{\mathbf{d}} f(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) + \tfrac{1}{2}(\mathbf{d}-\mathbf{d}_c)^T\nabla^2_{\mathbf{d}} f(\mathbf{d}_c)(\mathbf{d}-\mathbf{d}_c) \\ \text{subject to} \quad & z(\mathbf{d}_c) + \nabla_{\mathbf{d}} z(\mathbf{d}_c)^T(\mathbf{d}-\mathbf{d}_c) + \tfrac{1}{2}(\mathbf{d}-\mathbf{d}_c)^T\nabla^2_{\mathbf{d}} z(\mathbf{d}_c)(\mathbf{d}-\mathbf{d}_c) \geq \bar{z} \\ & \| \mathbf{d}-\mathbf{d}_c \|_\infty \leq \Delta^k \end{aligned} \tag{6.14}$$

where the sense of the z constraint may vary as described previously. The second-order information in Eqs. 6.12 and 6.14 will typically be approximated with quasi-Newton updates.

6.2 Stochastic Expansion-Based Design Optimization (SEBDO)

6.2.1 Stochastic Sensitivity Analysis

Section 2.9 describes sensitivity analysis of the polynomial chaos expansion with respect to the expansion variables. Here we extend this analysis to include sensitivity analysis of probabilistic moments with respect to non-

probabilistic (i.e., design or epistemic uncertain) variables.

6.2.1.1 Local sensitivity analysis: first-order probabilistic expansions

With the introduction of nonprobabilistic variables $\boldsymbol{s}$ (for example, design variables or epistemic uncertain variables), a polynomial chaos expansion only over the probabilistic variables $\boldsymbol{\xi}$ has the functional relationship:

$$R(\boldsymbol{\xi}, \boldsymbol{s}) \cong \sum_{j=0}^{P} \alpha_j(\boldsymbol{s}) \Psi_j(\boldsymbol{\xi}) \tag{6.15}$$

For computing sensitivities of response mean and variance, the ij indices may be dropped from Eqs. 2.37 and 2.38, simplifying to

$$\mu(s) \;=\; \alpha_0(s), \quad \sigma^2(s) = \sum_{k=1}^{P} \alpha_k^2(s) \langle \Psi_k^2 \rangle \tag{6.16}$$

Sensitivities of Eq. 6.16 with respect to the nonprobabilistic variables are as follows, where independence of $\boldsymbol{s}$ and $\boldsymbol{\xi}$ is assumed:

$$\frac{d\mu}{ds} \;=\; \frac{d\alpha_0}{ds} \;=\; \langle \frac{dR}{ds} \rangle \tag{6.17}$$

$$\frac{d\sigma^2}{ds} \;=\; \sum_{k=1}^{P} \langle \Psi_k^2 \rangle \frac{d\alpha_k^2}{ds} \;=\; 2 \sum_{k=1}^{P} \alpha_k \langle \frac{dR}{ds}, \Psi_k \rangle \tag{6.18}$$

where

$$\frac{d\alpha_k}{ds} = \frac{\langle \frac{dR}{ds}, \Psi_k \rangle}{\langle \Psi_k^2 \rangle} \tag{6.19}$$

has been used. Due to independence, the coefficients calculated in Eq. 6.19 may be interpreted as either the derivatives of the expectations or the expectations of the derivatives, or more precisely, the nonprobabilistic sensitivities of the chaos coefficients for the response expansion or the chaos coefficients of an expansion for the nonprobabilistic sensitivities of the response. The evaluation of integrals involving $\frac{dR}{ds}$ extends the data requirements for the PCE approach to include response sensitivities at each of the sampled points.The resulting expansions are valid only for a particular set of nonprobabilistic variables and must be recalculated each time the nonprobabilistic variables are modified.

Similarly for stochastic collocation,

$$R(\boldsymbol{\xi}, \boldsymbol{s}) \cong \sum_{k=1}^{N_p} r_k(\boldsymbol{s}) \boldsymbol{L}_k(\boldsymbol{\xi}) \tag{6.20}$$

leads to

$$\mu(s) \;=\; \sum_{k=1}^{N_p} r_k(s) w_k, \quad \sigma^2(s) \;=\; \sum_{k=1}^{N_p} r_k^2(s) w_k - \mu^2(s) \tag{6.21}$$

$$\frac{d\mu}{ds} \;=\; \sum_{k=1}^{N_p} w_k \frac{dr_k}{ds} \tag{6.22}$$

$$\frac{d\sigma^2}{ds} \;=\; \sum_{k=1}^{N_p} 2 w_k r_k \frac{dr_k}{ds} - 2\mu \frac{d\mu}{ds} \;=\; \sum_{k=1}^{N_p} 2 w_k (r_k - \mu) \frac{dr_k}{ds} \tag{6.23}$$

6.2.1.2 Local sensitivity analysis: zeroth-order combined expansions

Alternatively, a stochastic expansion can be formed over both $\boldsymbol{\xi}$ and $\boldsymbol{s}$. Assuming a bounded design domain $\boldsymbol{s}_L \leq \boldsymbol{s} \leq \boldsymbol{s}_U$ (with no implied probability content), a Legendre chaos basis would be appropriate for each of the dimensions in $\boldsymbol{s}$ within a polynomial chaos expansion.

$$R(\boldsymbol{\xi}, \boldsymbol{s}) \cong \sum_{j=0}^{P} \alpha_j \Psi_j(\boldsymbol{\xi}, \boldsymbol{s}) \tag{6.24}$$

In this case, design sensitivities for the mean and variance do not require response sensitivity data, but this comes at the cost of forming the PCE over additional dimensions. For this combined variable expansion, the mean and variance are evaluated by performing the expectations over only the probabilistic expansion variables, which eliminates the polynomial dependence on $\boldsymbol{\xi}$, leaving behind the desired polynomial dependence of the moments on $\boldsymbol{s}$:

$$\mu_R(\boldsymbol{s}) = \sum_{j=0}^{P} \alpha_j \langle \Psi_j(\boldsymbol{\xi}, \boldsymbol{s}) \rangle_{\boldsymbol{\xi}} \tag{6.25}$$

$$\sigma_R^2(\boldsymbol{s}) = \sum_{j=0}^{P} \sum_{k=0}^{P} \alpha_j \alpha_k \langle \Psi_j(\boldsymbol{\xi}, \boldsymbol{s}) \Psi_k(\boldsymbol{\xi}, \boldsymbol{s}) \rangle_{\boldsymbol{\xi}} - \mu_R^2(\boldsymbol{s}) \tag{6.26}$$

The remaining polynomials may then be differentiated with respect to $\boldsymbol{s}$. In this approach, the combined PCE is valid for the full design variable range ($\boldsymbol{s}_L \leq \boldsymbol{s} \leq \boldsymbol{s}_U$) and does not need to be updated for each change in nonprobabilistic variables, although adaptive localization techniques (i.e., trust region model management approaches) can be employed when improved local accuracy of the sensitivities is required.

Similarly for stochastic collocation,

$$R(\boldsymbol{\xi}, \boldsymbol{s}) \cong \sum_{j=1}^{N_p} r_j \boldsymbol{L}_j(\boldsymbol{\xi}, \boldsymbol{s}) \tag{6.27}$$

leads to

$$\mu_R(\boldsymbol{s}) = \sum_{j=1}^{N_p} r_j \langle \boldsymbol{L}_j(\boldsymbol{\xi}, \boldsymbol{s}) \rangle_{\boldsymbol{\xi}} \tag{6.28}$$

$$\sigma_R^2(\boldsymbol{s}) = \sum_{j=1}^{N_p} \sum_{k=1}^{N_p} r_j r_k \langle \boldsymbol{L}_j(\boldsymbol{\xi}, \boldsymbol{s}) \boldsymbol{L}_k(\boldsymbol{\xi}, \boldsymbol{s}) \rangle_{\boldsymbol{\xi}} - \mu_R^2(\boldsymbol{s}) \tag{6.29}$$

where the remaining polynomials not eliminated by the expectation over $\boldsymbol{\xi}$ are again differentiated with respect to $\boldsymbol{s}$.

6.2.1.3 Inputs and outputs

There are two types of nonprobabilistic variables for which sensitivities must be calculated: "augmented," where the nonprobabilistic variables are separate from and augment the probabilistic variables, and "inserted," where the nonprobabilistic variables define distribution parameters for the probabilistic variables. Any inserted nonprobabilistic variable sensitivities must be handled using Eqs. 6.17-6.18 and Eqs. 6.22-6.23 where $\frac{dR}{ds}$ is calculated as

$\frac{dR}{dx}\frac{dx}{ds}$ and $\frac{dx}{ds}$ is the Jacobian of the variable transformation $\mathbf{x} = T^{-1}(\boldsymbol{\xi})$ with respect to the inserted nonprobabilistic variables. In addition, parameterized polynomials (generalized Gauss-Laguerre, Jacobi, and numerically-generated polynomials) may introduce a $\frac{d\Psi}{ds}$ or $\frac{d\boldsymbol{L}}{ds}$ dependence for inserted s that will introduce additional terms in the sensitivity expressions.

While moment sensitivities directly enable robust design optimization and interval estimation formulations which seek to control or bound response variance, control or bounding of reliability requires sensitivities of tail statistics. In this work, the sensitivity of simple moment-based approximations to cumulative distribution function (CDF) and complementary cumulative distribution function (CCDF) mappings (Eqs. 1.3–1.4) are employed for this purpose, such that it is straightforward to form approximate design sensitivities of reliability index β (forward reliability mapping $\bar{z} \rightarrow \beta$) or response level z (inverse reliability mapping $\bar{\beta} \rightarrow z$) from the moment design sensitivities and the specified levels $\bar{\beta}$ or $\bar{z}$.

6.2.2 Optimization Formulations

Given the capability to compute analytic statistics of the response along with design sensitivities of these statistics, DAKOTA supports bi-level, sequential, and multifidelity approaches for optimization under uncertainty (OUU). The latter two approaches apply surrogate modeling approaches (data fits and multifidelity modeling) to the uncertainty analysis and then apply trust region model management to the optimization process.

6.2.2.1 Bi-level SEBDO

The simplest and most direct approach is to employ these analytic statistics and their design derivatives from Section 6.2.1 directly within an optimization loop. This approach is known as bi-level OUU, since there is an inner level uncertainty analysis nested within an outer level optimization.

Consider the common reliability-based design example of a deterministic objective function with a reliability constraint:

$$\begin{aligned} \text{minimize} \quad & f \\ \text{subject to} \quad & \beta \geq \bar{\beta} \end{aligned} \tag{6.30}$$

where β is computed relative to a prescribed threshold response value $\bar{z}$ (e.g., a failure threshold) and is constrained by a prescribed reliability level $\bar{\beta}$ (minimum allowable reliability in the design), and is either a CDF or CCDF index depending on the definition of the failure domain (i.e., defined from whether the associated failure probability is cumulative, $p(g \leq \bar{z})$, or complementary cumulative, $p(g > \bar{z})$).

Another common example is robust design in which the constraint enforcing a reliability lower-bound has been replaced with a constraint enforcing a variance upper-bound $\bar{\sigma}^2$ (maximum allowable variance in the design):

$$\begin{aligned} \text{minimize} \quad & f \\ \text{subject to} \quad & \sigma^2 \leq \bar{\sigma}^2 \end{aligned} \tag{6.31}$$

Solving these problems using a bi-level approach involves computing β and $\frac{d\beta}{d\boldsymbol{s}}$ for Eq. 6.30 or σ^2 and $\frac{d\sigma^2}{d\boldsymbol{s}}$ for Eq. 6.31 for each set of design variables $\boldsymbol{s}$ passed from the optimizer. This approach is supported for both probabilistic and combined expansions using PCE and SC.

6.2.2.2 Sequential/Surrogate-Based SEBDO

An alternative OUU approach is the sequential approach, in which additional efficiency is sought through breaking the nested relationship of the UQ and optimization loops. The general concept is to iterate between optimization and uncertainty quantification, updating the optimization goals based on the most recent uncertainty assessment results. This approach is common with the reliability methods community, for which the updating strategy may be based on safety factors [85] or other approximations [23].

A particularly effective approach for updating the optimization goals is to use data fit surrogate models, and in particular, local Taylor series models allow direct insertion of stochastic sensitivity analysis capabilities. In Ref. [25], first-order Taylor series approximations were explored, and in Ref. [26], second-order Taylor series approximations are investigated. In both cases, a trust-region model management framework [28] is used to adaptively manage the extent of the approximations and ensure convergence of the OUU process. Surrogate models are used for both the objective and the constraint functions, although the use of surrogates is only required for the functions containing statistical results; deterministic functions may remain explicit is desired.

In particular, trust-region surrogate-based optimization for reliability-based design employs surrogate models of f and β within a trust region Δ^k centered at $\mathbf{s}_c$:

$$\begin{array}{rc} \text{minimize} & f(\mathbf{s}_c) + \nabla_s f(\mathbf{s}_c)^T(\mathbf{s} - \mathbf{s}_c) \\ \text{subject to} & \beta(\mathbf{s}_c) + \nabla_s \beta(\mathbf{s}_c)^T(\mathbf{s} - \mathbf{s}_c) \geq \bar{\beta} \\ & \| \mathbf{s} - \mathbf{s}_c \|_\infty \leq \Delta^k \end{array} \tag{6.32}$$

and trust-region surrogate-based optimization for robust design employs surrogate models of f and σ^2 within a trust region Δ^k centered at $\mathbf{s}_c$:

$$\begin{array}{rc} \text{minimize} & f(\mathbf{s}_c) + \nabla_s f(\mathbf{s}_c)^T(\mathbf{s} - \mathbf{s}_c) \\ \text{subject to} & \sigma^2(\mathbf{s}_c) + \nabla_s \sigma^2(\mathbf{s}_c)^T(\mathbf{s} - \mathbf{s}_c) \leq \bar{\sigma}^2 \\ & \| \mathbf{s} - \mathbf{s}_c \|_\infty \leq \Delta^k \end{array} \tag{6.33}$$

Second-order local surrogates may also be employed, where the Hessians are typically approximated from an accumulation of curvature information using quasi-Newton updates [61] such as Broyden-Fletcher-Goldfarb-Shanno (BFGS, Eq. 1.41) or symmetric rank one (SR1, Eq. 1.42). The sequential approach is available for probabilistic expansions using PCE and SC.

6.2.2.3 Multifidelity SEBDO

The multifidelity OUU approach is another trust-region surrogate-based approach. Instead of the surrogate UQ model being a simple data fit (in particular, first-/second-order Taylor series model) of the truth UQ model results, distinct UQ models of differing fidelity are now employed. This differing UQ fidelity could stem from the fidelity of the underlying simulation model, the fidelity of the UQ algorithm, or both. In this section, we focus on the fidelity of the UQ algorithm. For reliability methods, this could entail varying fidelity in approximating assumptions (e.g., Mean Value for low fidelity, SORM for high fidelity), and for stochastic expansion methods, it could involve differences in selected levels of p and h refinement.

Here, we define UQ fidelity as point-wise accuracy in the design space and take the high fidelity truth model to be the probabilistic expansion PCE/SC model, with validity only at a single design point. The low fidelity model, whose validity over the design space will be adaptively controlled, will be either the combined expansion PCE/SC model, with validity over a range of design parameters, or the MVFOSM reliability method, with validity only at a single design point. The combined expansion low fidelity approach will span the current trust region

of the design space and will be reconstructed for each new trust region. Trust region adaptation will ensure that the combined expansion approach remains sufficiently accurate for design purposes. By taking advantage of the design space spanning, one can eliminate the cost of multiple low fidelity UQ analyses within the trust region, with fallback to the greater accuracy and higher expense of the probabilistic expansion approach when needed. The MVFOSM low fidelity approximation must be reformed for each change in design variables, but it only requires a single evaluation of a response function and its derivative to approximate the response mean and variance from the input mean and covariance (Eqs. 1.1–1.2) from which forward/inverse CDF/CCDF reliability mappings can be generated using Eqs. 1.3–1.4. This is the least expensive UQ option, but its limited accuracy[1] may dictate the use of small trust regions, resulting in greater iterations to convergence. The expense of optimizing a combined expansion, on the other hand, is not significantly less than that of optimizing the high fidelity UQ model, but its representation of global trends should allow the use of larger trust regions, resulting in reduced iterations to convergence. The design derivatives of each of the PCE/SC expansion models provide the necessary data to correct the low fidelity model to first-order consistency with the high fidelity model at the center of each trust region, ensuring convergence of the multifidelity optimization process to the high fidelity optimum. Design derivatives of the MVFOSM statistics are currently evaluated numerically using forward finite differences.

Multifidelity optimization for reliability-based design can be formulated as:

$$\begin{array}{rc} \text{minimize} & f(\mathbf{s}) \\ \text{subject to} & \hat{\beta_{hi}}(\mathbf{s}) \geq \bar{\beta} \\ & \| \mathbf{s} - \mathbf{s}_c \|_\infty \leq \Delta^k \end{array} \tag{6.34}$$

and multifidelity optimization for robust design can be formulated as:

$$\begin{array}{rc} \text{minimize} & f(\mathbf{s}) \\ \text{subject to} & \hat{\sigma_{hi}}^2(\mathbf{s}) \leq \bar{\sigma}^2 \\ & \| \mathbf{s} - \mathbf{s}_c \|_\infty \leq \Delta^k \end{array} \tag{6.35}$$

where the deterministic objective function is not approximated and $\hat{\beta_{hi}}$ and $\hat{\sigma_{hi}}^2$ are the approximated high-fidelity UQ results resulting from correction of the low-fidelity UQ results. In the case of an additive correction function:

$$\hat{\beta_{hi}}(\mathbf{s}) = \beta_{lo}(\mathbf{s}) + \alpha_\beta(\mathbf{s}) \tag{6.36}$$

$$\hat{\sigma_{hi}}^2(\mathbf{s}) = \sigma^2_{lo}(\mathbf{s}) + \alpha_{\sigma^2}(\mathbf{s}) \tag{6.37}$$

where correction functions $\alpha(\mathbf{s})$ enforcing first-order consistency [29] are typically employed. Quasi-second-order correction functions [29] can also be employed, but care must be taken due to the different rates of curvature accumulation between the low and high fidelity models. In particular, since the low fidelity model is evaluated more frequently than the high fidelity model, it accumulates curvature information more quickly, such that enforcing quasi-second-order consistency with the high fidelity model can be detrimental in the initial iterations of the algorithm[2]. Instead, this consistency should only be enforced when sufficient high fidelity curvature information has been accumulated (e.g., after n rank one updates).

[1] MVFOSM is exact for linear functions with Gaussian inputs, but quickly degrades for nonlinear and/or non-Gaussian.
[2] Analytic and numerical Hessians, when available, are instantaneous with no accumulation rate concerns.

Bibliography

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, 1965. 22, 29, 34

[2] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, M. S. Eldred, P. D. Hough, S. Lefantzi, L. P. Swiler, and D. M. Vigil. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.2 user's manual. Technical Report SAND2010-2183, Sandia National Laboratories, Albuquerque, NM, Updated Nov. 2011. Available online from http://www.cs.sandia.gov/dakota/documentation.html. 34, 38

[3] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, M. S. Eldred, P. D. Hough, S. Lefantzi, L. P. Swiler, and D. M. Vigil. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.2 reference manual. Technical Report SAND2010-2184, Sandia National Laboratories, Albuquerque, NM, Updated Nov. 2011. Available online from http://www.cs.sandia.gov/dakota/documentation.html. 35, 55

[4] N. M. Alexandrov, R. M. Lewis, C. R. Gumbert, L. L. Green, and P. A. Newman. Optimization with variable-fidelity models applied to wing design. In *Proceedings of the 38th Aerospace Sciences Meeting and Exhibit*, Reno, NV, 2000. AIAA Paper 2000-0841. 53, 54, 55

[5] M. Allen and K. Maute. Reliability-based design optimization of aeroelastic structures. *Struct. Multidiscip. O.*, 27:228–242, 2004. 62

[6] R. Askey and J. Wilson. Some basic hypergeometric polynomials that generalize jacobi polynomials. In *Mem. Amer. Math. Soc. 319*, Providence, RI, 1985. AMS. 21

[7] V. Barthelmann, E. Novak, and K. Ritter. High dimensional polynomial interpolation on sparse grids. *Adv. Comput. Math.*, 12(4):273–288, 2000. Multivariate polynomial interpolation. 30

[8] C. Berg, J. Mateu, and E. Porcu. The dagum family of isotropic correlation functions. *Bernoulli*, 14(4):1134–1149, 2008. 45

[9] B. J. Bichon, M. S. Eldred, L. P. Swiler, S. Mahadevan, and J. M. McFarland. Efficient global reliability analysis for nonlinear implicit performance functions. *AIAA Journal*, 46(10):2459–2468, 2008. 15

[10] G. E. P. Box and D. R. Cox. An analysis of transformations. *J. Royal Stat. Soc., Series B*, 26:211–252, 1964. 10, 28

[11] K. Breitung. Asymptotic approximation for multinormal integrals. *J. Eng. Mech., ASCE*, 110(3):357–366, 1984. 13

[12] J. Burkardt. The "combining coefficient" for anisotropic sparse grids. Technical report, Virginia Tech, Blacksburg, VA, 2009. http://people.sc.fsu.edu/~burkardt/presentations/sgmga_coefficient.pdf. 32

[13] J. Burkardt. Computing the hermite interpolation polynomial. Technical report, Florida State University, Gainesville, FL, 2009. http://people.sc.fsu.edu/~jburkardt/presentations/hermite_interpolant.pdf. 23

[14] M. R. Celis, J. .E. Dennis, and R. .A Tapia. A trust region strategy for nonlinear equality constrained optimization. In P. .T. Boggs, R. H. Byrd, and R. B. Schnabel, editors, *Numerical Optimization 1984*, pages 71–82. SIAM, Philadelphia, USA, 1985. 55

[15] X. Chen and N.C. Lind. Fast probability integration by three-parameter normal tail approximation. *Struct. Saf.*, 1:269–276, 1983. 10

[16] A. R. Conn, N. I. M. Gould, and P. L. Toint. *Trust-Region Methods*. MPS-SIAM Series on Optimization, SIAM-MPS, Philadelphia, 2000. 56

[17] P. G. Constantine and M. S. Eldred. Sparse polynomial chaos expansions. *International Journal for Uncertainty Quantification*, in preparation. 31

[18] P. G. Constantine, D. F. Gleich, and G. Iaccarino. Spectral methods for parameterized matrix equations. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2681–2699, 2010. 30

[19] N. Cressie. *Statistics of Spatial Data*. John Wiley and Sons, New York, 1991. 17, 43

[20] G.J. Davis and M.D. Morris. Six factors which affect the condition number of matrices associated with kriging. *Mathematical geology*, 29(5):669–683, 1997. 45

[21] A. Der Kiureghian and P. L. Liu. Structural reliability under incomplete information. *J. Eng. Mech., ASCE*, 112(EM-1):85–104, 1986. 10, 11, 22, 28

[22] A. Dey and S. Mahadevan. Ductile structural system reliability analysis using adaptive importance sampling. *Structural Safety*, 20:137–154, 1998. 16

[23] X. Du and W. Chen. Sequential optimization and reliability assessment method for efficient probabilistic design. *J. Mech. Design*, 126:225–233, 2004. 63, 67

[24] R.P. Dwight and Z. Han. Efficient uncertainty quantification using gradient-enhanced kriging. In *11th AIAA Non-Deterministic Approaches Conference*, Reston, VA, USA, May 2009. American Institute of Aeronautics and Astronautics. 50

[25] M. S. Eldred, H. Agarwal, V. M. Perez, Jr. Wojtkiewicz, S. F., and J. E. Renaud. Investigation of reliability method formulations in dakota/uq. *Structure & Infrastructure Engineering: Maintenance, Management, Life-Cycle Design & Performance*, 3(3):199–213, 2007. 11, 15, 61, 63, 67

[26] M. S. Eldred and B. J. Bichon. Second-order reliability formulations in DAKOTA/UQ. In *Proceedings of the 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, number AIAA-2006-1828, Newport, RI, May 1–4 2006. 11, 14, 61, 63, 67

[27] M. S. Eldred and J. Burkardt. Comparison of non-intrusive polynomial chaos and stochastic collocation methods for uncertainty quantification. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA-2009-0976, Orlando, FL, January 5–8, 2009. 30, 31

[28] M. S. Eldred and D. M. Dunlavy. Formulations for surrogate-based optimization with data fit, multifidelity, and reduced-order models. In *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, number AIAA-2006-7117, Portsmouth, VA, September 6–8 2006. 67

[29] M. S. Eldred, A. A. Giunta, and S. S. Collis. Second-order corrections for surrogate-based optimization with model hierarchies. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY,, Aug. 30–Sept. 1, 2004. AIAA Paper 2004-4457. 68

[30] M. S. Eldred, A. A. Giunta, S. F. Wojtkiewicz Jr., and T. G. Trucano. Formulations for surrogate-based optimization under uncertainty. In *Proc. 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-2002-5585, Atlanta, GA, September 4–6, 2002. 62

[31] M. S. Eldred, C. G. Webster, and P. Constantine. Evaluation of non-intrusive approaches for wiener-askey generalized polynomial chaos. In *Proceedings of the 10th AIAA Non-Deterministic Approaches Conference*, number AIAA-2008-1892, Schaumburg, IL, April 7–10 2008. 22

[32] M.S. Eldred, B.M. Adams, D.M. Gay, L.P. Swiler, K. Haskell, W.J. Bohnhoff, J.P. Eddy, W.E. Hart, J.P Watson, J.D. Griffin, P.D. Hough, T.G. Kolda, P.J. Williams, and M.L. Martinez-Canales. Dakota version 4.1 users manual. Sandia Technical Report SAND2006-6337, Sandia National Laboratories, Albuquerque, NM, 2007. 45

[33] G. M. Fadel, M. F. Riley, and J.-F. M. Barthelemy. Two point exponential approximation method for structural optimization. *Structural Optimization*, 2(2):117–124, 1990. 12

[34] R. Fletcher, S. Leyffer, and P. L. Toint. On the global convergence of a filter-SQP algorithm. *SIAM J. Optim.*, 13(1):44–59, 2002. 55, 56

[35] P. Frauenfelder, C. Schwab, and R. A. Todor. Finite elements for elliptic problems with stochastic coefficients. *Comput. Methods Appl. Mech. Engrg.*, 194(2-5):205–228, 2005. 30

[36] J. Gablonsky. Direct version 2.0 userguide technical report. Technical Report CRSC-TR01-08, North Carolina State University, Center for Research in Scientific Computation, Raleigh, NC, 2001. 18

[37] W. Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, New York, 2004. 22

[38] T. Gerstner and M. Griebel. Numerical integration using sparse grids. *Numer. Algorithms*, 18(3-4):209–232, 1998. 30

[39] T. Gerstner and M. Griebel. Dimension-adaptive tensor-product quadrature. *Computing*, 71(1):65–87, 2003. 36, 37

[40] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. User's guide for NPSOL (Version 4.0): A Fortran package for nonlinear programming. Technical Report TR SOL-86-2, System Optimization Laboratory, Stanford University, Stanford, CA, 1986. 15

[41] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, San Diego, CA, 1981. 54

[42] A. A. Giunta and M. S. Eldred. Implementation of a trust region model management strategy in the DAKOTA optimization toolkit. In *Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, number AIAA-2000-4935, Long Beach, CA, September 6–8, 2000. 63

[43] T. Gneiting, M.G. Genton, and P. Guttorp. Geostatistical space-time models, stationarity, separability and full symmetry. In B. Finkenstadt, L. Held, and V. Isham, editors, *Statistical Methods for Spatio-Temporal Systems*, chapter 4, pages 151–172. Boca Raton: Chapman & Hall/CRC, 2007. 45

[44] G. H.. Golub and J. H. Welsch. Caclulation of gauss quadrature rules. *Mathematics of Computation*, 23(106):221–230, 1969. 22

[45] A. Haldar and S. Mahadevan. *Probability, Reliability, and Statistical Methods in Engineering Design*. Wiley, New York, 2000. 9, 10, 15

[46] M. Hohenbichler and R. Rackwitz. Sensitivity and importance measures in structural reliability. *Civil Eng. Syst.*, 3:203–209, 1986. 62

[47] M. Hohenbichler and R. Rackwitz. Improvement of second-order reliability estimates by importance sampling. *J. Eng. Mech., ASCE*, 114(12):2195–2199, 1988. 13, 14

[48] H.P. Hong. Simple approximations for improving second-order reliability estimates. *J. Eng. Mech., ASCE*, 125(5):592–595, 1999. 13, 14

[49] S. Hosder, R. W. Walters, and M. Balch. Efficient sampling for non-intrusive polynomial chaos applications with multiple uncertain input variables. In *Proceedings of the 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number AIAA-2007-1939, Honolulu, HI, April 23–26, 2007. 32

[50] D. Huang, T. T. Allen, W. I. Notz, and N. Zeng. Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of Global Optimization*, 34:441–466, 2006. 16, 18

[51] D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998. 15, 16, 18

[52] A. Karamchandani and C. A. Cornell. Sensitivity estimation within first and second order reliability methods. *Struct. Saf.*, 11:95–107, 1992. 62

[53] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Prentice–Hall, 1974. 57

[54] C. Lucas. Lapack-style codes for level 2 and 3 pivoted cholesky factorizations. Numerical Analysis Reports 442, Manchester Centre for Computational Mathematics, Manchester, England, February 2004. LAPACK Working Note 161. 50

[55] J.D. Martin. Computational improvements to estimating kriging metamodel parameters. *Journal of Mechanical Design*, 131:084501, 2009. 45

[56] G. Matheron. *The theory of regionalized variables and its applications*. Les Cahiers du Centre de morphologie mathématique de Fontainebleau. École national supérieure des mines, 1971. 44

[57] J. C. Meza, R. A. Oliva, P. D. Hough, and P. J. Williams. OPT++: an object oriented toolkit for nonlinear optimization. *ACM Transactions on Mathematical Software*, 33(2), 2007. 15

[58] L. W. T. Ng and M. S. Eldred. Multifidelity uncertainty quantification using nonintrusive polynomial chaos and stochastic collocation. In *Proceedings of the 14th AIAA Non-Deterministic Approaches Conference*, Honolulu, HI, April 23-26, 2012. 37

[59] F. Nobile, R. Tempone, and C. G. Webster. A sparse grid stochastic collocation method for partial differential equations with random input data. Technical Report Technical report TRITA-NA 2007:7, Royal Institute of Technology, Stockholm, Sweden, 2007. 30

[60] F. Nobile, R. Tempone, and C. G. Webster. An anisotropic sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM J. on Num. Anal.*, 46(5):2411–2442, 2008. 30

[61] J. Nocedal and Wright S. J. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 1999. 54, 67

[62] E. O. Omojokun. *Trust Region Algorithms for Optimization with Nonlinear Equality and Inequality Constraints*. PhD thesis, University of Colorado, Boulder, Colorado, 1989. 55

[63] V. M. Pérez, M. S. Eldred, , and J. E. Renaud. Solving the infeasible trust-region problem using approximations. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, Aug. 30–Sept. 1, 2004. AIAA Paper 2004-4312. 55, 57

[64] V. M. Pérez, J. E. Renaud, and L. T. Watson. An interior-point sequetial approximation optimization methodology. *Structural and Multidisciplinary Optimization*, 27(5):360–370, July 2004. 54, 55, 57, 59

[65] R. Rackwitz. Optimization and risk acceptability based on the Life Quality Index. *Struct. Saf*, 24:297–331, 2002. 62

[66] R. Rackwitz and B. Fiessler. Structural reliability under combined random load sequences. *Comput. Struct.*, 9:489–494, 1978. 10

[67] P. Ranjan, Bingham D., and Michailidis G. Sequential experiment design for contour estimation from complex computer codes. *Technometrics*, 50(4):527–541, 2008. 19

[68] M. T. Reagan, H. N. Najm, P. P. Pebay, O. M. Knio, and R. G. Ghanem. Quantifying uncertainty in chemical systems modeling. *Int. J. Chem. Kinet.*, 37:368–382, 2005. 33

[69] G. Rennen. Subset selection from large datasets for kriging modeling. *Structural and Multidisciplinary Optimization*, 38(6):545–569, 2009. 45

[70] J. F. Rodriguez, J. E. Renaud, and L. T. Watson. Convergence of trust region augmented lagrangian methods using variable fidelity approximation data. *Structural Optimization*, 15:1–7, 1998. 53

[71] M. Rosenblatt. Remarks on a multivariate transformation. *Annals of Mathematical Statistics*, 23(3):470–472, 1952. 10, 28

[72] J. Sacks, S. B. Schiller, and W. Welch. Design for computer experiments. *Technometrics*, 31:41–47, 1989. 17

[73] I.C. Simpson. Numerical integration over a semi-infinite interval, using the lognormal distibution. *Numerische Mathematik*, 31:71–76, 1978. 22

[74] S.A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl. Akad. Nauk SSSR*, 4:240–243, 1963. 30

[75] A. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice Hall, 1971. 32

[76] G. Tang, G. Iaccarino, and M. S Eldred. Global sensitivity analysis for stochastic collocation expansion. In *Proceedings of the 51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (12th AIAA Non-Deterministic Approaches conference)*, Orlando, FL, April 12-15, 2010. AIAA Paper 2010-XXXX. 35

[77] M.A. Tatang. *Direct incorporation of uncertainty in chemical and environmental engineering systems*. PhD thesis, MIT, 1995. 32

[78] J. Tu, K. K. Choi, and Y. H. Park. A new study on reliability-based design optimization. *J. Mech. Design*, 121:557–564, 1999. 11

[79] A. van der Sluis. Condition numbers and equilibration of matrices. *Numerische Mathematik*, 14(1):14–23, 1969. 50

[80] G. N. Vanderplaats. *Numerical Optimization Techniques for Engineering Design: With Applications*. McGraw-Hill, New York, 1984. 54, 56

[81] R. W. Walters. Towards stochastic fluid mechanics via polynomial chaos. In *Proceedings of the 41st AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA-2003-0413, Reno, NV, January 6–9, 2003. 32

[82] G. W. Wasilkowski and H. Woźniakowski. Explicit cost bounds of algorithms for multivariate tensor product problems. *Journal of Complexity*, 11:1–56, 1995. 30

[83] N. Wiener. The homogoeneous chaos. *Amer. J. Math.*, 60:897–936, 1938. 21

[84] J. A. S. Witteveen and H. Bijl. Modeling arbitrary uncertainties using gram-schmidt polynomial chaos. In *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA-2006-0896, Reno, NV, January 9–12 2006. 22

[85] Y.-T. Wu, Y. Shin, R. Sues, and M. Cesare. Safety-factor based approach for probability-based design optimization. In *Proc. 42nd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number AIAA-2001-1522, Seattle, WA, April 16–19 2001. 63, 67

[86] Y.-T. Wu and P.H. Wirsching. A new algorithm for structural reliability estimation. *J. Eng. Mech., ASCE*, 113:1319–1336, 1987. 10

[87] B. A. Wujek and J. E. Renaud. New adaptive move-limit management strategy for approximate optimization, part 1. *AIAA Journal*, 36(10):1911–1921, 1998. 55

[88] B. A. Wujek and J. E. Renaud. New adaptive move-limit management strategy for approximate optimization, part 2. *AIAA Journal*, 36(10):1922–1934, 1998. 55

[89] D. Xiu. Numerical integration formulas of degree two. *Applied Numerical Mathematics*, 58:1515–1520, 2008. 32

[90] D. Xiu and J.S. Hesthaven. High-order collocation methods for differential equations with random inputs. *SIAM J. Sci. Comput.*, 27(3):1118–1139 (electronic), 2005. 30

[91] S. Xu and R. V. Grandhi. Effective two-point function approximation for design optimization. *AIAA J.*, 36(12):2269–2275, 1998. 12

[92] T. Zou, S. Mahadevan, and R. Rebba. Computational efficiency in reliability-based optimization. In *Proceedings of the 9th ASCE Specialty Conference on Probabilistic Mechanics and Structural Reliability*, Albuquerque, NM, July 26–28, 2004. 63

[93] T. Zou, Z. Mourelatos, S. Mahadevan, and P. Meernik. Reliability analysis of automotive body-door subsystem. *Rel. Eng. and Sys. Safety*, 78:315–324, 2002. 16

SAND2010-2185
Unlimited Release
December 2009
Updated November 30, 2011

DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis

Version 5.2 Developers Manual

Brian M. Adams, Keith R. Dalbey, Michael S. Eldred, Laura P. Swiler
Optimization and Uncertainty Quantification Department

William J. Bohnhoff
Radiation Transport Department

John P. Eddy
System Readiness and Sustainment Technologies Department

Dena M. Vigil
Multiphysics Simulation Technologies Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Patricia D. Hough, Sophia Lefantzi
Quantitative Modeling and Analysis Department

Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551

Abstract

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible and extensible interface between simulation codes and iterative analysis methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods; uncertainty quantification with sampling, reliability, and stochastic expansion methods; parameter estimation with nonlinear least squares methods; and sensitivity/variance analysis with design of experiments and parameter study methods. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for design and performance analysis of computational models on high performance computers.

This report serves as a developers manual for the DAKOTA software and describes the DAKOTA class hierarchies and their interrelationships. It derives directly from annotation of the source code and provides detailed class documentation, including all member functions and attributes.

Contents

1 DAKOTA Developers Manual **7**

1.1 Introduction 7

1.2 Overview of DAKOTA 7

1.3 Services 12

1.4 Development Practices and Guidance 12

1.5 Additional Resources 13

2 Coding Style Guidelines and Conventions **15**

3 Introduction **17**

4 C++/c Style Guidelines **19**

4.1 Class and variable styles 19

4.2 Function styles 19

4.3 Miscellaneous 20

5 File Naming Conventions **23**

6 Class Documentation Conventions **25**

7 CMake Style Guidelines **27**

7.1 CMake Code Formatting 27

7.2 CMake Variable Naming Conventions 27

8 Instructions for Modifying DAKOTA's Input Specification **29**

9 Modify dakota.input.nspec **31**

10 Rebuild generated files **33**

11 Update NIDRProblemDescDB.C in Dakota/src **35**

12 Update ProblemDescDB.C in Dakota/src **37**

12.1 Augment/update get_<data_type>() functions 37

13 Update Corresponding Data Classes **39**

13.1 Update the Data class header file 39

13.2 Update the .C file 39

14 Use get_<data_type>() Functions **41**

15 Update the Documentation **43**

16 Understanding Iterator Flow **45**

17 Interfacing with DAKOTA as a Library **47**

18 Introduction **49**

19 Quick start: examples and test code **51**

20 Comparison to main.C **53**

21 Problem database population **55**

21.1 Input file parsing 55

21.2 Data node insertion 55

21.3 Mixed mode 56

22 Instantiating the strategy **59**

23 Defining the direct application interface **61**

23.1 Extension 61

23.2 Derivation 61

24 Additional updates **65**

25 Executing the strategy **67**

26 Retrieving data after a run **69**

27 Linking against the DAKOTA library **71**

28 Summary 73

29 Performing Function Evaluations 75

30 Synchronous function evaluations 77

31 Asynchronous function evaluations 79

32 Analyses within each function evaluation 81

33 Working with Variable Containers and Views 83

34 Storage in Variables 85

35 Storage in SharedVariablesData 87

36 Active and inactive views 89

37 Todo List 91

38 Namespace Index 93

38.1 Namespace List 93

39 Class Index 95

39.1 Class Hierarchy 95

40 Class Index 99

40.1 Class List 99

41 File Index 105

41.1 File List 105

42 Namespace Documentation 107

42.1 Dakota Namespace Reference 107

42.2 SIM Namespace Reference 263

43 Class Documentation 265

43.1 ActiveSet Class Reference 265

43.2 AnalysisCode Class Reference 269

43.3 Analyzer Class Reference 274

43.4 ApplicationInterface Class Reference . . . 280
43.5 Approximation Class Reference . . . 294
43.6 ApproximationInterface Class Reference . . . 303
43.7 APPSEvalMgr Class Reference . . . 309
43.8 APPSOptimizer Class Reference . . . 312
43.9 BaseConstructor Struct Reference . . . 316
43.10 BiStream Class Reference . . . 317
43.11 BoStream Class Reference . . . 320
43.12 COLINApplication Class Reference . . . 323
43.13 COLINOptimizer Class Reference . . . 327
43.14 CollaborativeHybridStrategy Class Reference . . . 332
43.15 CommandLineHandler Class Reference . . . 334
43.16 CommandShell Class Reference . . . 336
43.17 ConcurrentStrategy Class Reference . . . 338
43.18 CONMINOptimizer Class Reference . . . 341
43.19 Constraints Class Reference . . . 349
43.20 DataFitSurrModel Class Reference . . . 362
43.21 DataInterface Class Reference . . . 375
43.22 DataMethod Class Reference . . . 377
43.23 DataMethodRep Class Reference . . . 379
43.24 DataModel Class Reference . . . 394
43.25 DataModelRep Class Reference . . . 396
43.26 DataResponses Class Reference . . . 401
43.27 DataResponsesRep Class Reference . . . 403
43.28 DataStrategy Class Reference . . . 408
43.29 DataStrategyRep Class Reference . . . 410
43.30 DataVariables Class Reference . . . 413
43.31 DataVariablesRep Class Reference . . . 415
43.32 DDACEDesignCompExp Class Reference . . . 426
43.33 DirectApplicInterface Class Reference . . . 431
43.34 DiscrepancyCorrection Class Reference . . . 441
43.35 DOTOptimizer Class Reference . . . 446
43.36 Driver Class Reference . . . 451
43.37 EffGlobalMinimizer Class Reference . . . 453

43.38EmbeddedHybridStrategy Class Reference 456
43.39Evaluator Class Reference 458
43.40EvaluatorCreator Class Reference 465
43.41ForkAnalysisCode Class Reference 467
43.42ForkApplicInterface Class Reference 469
43.43FSUDesignCompExp Class Reference 473
43.44GaussProcApproximation Class Reference 478
43.45GetLongOpt Class Reference 485
43.46Graphics Class Reference 489
43.47GridApplicInterface Class Reference 493
43.48HierarchSurrModel Class Reference 496
43.49HybridStrategy Class Reference 501
43.50Interface Class Reference 503
43.51Iterator Class Reference 513
43.52JEGAOptimizer Class Reference 527
43.53LeastSq Class Reference 536
43.54MergedConstraints Class Reference 541
43.55MergedVariables Class Reference 543
43.56Minimizer Class Reference 545
43.57MixedConstraints Class Reference 555
43.58MixedVariables Class Reference 557
43.59Model Class Reference 559
43.60MPIPackBuffer Class Reference 592
43.61MPIUnpackBuffer Class Reference 595
43.62NCSUOptimizer Class Reference 598
43.63NestedModel Class Reference 602
43.64NIDRProblemDescDB Class Reference 612
43.65NL2Res Struct Reference 617
43.66NL2SOLLeastSq Class Reference 618
43.67NLPQLPOptimizer Class Reference 621
43.68NLSSOLLeastSq Class Reference 627
43.69NoDBBaseConstructor Struct Reference 629
43.70NonD Class Reference 630
43.71NonDAdaptImpSampling Class Reference 640

43.72 NonDBayesCalibration Class Reference . . . 644
43.73 NonDCalibration Class Reference . . . 646
43.74 NonDCubature Class Reference . . . 649
43.75 NonDExpansion Class Reference . . . 653
43.76 NonDGlobalEvidence Class Reference . . . 660
43.77 NonDGlobalInterval Class Reference . . . 662
43.78 NonDGlobalReliability Class Reference . . . 666
43.79 NonDGlobalSingleInterval Class Reference . . . 669
43.80 NonDGPMSABayesCalibration Class Reference . . . 671
43.81 NonDIncremLHSSampling Class Reference . . . 674
43.82 NonDIntegration Class Reference . . . 677
43.83 NonDInterval Class Reference . . . 680
43.84 NonDLHSEvidence Class Reference . . . 683
43.85 NonDLHSInterval Class Reference . . . 685
43.86 NonDLHSSampling Class Reference . . . 687
43.87 NonDLHSSingleInterval Class Reference . . . 690
43.88 NonDLocalEvidence Class Reference . . . 692
43.89 NonDLocalInterval Class Reference . . . 694
43.90 NonDLocalReliability Class Reference . . . 697
43.91 NonDLocalSingleInterval Class Reference . . . 706
43.92 NonDPolynomialChaos Class Reference . . . 708
43.93 NonDQuadrature Class Reference . . . 711
43.94 NonDQUESOBayesCalibration Class Reference . . . 716
43.95 NonDReliability Class Reference . . . 719
43.96 NonDSampling Class Reference . . . 724
43.97 NonDSparseGrid Class Reference . . . 731
43.98 NonDStochCollocation Class Reference . . . 735
43.99 NPSOLOptimizer Class Reference . . . 737
43.100 Optimizer Class Reference . . . 741
43.101 ParallelConfiguration Class Reference . . . 747
43.102 ParallelDirectApplicInterface Class Reference . . . 749
43.103 ParallelLevel Class Reference . . . 750
43.104 ParallelLibrary Class Reference . . . 754
43.105 ParamResponsePair Class Reference . . . 768

43.106 ParamStudy Class Reference 772
43.107 partial_prp_equality Struct Reference 777
43.108 partial_prp_hash Struct Reference 778
43.109 PecosApproximation Class Reference 779
43.110 ProblemDescDB Class Reference 786
43.111 PStudyDACE Class Reference 797
43.112 PSUADEDesignCompExp Class Reference 800
43.113 RecastBaseConstructor Struct Reference 804
43.114 RecastModel Class Reference 805
43.115 Response Class Reference 814
43.116 ResponseRep Class Reference 820
43.117 RichExtrapVerification Class Reference 828
43.118 SensAnalysisGlobal Class Reference 831
43.119 SequentialHybridStrategy Class Reference 833
43.120 SerialDirectApplicInterface Class Reference 838
43.121 SharedVariablesData Class Reference 839
43.122 SharedVariablesDataRep Class Reference 842
43.123 SingleMethodStrategy Class Reference 845
43.124 SingleModel Class Reference 847
43.125 SNLLBase Class Reference 850
43.126 SNLLLeastSq Class Reference 853
43.127 SNLLOptimizer Class Reference 858
43.128 SOLBase Class Reference 866
43.129 Strategy Class Reference 869
43.130 String Class Reference 878
43.131 SurfpackApproximation Class Reference 881
43.132 SurrBasedGlobalMinimizer Class Reference 885
43.133 SurrBasedLocalMinimizer Class Reference 887
43.134 SurrBasedMinimizer Class Reference 896
43.135 SurrogateModel Class Reference 902
43.136 SysCallAnalysisCode Class Reference 907
43.137 SysCallApplicInterface Class Reference 909
43.138 TANA3Approximation Class Reference 912
43.139 TaylorApproximation Class Reference 915

43.140 TrackerHTTP Class Reference . . . 917
43.141 Variables Class Reference . . . 920
43.142 Verification Class Reference . . . 933

44 File Documentation **935**
44.1 dll_api.C File Reference . . . 935
44.2 dll_api.h File Reference . . . 937
44.3 JEGAOptimizer.C File Reference . . . 939
44.4 JEGAOptimizer.H File Reference . . . 940
44.5 library_mode.C File Reference . . . 941
44.6 library_split.C File Reference . . . 943
44.7 main.C File Reference . . . 944
44.8 restart_util.C File Reference . . . 945

Chapter 1

DAKOTA Developers Manual

Author:

Brian M. Adams, William J. Bohnhoff, Keith R. Dalbey, John P. Eddy, Michael S. Eldred, Patricia D. Hough, Sophia Lefantzi, Laura P. Swiler, Dena M. Vigil

1.1 Introduction

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible, extensible interface between analysis codes and iteration methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods, uncertainty quantification with sampling, reliability, stochastic expansion, and interval estimation methods, parameter estimation with nonlinear least squares methods, and sensitivity/variance analysis with design of experiments and parameter study capabilities. (Solution verification and Bayesian approaches are also in development.) These capabilities may be used on their own or as components within advanced algorithms such as surrogate-based optimization, mixed integer nonlinear programming, mixed aleatory-epistemic uncertainty quantification, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible problem-solving environment for design and performance analysis of computational models on high performance computers.

The Developers Manual focuses on documentation of DAKOTA design principles and class structures; it derives principally from annotated source code. For information on input command syntax, refer to the `Reference Manual`, and for more details on DAKOTA features and capabilities, refer to the Users Manual.

1.2 Overview of DAKOTA

In DAKOTA, the *strategy* creates and manages *iterators* and *models*. In the simplest case, the strategy creates a single iterator and a single model and executes the iterator on the model to perform a single study. In a more advanced case, a hybrid optimization strategy might manage a global optimizer operating on a low-fidelity model in coordination with a local optimizer operating on a high-fidelity model. And on the high end, a surrogate-based optimization under uncertainty strategy would employ an uncertainty quantification iterator nested within an optimization iterator and would employ truth models layered within surrogate models. Thus, iterators and

models provide both stand-alone capabilities as well as building blocks for more sophisticated studies.

A model contains a set of *variables*, an *interface*, and a set of *responses*, and the iterator operates on the model to map the variables into responses using the interface. Each of these components is a flexible abstraction with a variety of specializations for supporting different types of iterative studies. In a DAKOTA input file, the user specifies these components through strategy, method, model, variables, interface, and responses keyword specifications.

The use of class hierarchies provides a mechanism for extensibility in DAKOTA components. In each of the various class hierarchies, adding a new capability typically involves deriving a new class and providing a set of virtual function redefinitions. These redefinitions define the coding portions specific to the new derived class, with the common portions already defined at the base class. Thus, with a small amount of new code, the existing facilities can be extended, reused, and leveraged for new purposes. The following sections tour DAKOTA's class organization.

1.2.1 Strategies

Class hierarchy: Strategy.

Strategies provide a control layer for creation and management of iterators and models. Specific strategies include:

- SingleMethodStrategy: the simplest strategy. A single iterator is run on a single model to perform a single study.
- HybridStrategy: hybrid minimization using a set of iterators employing a corresponding set of models of varying fidelity. Coordination approaches among the iterators include collaborative, embedded, and sequential approaches, as embodied in the CollaborativeHybridStrategy, EmbeddedHybridStrategy, and SequentialHybridStrategy derived classes.
- ConcurrentStrategy: two similar algorithms are available: (1) multi-start iteration from several different starting points, and (2) pareto set optimization for several different multiobjective weightings. Employs a single iterator with a single model, but runs multiple instances of the iterator concurrently for different settings within the model.

1.2.2 Iterators

Class hierarchy: Iterator. Iterator implementations may choose to split operations up into run-time phases as described in Understanding Iterator Flow.

The iterator hierarchy contains a variety of iterative algorithms for optimization, uncertainty quantification, nonlinear least squares, design of experiments, and parameter studies. The hierarchy is divided into Minimizer and Analyzer branches. The Minimizer classes address optimization and deterministic calibration and are grouped into:

- Optimization: Optimizer provides a base class for the DOTOptimizer, CONMINOptimizer, NPSOLOptimizer, NLPQLPOptimizer, and SNLLOptimizer gradient-based optimization libraries and the APPSOptimizer, COLINOptimizer, JEGAOptimizer, and NCSUOptimizer nongradient-based optimization methods and libraries.
- Parameter estimation: LeastSq provides a base class for NL2SOLLeastSq, a least-squares solver based on NL2SOL, SNLLLeastSq, a Gauss-Newton least-squares solver, and NLSSOLLeastSq, an SQP-based least-squares solver.

- Surrogate-based minimization (both optimization and nonlinear least squares): SurrBasedMinimizer provides a base class for SurrBasedLocalMinimizer, SurrBasedGlobalMinimizer, and EffGlobalMinimizer. The surrogate-based local and global methods employ a single iterator with any of the available SurrogateModel capabilities (local, multipoint, or global data fits or hierarchical approximations) and perform a sequence of approximate optimizations, each involving build, optimize, and verify steps. The efficient global method, on the other hand, hard-wires a recursion involving Gaussian process surrogate models coupled with the DIRECT global optimizer to maximize an expected improvement function.

The Analyzer classes are grouped into:

- Uncertainty quantification: NonD provides a base class for non-deterministic methods NonDSampling, NonDReliability (reliability analysis), NonDExpansion (stochastic expansion methods), and NonDInterval (interval-based epistemic methods). Bayesian calibration methods are prototyped in NonDBayesCalibration.
  - NonDSampling is further specialized with the NonDLHSSampling class for Latin hypercube and Monte Carlo sampling, the NonDIncremLHSSampling class for incremental Latin hypercube sampling, and NonDAdaptImpSampling for multimodal adaptive importance sampling.
  - NonDReliability is further specialized with local and global methods (NonDLocalReliability and NonDGlobalReliability).
  - NonDExpansion includes specializations for generalized polynomial chaos (NonDPolynomialChaos) and stochastic collocation (NonDStochCollocation) and is supported by the NonDIntegration helper class, which supplies cubature, tensor-product quadrature and Smolyak sparse grid methods (NonDCubature, NonDQuadrature, and NonDSparseGrid).
  - NonDInterval provides a base class for epistemic interval-based UQ methods. Three interval analysis approaches are provided: LHS (NonDLHSInterval), efficient global optimization (NonDGlobalInterval), and local optimization (NonDLocalInterval). Each of these three has specializations for single interval (NonDLHSSingleInterval, NonDGlobalSingleInterval, NonDLocalSingleInterval) and Dempster-Shafer Theory of Evidence (NonDLHSEvidence, NonDGlobalEvidence, NonDLocalEvidence) approaches.
- Parameter studies and design of experiments: PStudyDACE provides a base class for ParamStudy, which provides capabilities for directed parameter space interrogation, PSUADEDesignCompExp, which provides access to the Morris One-At-a-Time (MOAT) method for parameter screening, and DDACEDesignCompExp and FSUDesignCompExp, which provide for parameter space exploration through design and analysis of computer experiments. NonDLHSSampling from the uncertainty quantification branch also supports design of experiments when in `all_variables` mode.
- Solution verification studies: Verification provides a base class for RichExtrapVerification (verification via Richardson extrapolation) and other solution verification methods in development.

1.2.3 Models

Class hierarchy: Model.

The model classes are responsible for mapping variables into responses when an iterator makes a function evaluation request. There are several types of models, some supporting sub-iterators and sub-models for enabling layered and nested relationships. When sub-models are used, they may be of arbitrary type so that a variety of recursions are supported.

- SingleModel: variables are mapped into responses using a single Interface object. No sub-iterators or sub-models are used.

- SurrogateModel: variables are mapped into responses using an approximation. The approximation is built and/or corrected using data from a sub-model (the truth model) and the data may be obtained using a sub-iterator (a design of experiments iterator). SurrogateModel has two derived classes: DataFitSurrModel for data fit surrogates and HierarchSurrModel for hierarchical models of varying fidelity. The relationship of the sub-iterators and sub-models is considered to be "layered" since they are not used as part of every response evaluation on the top level model, but rather used periodically in surrogate update and verification steps.

- NestedModel: variables are mapped into responses using a combination of an optional Interface and a sub-iterator/sub-model pair. The relationship of the sub-iterators and sub-models is considered to be "nested" since they are used to perform a complete iterative study as part of every response evaluation on the top level model.

- RecastModel: recasts the inputs and outputs of a sub-model for the purposes of variable transformations (e.g., variable scaling, transformations to standardized random variables) and problem reformulation (e.g., multiobjective optimization, response scaling, augmented Lagrangian merit functions, expected improvement).

1.2.4 Variables

Class hierarchy: Variables.

The Variables class hierarchy manages design, aleatory uncertain, epistemic uncertain, and state *variable types* for continuous, discrete integer, and discrete real *domain types*. This hierarchy is specialized according to how the domain types are managed:

- MixedVariables: domain type distinctions are retained, such that separate continuous, discrete integer, and discrete real domain types are managed. This is the default Variable perspective, and draws its name from "mixed continuous-discrete" optimization.

- MergedVariables: domain types are combined through relaxation of discrete constraints; i.e., continuous and discrete variables are merged into continuous arrays through relaxation of integrality (for discrete integer ranges) or set membership (for discrete integer or discrete real sets) requirements. The branch and bound minimizer is the only method using this approach at present.

Whereas domain types are controlled through the derived class selection, selection of active variable types are handled within each of these derived classes using variable views. These permit different algorithms to work on different subsets of variables. For details, see Working with Variable Containers and Views.

The Constraints hierarchy manages bound, linear, and nonlinear constraints and utilizes the same specializations for managing bounds on the variables (see MixedConstraints and MergedConstraints).

1.2.5 Interfaces

Class hierarchy: Interface.

Interfaces provide access to simulation codes or, conversely, approximations based on simulation code data. In the simulation case, an ApplicationInterface is used. ApplicationInterface is specialized according to the simulation invocation mechanism, for which the following nonintrusive approaches

- SysCallApplicInterface: the simulation is invoked using a system call (the C function `system()`). Asynchronous invocation utilizes a background system call. Utilizes the SysCallAnalysisCode class to define syntax for input filter, analysis code, output filter, or combined spawning, which in turn utilize the CommandShell utility.

- ForkApplicInterface: the simulation is invoked using a fork (the `fork/exec/wait` family of functions). Asynchronous invocation utilizes a nonblocking fork. Utilizes the ForkAnalysisCode class for lower level fork operations.

and the following semi-intrusive approach

- DirectApplicInterface: the simulation is linked into the DAKOTA executable and is invoked using a procedure call. Asynchronous invocations will utilize nonblocking threads (capability not yet available).

are supported. Scheduling of jobs for asynchronous local, message passing, and hybrid parallelism approaches is performed in the ApplicationInterface class, with job initiation and job capture specifics implemented in the derived classes.

In the approximation case, global, multipoint, or local data fit approximations to simulation code response data can be built and used as surrogates for the actual, expensive simulation. The interface class providing this capability is

- ApproximationInterface: builds an approximation using data from a truth model and then employs the approximation for mapping variables to responses. This class contains an array of Approximation objects, one per response function, which support a variety of approximation types using the different Approximation derived classes. These include SurfpackApproximation (provides kriging, MARS, moving least squares, neural network, polynomial regression, and radial basis functions), GaussProcApproximation (Gaussian process models), PecosApproximation (multivariate orthogonal and Lagrange interpolation polynomials from Pecos), TANA3Approximation (two-point adaptive nonlinearity approximation), and TaylorApproximation (local Taylor series).

which is an essential component within the DataFitSurrModel capability described above in Models.

1.2.6 Responses

Class: Response.

The Response class provides an abstract data representation of response functions and their first and second derivatives (gradient vectors and Hessian matrices). These response functions can be interpreted as objective functions and constraints (optimization data set), residual functions and constraints (least squares data set), or generic response functions (uncertainty quantification data set). This class is not currently part of a class hierarchy, since the abstraction has been sufficiently general and has not required specialization.

1.3 Services

A variety of services are provided in DAKOTA for parallel computing, failure capturing, restart, graphics, etc. An overview of the classes and member functions involved in performing these services is included below.

- Multilevel parallel computing: DAKOTA supports multiple levels of nested parallelism. A strategy can manage concurrent iterators, each of which manages concurrent function evaluations, each of which manages concurrent analyses executing on multiple processors. Partitioning of these levels with MPI communicators is managed in ParallelLibrary and scheduling routines for the levels are part of Strategy, ApplicationInterface, and ForkApplicInterface.

- Parsing: DAKOTA employs the NIDR parser (New Input Deck Reader) to retrieve information from user input files. Parsing options are processed in CommandLineHandler and parsing occurs in ProblemDescDB::manage_inputs() called from main.C. NIDR uses the keyword handlers in the NIDRProblemDescDB derived class to populate data within the ProblemDescDB base class, which maintains a DataStrategy specification and lists of DataMethod, DataModel, DataVariables, DataInterface, and DataResponses specifications. Procedures for modifying the parsing subsystem are described in Instructions for Modifying DAKOTA's Input Specification.

- Failure capturing: Simulation failures can be trapped and managed using exception handling in ApplicationInterface and its derived classes.

- Restart: DAKOTA maintains a record of all function evaluations both in memory (for capturing any duplication) and on the file system (for restarting runs). Restart options are processed in CommandLineHandler and retrieved in ParallelLibrary::specify_outputs_restart(), restart file management occurs in ParallelLibrary::manage_outputs_restart(), and restart file insertions occur in ApplicationInterface. The `dakota_restart_util` executable, built from restart_util.C, provides a variety of services for interrogating, converting, repairing, concatenating, and post-processing restart files.

- Memory management: DAKOTA employs the techniques of reference counting and representation sharing through the use of letter-envelope and handle-body idioms (Coplien, "Advanced C++"). The former idiom provides for memory efficiency and enhanced polymorphism in the following class hierarchies: Strategy, Iterator, Model, Variables, Constraints, Interface, ProblemDescDB, and Approximation. The latter idiom provides for memory efficiency in data-intensive classes which do not involve a class hierarchy. The Response and parser data (DataStrategy, DataMethod, DataModel, DataVariables, DataInterface, and DataResponses) classes use this idiom. When managing reference-counted data containers (e.g., Variables or Response objects), it is important to properly manage shallow and deep copies, to allow for both efficiency and data independence as needed in a particular context.

- Graphics: DAKOTA provides 2D iteration history graphics using Motif widgets. Graphics data can also be cataloged in a tabular data file for post-processing with 3rd party tools such as Matlab, Tecplot, etc. These capabilities are encapsulated within the Graphics class.

1.4 Development Practices and Guidance

The following links provide guidance for core software components or specific development activities:

- Coding Style Guidelines and Conventions - coding practices used by the DAKOTA development team.

- Instructions for Modifying DAKOTA's Input Specification - how to interact with NIDR and the associated DAKOTA classes.
- Interfacing with DAKOTA as a Library - embed DAKOTA as a service within your application.
- Understanding Iterator Flow - explanation of the full granularity of steps in Iterator execution.
- Performing Function Evaluations - an overview of the classes and member functions involved in performing function evaluations synchronously or asynchronously.
- Working with Variable Containers and Views - discussion of data storage for variables and explanation of active and inactive views of this data.

1.5 Additional Resources

Additional development resources include:

- The DAKOTA Developer Portal linked from `http://dakota.sandia.gov/developer/` includes information on getting started as a developer and links to project management resources.
- Project web pages are maintained at `http://dakota.sandia.gov/` including links to frequently asked questions, documentation, publications, mailing lists, and other resources.

Chapter 2

Coding Style Guidelines and Conventions

Chapter 3

Introduction

Common code development practices can be extremely useful in multiple developer environments. Particular styles for code components lead to improved readability of the code and can provide important visual cues to other developers. Much of this recommended practices document is borrowed from the CUBIT mesh generation project, which in turn borrows its recommended practices from other projects, yielding some consistency across Sandia projects. While not strict requirements, these guidelines suggest a best-practices starting point for coding in DAKOTA.

Chapter 4

C++/c Style Guidelines

Style guidelines involve the ability to discern at a glance the type and scope of a variable or function.

4.1 Class and variable styles

Class names should be composed of two or more descriptive words, with the first character of each word capitalized, e.g.:

```
class ClassName;
```

Class member variables should be composed of two or more descriptive words, with the first character of the second and succeeding words capitalized, e.g.:

```
double classMemberVariable;
```

Temporary (i.e. local) variables are lower case, with underscores separating words in a multiple word temporary variable, e.g.:

```
int temporary_variable;
```

Constants (i.e. parameters) and enumeration values are upper case, with underscores separating words, e.g.:

```
const double CONSTANT_VALUE;
```

4.2 Function styles

Function names are lower case, with underscores separating words, e.g.:

```
int function_name();
```

There is no need to distinguish between member and non-member functions by style, as this distinction is usually clear by context. This style convention allows member function names which set and return the value of a similarly-named private member variable, e.g.:

```
int memberVariable;
void member_variable(int a) { // set
  memberVariable = a;
}
int member_variable() const { // get
  return memberVariable;
}
```

In cases where the data to be set or returned is more than a few bytes, it is highly desirable to employ const references to avoid unnecessary copying, e.g.:

```
void continuous_variables(const RealVector& c_vars) { // set
  continuousVariables = c_vars;
}
const RealVector& continuous_variables() const {      // get
  return continuousVariables;
}
```

Note that it is not necessary to always accept the returned data as a const reference. If it is desired to be able change this data, then accepting the result as a new variable will generate a copy, e.g.:

```
// reference to continuousVariables cannot be changed
const RealVector& c_vars = model.continuous_variables();
// local copy of continuousVariables can be changed
RealVector c_vars = model.continuous_variables();
```

4.3 Miscellaneous

Appearance of typedefs to redefine or alias basic types is isolated to a few header files (`data_types.h`, `template_defs.h`), so that issues like program precision can be changed by changing a few lines of typedefs rather than many lines of code, e.g.:

```
typedef double Real;
```

`xemacs` is the preferred source code editor, as it has C++ modes for enhancing readability through color (turn on "Syntax highlighting"). Other helpful features include "Paren highlighting" for matching parentheses and the "New Frame" utility to have more than one window operating on the same set of files (note that this is still the same edit session, so all windows are synchronized with each other). Window width should be set to 80 internal columns, which can be accomplished by manual resizing, or preferably, using the following alias in your shell resource file (e.g., .cshrc):

```
alias xemacs "xemacs -g 81x63"
```

where an external width of 81 gives 80 columns internal to the window and the desired height of the window will vary depending on monitor size. This window width imposes a coding standard since you should avoid line wrapping by continuing anything over 80 columns onto the next line.

Indenting increments are 2 spaces per indent and comments are aligned with the code they describe, e.g.:

```
void abort_handler(int code)
{
  int initialized = 0;
  MPI_Initialized(&initialized);
  if (initialized) {
    // comment aligned to block it describes
    int size;
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size>1)
      MPI_Abort(MPI_COMM_WORLD, code);
    else
      exit(code);
  }
  else
    exit(code);
}
```

Also, the continuation of a long command is indented 2 spaces, e.g.:

```
  const String& iterator_scheduling
    = problem_db.get_string("strategy.iterator_scheduling");
```

and similar lines are aligned for readability, e.g.:

```
  cout << "Numerical gradients using " << finiteDiffStepSize*100. << "%"
       << finiteDiffType << " differences\nto be calculated by the "
       << methodSource << " finite difference routine." << endl;
```

Lastly, #ifdef's are not indented (to make use of syntax highlighting in xemacs).

Chapter 5

File Naming Conventions

In addition to the style outlined above, the following file naming conventions have been established for the DAKOTA project.

File names for C++ classes should, in general, use the same name as the class defined by the file. Exceptions include:

- with the introduction of the Dakota namespace, base classes which previously utilized prepended Dakota identifiers can now safely omit the identifiers. However, since file names do not have namespace protection from name collisions, they retain the prepended Dakota identifier. For example, a class previously named DakotaModel which resided in DakotaModel.[CH], is now Dakota::Model (class Model in namespace Dakota) residing in the same filenames. The retention of the previous filenames reduces the possibility of multiple instances of a Model.H causing problems. Derived classes (e.g., NestedModel) do not require a prepended Dakota identifier for either the class or file names.
- in a few cases, it is convenient to maintain several closely related classes in a single file, in which case the file name may reflect the top level class or some generalization of the set of classes (e.g., DakotaResponse.[CH] files contain Dakota::Response and Dakota::ResponseRep classes, and DakotaBinStream.[CH] files contain the Dakota::BiStream and Dakota::BoStream classes).

The type of file is determined by one of the four file name extensions listed below:

- **.H** A class header file ends in the suffix .H. The header file provides the class declaration. This file does not contain code for implementing the methods, except for the case of inline functions. Inline functions are to be placed at the bottom of the file with the keyword inline preceding the function name.
- **.C** A class implementation file ends in the suffix .C. An implementation file contains the definitions of the members of the class.
- **.h** A header file ends in the suffix .h. The header file contains information usually associated with procedures. Defined constants, data structures and function prototypes are typical elements of this file.
- **.c** A procedure file ends in the suffix .c. The procedure file contains the actual procedures.

Chapter 6

Class Documentation Conventions

Class documentation uses the doxygen tool available from http://www.doxygen.org and employs the JAVA-doc comment style. Brief comments appear in header files next to the attribute or function declaration. Detailed descriptions for functions should appear alongside their implementations (i.e., in the .C files for non-inlined, or in the headers next to the function definition for inlined). Detailed comments for a class or a class attribute must go in the header file as this is the only option.

NOTE: Previous class documentation utilities (class2frame and class2html) used the "//-" comment style and comment blocks such as this:

```
//- Class:       Model
//- Description: The model to be iterated by the Iterator.
//-              Contains Variables, Interface, and Response objects.
//- Owner:       Mike Eldred
//- Version: $Id: Dev_Recomm_Pract.dox 4549 2007-09-20 18:25:03Z mseldre $
```

These tools are no longer used, so remaining comment blocks of this type are informational only and will not appear in the documentation generated by doxygen.

Chapter 7

CMake Style Guidelines

DAKOTA conventions for CMake files, such as CMakeLists.txt, FooConfig.cmake, etc., follow. Our goal is ease of reading, maintenance, and support, similar to the C++ code itself. Current CMake versions and build hints are maintained at the Developer Portal http://dakota.sandia.gov/developer/.

7.1 CMake Code Formatting

- Indentation is 2 spaces, consistent with DAKOTA C++ style.
- Lines should be kept to less than 80 chars per line where possible.
- Wrapped lines may be indented two spaces or aligned with prior lines.
- For ease of viewing and correctness checking in Emacs, a customization file is available: http://www.cmake.org/CMakeDocs/cmake-mode.el

7.2 CMake Variable Naming Conventions

These variable naming conventions are especially important for those that ultimately become preprocessor defines and affect compilation of source files.

- Classic/core elements of the CMake language are set in lower_case, e.g., option, set, if, find_library.
- Static arguments to CMake functions and macros are set in UPPER_CASE, e.g. REQUIRED, NO_-MODULE, QUIET.
- Minimize "global" variables, i.e., don't use 2 variables with the same meaning when one will do the job.
- Feature toggling: when possible, use the "HAVE_<pkg/feature>" convention already in use by many CMake-enabled TPLs, e.g.,

```
$ grep HAVE_SYSTEM Dakota/src/CMakeLists.txt

check_function_exists(system HAVE_SYSTEM)
```

```
if(HAVE_SYSTEM)
  add_definitions("-DHAVE_SYSTEM")
endif(HAVE_SYSTEM)

$ grep HAVE_CONMIN Dakota/src/CMakeLists.txt Dakota/packages/CMakeLists.txt
Dakota/src/CMakeLists.txt:if(HAVE_CONMIN)
Dakota/src/CMakeLists.txt:endif(HAVE_CONMIN)
Dakota/packages/CMakeLists.txt:option(HAVE_CONMIN "Build the CONMIN package." ON)

Dakota/packages/CMakeLists.txt:if(HAVE_CONMIN)
Dakota/packages/CMakeLists.txt:endif(HAVE_CONMIN)
```

- When a variable/preprocessor macro could result in name clashes beyond DAKOTA scope, e.g., for library_mode users, consider prefixing the "HAVE_<pkg>" name with DAKOTA_, e.g. DAKOTA_HAVE_MPI. Currently, MPI is the only use case for such a variable in DAKOTA, but many examples can be found in the CMake Modules source, e.g.

```
grep _HAVE_ <cmake_prefix_dir>/share/cmake-2.8/Modules/*
```

Chapter 8

Instructions for Modifying DAKOTA's Input Specification

To modify DAKOTA's input specification (for maintenance or addition of new input syntax), specification maintenance mode must be enabled at DAKOTA configure time with the `--enable-spec-maint` option, e.g.,

```
./configure --enable-spec-maint
```

This will enable regeneration of NIDR and DAKOTA components which must be updated following a spec change.

Chapter 9

Modify dakota.input.nspec

The master input specification dakota.input.nspec in Dakota/src is the primary file to update when making a specification change. It uses the following syntactic elements:

- () for required group specifications
- [] for optional specifications
- | for alternatives
- {} for functions to process keywords to express logical relationships. These syntactic elements can be used to express various dependency relationships in the input specification. It is recommended that you review the existing specification and have an understanding of the constructs in use before attempting to add new ones.

Warning:

- Do *not* skip this step. Attempts to modify the NIDR_keywds.H file in Dakota/src without using the NIDR table generator are very error-prone. Moreover, the input specification provides a reference to the allowable inputs of a particular executable and should be kept in synch with the parser files; modifying the parser files independent of the input specification creates, at a minimum, undocumented features.
- All keywords in dakota.input.nspec are lower case by convention. All user inputs are converted to lower case by the parser prior to keyword match testing, resulting in case insensitive parsing.
- Since the NIDR parser allows abbreviation of keywords, you *must* avoid adding a keyword that could be misinterpreted as an abbreviation for a different keyword within the same top-level keyword, such as "strategy" and "method". For example, adding the keyword "expansion" within the method specification would be a mistake if the keyword "expansion_factor" already was being used in this specification.
- The NIDR input is somewhat order-dependent, allowing the same keyword to be reused multiple times in the specification. This often happens with aliases, such as `lower_bounds`, `upper_bounds` and `initial_point`. Ambiguities are resolved by attaching a keyword to the most recently seen context in which it could appear, if such exists, or to the first relevant context that subsequently comes along in the input file. With the earlier IDR parser, non-exclusive specifications (those not in mutually exclusive blocks) were required to be unique. That is why there are such aliases for `initial_point` as `cdv_initial_point` and `ddv_initial_point`: so older input files can be used with no or fewer changes.

Chapter 10

Rebuild generated files

As of DAKOTA 5.1, a separate make command in `packages/nidr` is no longer necessary. When configured with `--enable-spec-maint`, performing a make in Dakota/src will regenerate all files which derive from dakota.input.nspec, including NIDR_keywds.H, **dakota.input.summary**, NIDR_guikywds.H, and dakota.input.desc. If you commit changes to a source repository, be sure to commit any automatically generated files in addition to any modified in the following steps. It is not strictly necessary to run make at this point in the sequence, and in fact may generate errors if necessary handlers aren't yet available. One may optionally

```
cd Dakota/src
make nidr-files
```

to only rebuild generated dependencies of dakota.input.nspec.

Chapter 11

Update NIDRProblemDescDB.C in Dakota/src

Many keywords have data associated with them: an integer, a floating-point number, a string, or arrays of such entities. Data requirements are specified in dakota.input.nspec by the tokens INTEGER, REAL, STRING, INTEGERLIST, REALLIST, STRINGLIST. (Some keywords have no associated data and hence no such token.) After each keyword and data token, the dakota.input.nspec file specifies functions that the NIDR parser should call to record the appearance of the keyword and deal with any associated data. The general form of this specification is

{ startfcn, startdata, stopfcn, stopdata }

i.e., a brace-enclosed list of one to four functions and data pointers, with trailing entities taken to be zero if not present; zero for a function means no function will be called. The startfcn must deal with any associated data. Otherwise, the distinction between startfcn and stopfcn is relevant only to keywords that begin a group of keywords (enclosed in parentheses or square brackets). The startfcn is called before other entities in the group are processed, and the stop function is called after they are processed. Top-level keywords often have both startfcn and stopfcn; stopfcn is uncommon but possible for lower-level keywords. The startdata and (if needed) stopdata values are usually pointers to little structures that provide keyword-specific details to generic functions for startfcn and stopfcn. Some keywords that begin groups (such as "approx_problem" within the top-level "strategy" keyword) have no need of either a startfcn or a stopfcn; this is indicated by "`{0}`".

Most of the things within braces in dakota.input.nspec are invocations of macros defined in NIDRProblemDescDB.C. The macros simplify writing dakota.input.nspec and make it more readable. Most macro invocations refer to little structures defined in NIDRProblemDescDB.C, usually with the help of other macros, some of which have different definitions in different parts of NIDRProblemDescDB.C. When adding a keyword to dakota.input.nspec, you may need to add a structure definition or even introduce a new data type. NIDRProblemDescDB.C has sections corresponding to each top-level keyword. The top-level keywords are in alphabetical order, and most entities in the section for a top-level keyword are also in alphabetical order. While not required, it is probably good practice to maintain this structure, as it makes things easier to find.

Any integer, real, or string data associated with a keyword are provided to the keyword's startfcn, whose second argument is a pointer to a `Values` structure, defined in header file `nidr.h`.

Example 1: if you added the specification:

```
[method_setting REAL {method_setting_start, &method_setting_details} ]
```

you would provide a function

```
    void NIDRProblemDescDB::
method_setting_start(const char *keyname, Values *val, void **g, void *v)
{ ... }
```

in NIDRProblemDescDB.C. In this example, argument `&method_setting_details` would be passed as `v`, `val->n` (the number of values) would be 1 and `*val->r` would be the REAL value given for the `method_setting` keyword. The `method_setting_start` function would suitably store this value with the help of `method_setting_details`.

For some top-level keywords, `g` (the third argument to the startfcn and stopfcn) provides access to a relevant context. For example, `method_start` (the startfcn for the top-level `method` keyword) executes

```
DataMethod *dm = new DataMethod;
*g = (void*)dm;
```

(and supplies a couple of default values to `dm`). The start functions for lower-level keywords within the `method` keyword get access to `dm` through their `g` arguments. Here is an example:

```
 void NIDRProblemDescDB::
method_str(const char *keyname, Values *val, void **g, void *v)
{
        (*(DataMethod**)g)->**(String DataMethod::**)v = *val->s;
        }
```

In this example, `v` points to a pointer-to-member, and an assignment is made to one of the components of the DataMethod object pointed to by `*g`. The corresponding stopfcn for the top-level `method` keyword is

```
 void NIDRProblemDescDB::
method_stop(const char *keyname, Values *val, void **g, void *v)
{
        DataMethod *p = *(DataMethod**)g;
        pDDBInstance->dataMethodList.insert(*p);
        delete p;
        }
```

which copies the now populated DataMethod object to the right place and cleans up.

Example 2: if you added the specification

```
[method_setting REALLIST {{N_mdm(RealL,methodCoeffs)}
```

then method_RealL (defined in NIDRProblemDescDB.C) would be called as the startfcn, and methodCoeffs would be the name of a (currently nonexistent) component of DataMethod. The N_mdm macro is defined in NIDRProblemDescDB.C; among other things, it turns `RealL` into `NIDRProblemDescDB::method_RealL`. This function is used to process lists of REAL values for several keywords. By looking at the source, you can see that the list values are val->r[i] for 0 <= `i` < val->n.

Chapter 12

Update ProblemDescDB.C in Dakota/src

12.1 Augment/update get_<data_type>() functions

The next update step involves extending the database retrieval functions in ProblemDescDB.C. These retrieval functions accept an identifier string and return a database attribute of a particular type, e.g., a RealVector:

```
const RealVector& get_rdv(const String& entry_name);
```

The implementation of each of these functions contains tables of possible entry_name values and associated pointer-to-member values. There is one table for each relevant top-level keyword, with the top-level keyword omitted from the names in the table. Since binary search is used to look for names in these tables, each table must be kept in alphabetical order of its entry names. For example,

```
...
else if ((L = Begins(entry_name, "model."))) {
  if (dbRep->methodDBLocked)
      Locked_db();

  #define P &DataModelRep::
  static KW<RealVector, DataModelRep> RVdmo[] = {     // must be sorted
      {"nested.primary_response_mapping", P primaryRespCoeffs},
      {"nested.secondary_response_mapping", P secondaryRespCoeffs},
      {"surrogate.kriging_conmin_seed", P krigingConminSeed},
      {"surrogate.kriging_correlations", P krigingCorrelations},
      {"surrogate.kriging_max_correlations", P krigingMaxCorrelations},
      {"surrogate.kriging_min_correlations", P krigingMinCorrelations}};
  #undef P

  KW<RealVector, DataModelRep> *kw;
  if ((kw = (KW<RealVector, DataModelRep>*)Binsearch(RVdmo, L)))
      return dbRep->dataModelIter->dataModelRep->*kw->p;
}
```

is the "model" portion of ProblemDescDB::get_rdv(). Based on entry_name, it returns the relevant attribute from a DataModel object. Since there may be multiple model specifications, the `dataModelIter` list iterator identifies which node in the list of DataModel objects is used. In particular, `dataModelList` contains a list of all of the `data_model` objects, one for each time a top-level `model` keyword was seen by the parser. The particular model object used for the data retrieval is managed by `dataModelIter`, which is set in a `set_db_list_nodes()` operation that will not be described here.

There may be multiple DataMethod, DataModel, DataVariables, DataInterface, and/or DataResponses objects. However, only one strategy specification is currently allowed so a list of DataStrategy objects is not needed. Rather, ProblemDescDB::strategySpec is the lone DataStrategy object.

To augment the get_<data_type>() functions, add table entries with new identifier strings and pointer-to-member values that address the appropriate data attributes from the Data class object. The style for the identifier strings is a top-down hierarchical description, with specification levels separated by periods and words separated with underscores, e.g., `"keyword.group_specification.individual_specification"`. Use the `dbRep->listIter->attribute` syntax for variables, interface, responses, and method specifications. For example, the `method_setting` example attribute would be added to `get_drv()` as:

```
{"method_name.method_setting", P methodSetting},
```

inserted at the beginning of the `RVdmo` array shown above (since the name in the existing first entry, i.e., "nested.primary_response_mapping", comes alphabetically after "method_name.method_setting").

Chapter 13

Update Corresponding Data Classes

In this step, we extend the Data class definitions (DataStrategy, DataMethod, DataModel, DataVariables, DataInterface, and/or DataResponses) to include the new attributes referenced in Update NIDRProblemDescDB.C in Dakota/src and Augment/update get_<data_type>() functions.

13.1 Update the Data class header file

Add a new attribute to the public data for each of the new specifications. Follow the style guide for class attribute naming conventions (or mimic the existing code).

13.2 Update the .C file

Define defaults for the new attributes in the constructor initialization list. Add the new attributes to the assign() function for use by the copy constructor and assignment operator. Add the new attributes to the write(MPIPackBuffer&), read(MPIUnpackBuffer&), and write(ostream&) functions, paying careful attention to the use of a consistent ordering.

Chapter 14

Use get_<data_type>() Functions

At this point, the new specifications have been mapped through all of the database classes. The only remaining step is to retrieve the new data within the constructors of the classes that need it. This is done by invoking the get_-<data_type>() function on the ProblemDescDB object using the identifier string you selected in Augment/update get_<data_type>() functions. For example:

```
const String& interface_type = problem_db.get_string("interface.type");
```

passes the `"interface.type"` identifier string to the ProblemDescDB::get_string() retrieval function, which returns the desired attribute from the active DataInterface object.

Warning:

> Use of the get_<data_type>() functions is restricted to class constructors, since only in class constructors are the data list iterators (i.e., `dataMethodIter`, `dataModelIter`, `dataVariablesIter`, `dataInterfaceIter`, and `dataResponsesIter`) guaranteed to be set correctly. Outside of the constructors, the database list nodes will correspond to the last set operation, and may not return data from the desired list node.

Chapter 15

Update the Documentation

Doxygen comments should be added to the Data class headers for the new attributes, and the reference manual sections describing the portions of dakota.input.nspec that have been modified should be updated. In particular, the reference manual tables summarizing keywords provide help data to the Jaguar user interface so need to be kept updated.

Chapter 16

Understanding Iterator Flow

This page explains the various phases comprising Iterator::run_iterator(). Prior to Iterator construction, when command-line options are parsed, Boolean run mode flags corresponding to PRERUN, RUN, and POSTRUN are set in ParallelLibrary. If the user didn't specify any specific run modes, the default is for all three to be true (all phases will execute).

Iterator is constructed.

When called, `run_iterator()` sequences:

- `initialize_run()`: unconditionally called, virtual. Performs common initialization such as allocating workspaces, setting communicators and evaluation counts. When re-implementing this virtual, a derived class must call its nearest parent's initialize_run(), typically *before* performing its own implementation steps.
- *Not implemented: pre-run input*
- IF PRERUN, invoke `pre_run()`: virtual function; default no-op. Purpose: derived classes should implement `pre_run()` if they are able to generate all parameter sets (variables) at once, separate from `run()`. Derived implementations should call their nearest parent's pre_run(), typically *before* performing their own steps.
- IF PRERUN, invoke `pre_output()`: non-virtual function; if user requested, output variables to file.
- *Not implemented: run input*
- IF RUN, invoke virtual function `run()`. Purpose: at a minimum, evaluate parameter sets through computing responses; for iterators without pre/post capability, their entire implementation is in `run()` and this is a reasonable default for new Iterators.
- *Not implemented: run output*
- IF POSTRUN, invoke `post_input()`: virtual function, default only print helpful message on mode. Purpose: derived iterators supporting post-run input from file must implement to read file and populate variables/responses (and possibly best points) appropriately. Implementations must check if the user requested file input.

- IF POSTRUN, invoke `post_run()`: virtual function. Purpose: generate statistics / final results. Any analysis that can be done solely on tabular data read by `post_input()` can be done here. Derived re-implementations should call their nearest parent's post-run(), typically *after* performing their specific post-run activities.
- *Not implemented: post-run output*
- `finalize_run()`: unconditionally called, virtual. Purpose: free workspaces. Default base class behavior is no-op, however, derived implementations should call their nearest parent's finalize_run *after* performing their specialized portions.

Iterator is destructed.

Chapter 17

Interfacing with DAKOTA as a Library

Chapter 18

Introduction

It is possible to link the DAKOTA toolkit into another application for use as an algorithm library. This section describes facilities which permit this type of integration.

As part of the normal DAKOTA build process, where `Dakota/configure --prefix=PREFIX` has been run prior to `make` and `make install`, a `libdakota.a` is created and a copy of it is placed in `PREFIX/lib` (PREFIX defaults to /usr/local/Dakota). This library contains all source files from `Dakota/src` excepting the main.C, restart_util.C, and library_mode.C main programs. This library may be linked with another application through inclusion of `-ldakota` on the link line. Library and header paths may also be specified using the `-L` and `-I` compiler options (using `PREFIX/lib` and `PREFIX/include`, respectively). Depending on the configuration used when building this library, other libraries for the vendor optimizers and vendor packages will also be needed to resolve DAKOTA symbols for DOT, NPSOL, OPT++, NCSUOpt, LHS, Teuchos, etc. Copies of these libraries are also placed in `Dakota/lib`. Refer to Linking against the DAKOTA library for additional information.

Warning:

> Users may interface to DAKOTA as a library within other software applications provided that they abide by the terms of the GNU Lesser General Public License (LGPL). Refer to `http://www.gnu.org/licenses/lgpl.html` or contact the DAKOTA team for additional information.

Attention:

> The use of DAKOTA as an algorithm library should be distinguished from the linking of simulations within DAKOTA using the direct application interface (see DirectApplicInterface). In the former, DAKOTA is providing algorithm services to another software application, and in the latter, a linked simulation is providing analysis services to DAKOTA. It is not uncommon for these two capabilities to be used in combination, where a simulation framework provides both the "front end" and the "back end" for DAKOTA.

Chapter 19

Quick start: examples and test code

To learn by example, refer to the files PluginSerialDirectApplicInterface.[CH] and PluginParallelDirectApplicInterface.[CH] in Dakota/src for simple examples of serial and parallel plug-in interfaces. The file library_mode.C in Dakota/src provides example usage of these plug-ins within a mock simulator program that demonstrates the required object instantiation syntax in combination with the three problem database population approaches (input file parsing, data node insertion, and mixed mode). All of this code may be compiled and tested by configuring DAKOTA using the `--with-plugin` option.

Chapter 20

Comparison to main.C

The procedure for utilizing DAKOTA as a library within another application involves a number of steps that are similar to those used in the stand-alone DAKOTA application. The stand-alone procedure can be viewed in the file main.C, and the differences for the library approach are most easily explained with reference to that file. The basic steps of executing DAKOTA include instantiating the ParallelLibrary, CommandLineHandler, and ProblemDescDB objects; managing the DAKOTA input file (ProblemDescDB::manage_inputs()); specifying restart files and output streams (ParallelLibrary::specify_outputs_restart()); and instantiating the Strategy and running it (Strategy::run_strategy()). When using DAKOTA as an algorithm library, the operations are quite similar, although command line information (argc, argv, and therefore CommandLineHandler) will not in general be accessible. In particular, main.C can pass argc and argv into the ParallelLibrary and CommandLineHandler constructors and then pass the CommandLineHandler object into ProblemDescDB::manage_inputs() and ParallelLibrary::specify_outputs_restart(). In an algorithm library approach, a CommandLineHandler object is not instantiated and overloaded forms of the ParallelLibrary constructor, ProblemDescDB::manage_inputs(), and ParallelLibrary::specify_outputs_restart() are used.

The overloaded forms of these functions are as follows. For instantiation of the ParallelLibrary object, the default constructor may be used. This constructor assumes that MPI is administered by the parent application such that the MPI configuration will be detected rather than explicitly created (i.e., DAKOTA will not call MPI_Init or MPI_Finalize). In code, the instantiation

```
ParallelLibrary parallel_lib(argc, argv);
```

is replaced with

```
ParallelLibrary parallel_lib;
```

In the case of specifying restart files and output streams, the call to

```
parallel_lib.specify_outputs_restart(cmd_line_handler);
```

should be replaced with its overloaded form in order to pass the required information through the parameter list

```
parallel_lib.specify_outputs_restart(std_output_filename, std_error_filename,
  read_restart_filename, write_restart_filename, stop_restart_evals);
```

where file names for standard output and error and restart read and write as well as the integer number of restart evaluations are passed through the parameter list rather than read from the command line of the main DAKOTA

program. The definition of these attributes is performed elsewhere in the parent application (e.g., specified in the parent application input file or GUI). In this function call, specify `NULL` for any files not in use, which will elicit the desired subset of the following defaults: standard output and standard error are directed to the terminal, no restart input, and restart output to file `dakota.rst`. The `stop_restart_evals` specification is an optional parameter with a default of 0, which indicates that restart processing should process all records. If no overrides of these defaults are intended, the call to `specify_outputs_restart()` may be omitted entirely.

With respect to alternate forms of ProblemDescDB::manage_inputs(), the following section describes different approaches to populating data within DAKOTA's problem description database. It is this database from which all DAKOTA objects draw data upon instantiation.

Chapter 21

Problem database population

Now that the ProblemDescDB object has been instantiated, we must populate it with data, either via parsing an input file, direct data insertion, or a mixed approach, as described in the following sections.

21.1 Input file parsing

The simplest approach to linking an application with the DAKOTA library is to rely on DAKOTA's normal parsing system to populate DAKOTA's problem database (ProblemDescDB) through the reading of an input file. The disadvantage to this approach is the requirement for an additional input file beyond those already required by the parent application.

In this approach, the main.C call to

```
problem_db.manage_inputs(cmd_line_handler);
```

would be replaced with its overloaded form

```
problem_db.manage_inputs(dakota_input_file);
```

where the file name for the DAKOTA input is passed through the parameter list rather than read from the command line of the main DAKOTA program. Again, the definition of the DAKOTA input file name is performed elsewhere in the parent application (e.g., specified in the parent application input file or GUI). Refer to run_dakota_parse() in library_mode.C for a complete example listing.

ProblemDescDB::manage_inputs() invokes ProblemDescDB::parse_inputs() (which in turn invokes ProblemDescDB::check_input()), ProblemDescDB::broadcast(), and ProblemDescDB::post_process(), which are lower level functions that will be important in the following two sections. Thus, the input file parsing approach may employ a single coarse grain function to coordinate all aspects of problem database population, whereas the two approaches to follow will use lower level functions to accomplish a finer grain of control.

21.2 Data node insertion

This approach is more involved than the previous approach, but it allows the application to publish all needed data to DAKOTA's database directly, thereby eliminating the need for the parsing of a separate DAKOTA input file.

In this case, ProblemDescDB::manage_inputs() is not called. Rather, DataStrategy, DataMethod, DataModel, DataVariables, DataInterface, and DataResponses objects are instantiated and populated with the desired problem data. These objects are then published to the problem database using ProblemDescDB::insert_node(), e.g.:

```
// instantiate the data object
DataMethod data_method;

// set the attributes within the data object
data_method.methodName = "nond_sampling";
...

// publish the data object to the ProblemDescDB
problem_db.insert_node(data_method);
```

The data objects are populated with their default values upon instantiation, so only the non-default values need to be specified. Refer to the DataStrategy, DataMethod, DataModel, DataVariables, DataInterface, and DataResponses class documentation and source code for lists of attributes and their defaults.

The default strategy is `single_method`, which runs a single iterator on a single model, and the default model is `single`, so it is not necessary to instantiate and publish a DataStrategy or DataModel object if advanced multi-component capabilities are not required. Rather, instantiation and insertion of a single DataMethod, DataVariables, DataInterface, and DataResponses object is sufficient for basic DAKOTA capabilities.

Once the data objects have been published to the ProblemDescDB object, calls to

```
problem_db.check_input();
problem_db.broadcast();
problem_db.post_process();
```

will perform basic database error checking, broadcast a packed MPI buffer of the specification data to other processors, and post-process specification data to fill in vector defaults (scalar defaults are handled in the Data class constructors), respectively. For parallel applications, processor rank 0 should be responsible for Data node population and insertion and the call to ProblemDescDB::check_input(), and all processors should participate in ProblemDescDB::broadcast() and ProblemDescDB::post_process(). Moreover, preserving the order shown assures that large default vectors are not transmitted by MPI. Refer to run_dakota_data() in library_mode.C for a complete example listing.

21.3 Mixed mode

In this case, we will combine the parsing of a DAKOTA input file with some direct database updates. The motivation for this approach arises in large-scale applications where large vectors can be awkward to specify in a DAKOTA input file. The first step is to parse the input file, but rather than using

```
problem_db.manage_inputs(dakota_input_file);
```

as described in Input file parsing, we will use the lower level function

```
problem_db.parse_inputs(dakota_input_file);
```

to provide a finer grain of control. The passed input file `dakota_input_file` must contain all required inputs. Since vector data like variable values/bounds/tags, linear/nonlinear constraint coefficients/bounds, etc. are optional, these potentially large vector specifications can be omitted from the input file. Only the variable/response counts, e.g.:

```
method
        linear_inequality_constraints = 500

variables
        continuous_design = 1000

responses
        objective_functions = 1
        nonlinear_inequality_constraints = 100000
```

are required in this case. To update the data omissions from their defaults, one uses the ProblemDescDB::set() family of overloaded functions, e.g.

```
Dakota::RealVector drv(1000, 1.); // vector of length 1000, values initialized
    to 1.
problem_db.set("variables.continuous_design.initial_point", drv);
```

where the string identifiers are the same identifiers used when pulling information from the database using one of the get_<datatype>() functions (refer to the source code of ProblemDescDB.C for a full list). However, the supported ProblemDescDB::set() options are a restricted subset of the database attributes, focused on vector inputs that can be large scale.

If performing these updates within the constructor of a DirectApplicInterface extension/derivation (see Defining the direct application interface), then this code is sufficient since the database is unlocked, the active list nodes of the ProblemDescDB have been set for you, and the correct strategy/method/model/variables/interface/responses specification instance will get updated. The difficulty in this case stems from the order of instantiation. Since the Variables and Response instances are constructed in the base Model class, prior to construction of Interface instances in derived Model classes, database information related to Variables and Response objects will have already been extracted by the time the Interface constructor is invoked and the database update will not propagate.

Therefore, it is preferred to perform these operations at a higher level (e.g., within your main program), prior to Strategy instantiation and execution, such that instantiation order is not an issue. However, in this case, it is necessary to explicitly manage the list nodes of the ProblemDescDB using a specification instance identifier that corresponds to an identifier from the input file, e.g.:

```
problem_db.set_db_variables_node("MY_VARIABLES_ID");
Dakota::RealVector drv(1000, 1.); // vector of length 1000, values initialized
    to 1.
problem_db.set("variables.continuous_design.initial_point", drv);
```

Alternatively, rather than setting just a single data node, all data nodes may be set using a method specification identifier:

```
problem_db.set_db_list_nodes("MY_METHOD_ID");
```

since the method specification is responsible for identifying a model specification, which in turn identifies variables, interface, and responses specifications. If hardwiring specification identifiers is undesirable, then

```
problem_db.resolve_top_method();
```

can also be used to deduce the active method specification and set all list nodes based on it. This is most appropriate in the case where only single specifications exist for method/model/variables/interface/responses. In each of these cases, setting list nodes unlocks the corresponding portions of the database, allowing set/get operations.

Once all direct database updates have been performed in this manner, calls to ProblemDescDB::broadcast() and ProblemDescDB::post_process() should be used on all processors. The former will broadcast a packed MPI

buffer with the aggregated set of specification data from rank 0 to other processors, and the latter will post-process specification data to fill in any vector defaults that have not yet been provided through either file parsing or direct updates (Note: scalar defaults are handled in the Data class constructors). Refer to run_dakota_mixed() in library_mode.C for a complete example listing.

Chapter 22

Instantiating the strategy

With the ProblemDescDB object populated with problem data, we may now instantiate the strategy.

```
// instantiate the strategy
Strategy selected_strategy(problem_db);
```

Following strategy construction, all MPI communicator partitioning has been performed and the ParallelLibrary instance may be interrogated for parallel configuration data. For example, the lowest level communicators in DAKOTA's multilevel parallel partitioning are the analysis communicators, which can be retrieved using:

```
// retrieve the set of analysis communicators for simulation initialization:
// one analysis comm per ParallelConfiguration (PC), one PC per Model.
Array<MPI_Comm> analysis_comms = parallel_lib.analysis_intra_communicators();
```

These communicators can then be used for initializing parallel simulation instances, where the number of MPI communicators in the array corresponds to one communicator per ParallelConfiguration instance.

Chapter 23

Defining the direct application interface

When employing a library interface to DAKOTA, it is frequently desirable to also use a direct interface between DAKOTA and the simulation. There are two approaches to defining this direct interface.

23.1 Extension

The first approach involves extending the existing DirectApplicInterface class to support additional direct simulation interfaces. In this case, a new simulation interface function can be added to Dakota/src/DirectApplicInterface.[CH] for the simulation of interest. If the new function will not be a member function, then the following prototype should be used in order to pass the required data:

```
int sim(const Dakota::Variables& vars, const Dakota::ActiveSet& set,
        Dakota::Response& response);
```

If the new function will be a member function, then this can be simplified to

```
int sim();
```

since the data access can be performed through the DirectApplicInterface class attributes.

This simulation can then be added to the logic blocks in DirectApplicInterface::derived_map_ac(). In addition, DirectApplicInterface::derived_map_if() and DirectApplicInterface::derived_map_of() can be extended to perform pre- and post-processing tasks if desired, but this is not required.

While this approach is the simplest, it has the disadvantage that the DAKOTA library may need to be recompiled when the simulation or its direct interface is modified. If it is desirable to maintain the independence of the DAKOTA library from the host application, then the following derivation approach should be employed.

23.2 Derivation

The second approach is to derive a new interface from DirectApplicInterface in order to redefine several virtual functions. A typical derived class declaration might be

```
namespace SIM {
```

```
class SerialDirectApplicInterface: public Dakota::DirectApplicInterface
{
public:

  // Constructor and destructor

  SerialDirectApplicInterface(const Dakota::ProblemDescDB& problem_db);
  ~SerialDirectApplicInterface();

protected:

  // Virtual function redefinitions

  int derived_map_if(const Dakota::String& if_name);
  int derived_map_ac(const Dakota::String& ac_name);
  int derived_map_of(const Dakota::String& of_name);

private:

  // Data
}

} // namespace SIM
```

where the new derived class resides in the simulation's namespace. Similar to the case of Extension, the DirectApplicInterface::derived_map_ac() function is the required redefinition, and DirectApplicInterface::derived_map_if() and DirectApplicInterface::derived_map_of() are optional.

The new derived interface object (from namespace SIM) must now be plugged into the strategy. In the simplest case of a single model and interface, one could use

```
  // retrieve the interface of interest
  ModelList& all_models  = problem_db.model_list();
  Model&     first_model = *all_models.begin();
  Interface& interface   = first_model.interface();
  // plug in the new direct interface instance (DB does not need to be set)
  interface.assign_rep(new SIM::SerialDirectApplicInterface(problem_db), false);
```

from within the Dakota namespace. In a more advanced case of multiple models and multiple interface plug-ins, one might use

```
  // retrieve the list of Models from the Strategy
  ModelList& models = problem_db.model_list();
  // iterate over the Model list
  for (ModelLIter ml_iter = models.begin(); ml_iter != models.end(); ml_iter++) {

    Interface& interface = ml_iter->interface();
    if (interface.interface_type() == "direct" &&
        interface.analysis_drivers().contains("SIM") ) {
      // set the correct list nodes within the DB prior to new instantiations
      problem_db.set_db_model_nodes(ml_iter->model_id());
      // plug in the new direct interface instance
      interface.assign_rep(new SIM::SerialDirectApplicInterface(problem_db), fals
      e);
    }
  }
```

In the case where the simulation interface instance should manage parallel simulations within the context of an MPI communicator, one should pass in the relevant analysis communicator(s) to the derived constructor. For the

latter case of looping over a set of models, the simplest approach of passing a single analysis communicator would use code similar to

```
const ParallelLevel& ea_level =
  ml_iter->parallel_configuration_iterator()->ea_parallel_level();
const MPI_Comm& analysis_comm = ea_level.server_intra_communicator();
interface.assign_rep(new SIM::ParallelDirectApplicInterface(problem_db, analysi
    s_comm),
                     false);
```

Since Models may be used in multiple parallel contexts and may therefore have a set of parallel configurations, a more general approach would extract and pass an array of analysis communicators to allow initialization for each of the parallel configurations.

New derived direct interface instances inherit various attributes of use in configuring the simulation. In particular, the ApplicationInterface::parallelLib reference provides access to MPI communicator data (e.g., the analysis communicators discussed in Instantiating the strategy), DirectApplicInterface::analysisDrivers provides the analysis driver names specified by the user in the input file, and DirectApplicInterface::analysisComponents provides additional analysis component identifiers (such as mesh file names) provided by the user which can be used to distinguish different instances of the same simulation interface. It is worth noting that inherited attributes that are set as part of the parallel configuration (instead of being extracted from the ProblemDescDB) will be set to their defaults following construction of the base class instance for the derived class plug-in. It is not until run-time (i.e., within derived_map_if/derived_map_ac/derived_map_of) that the parallel configuration settings are re-propagated to the plug-in instance. This is the reason that the analysis communicator should be passed in to the constructor of a parallel plug-in, if the constructor will be responsible for parallel application initialization.

Chapter 24

Additional updates

As part of strategy instantiation, all problem specification data is extracted from ProblemDescDB as various objects are constructed. Therefore, any updates that need to be performed following strategy instantiation must be performed through direct set operations on the constructed objects. In the previous section, the process for updating the Interface object used within a Model was shown. To update other data such as variable values/bounds/tags or response bounds/targets/tags, refer to the set functions documented in Iterator and Model. As an example, the following code updates the active continuous variable values, which will be employed as the initial guess for certain classes of Iterators:

```
ModelList& all_models  = problem_db.model_list();
Model&     first_model = *all_models.begin();
Dakota::RealVector drv(1000, 1.); // vector of length 1000, values initialized
    to 1.
first_model.continuous_variables(drv);
```

Chapter 25

Executing the strategy

Finally, with simulation configuration and plug-ins completed, we execute the strategy:

```
// run the strategy
selected_strategy.run_strategy();
```

Chapter 26

Retrieving data after a run

After executing the strategy, final results can be obtained through the use of Strategy::variables_results() and Strategy::response_results(), e.g.:

```
// retrieve the final parameter values
const Variables& vars = selected_strategy.variables_results();

// retrieve the final response values
const Response& resp  = selected_strategy.response_results();
```

In the case of optimization, the final design is returned, and in the case of uncertainty quantification, the final statistics are returned.

Chapter 27

Linking against the DAKOTA library

This section presumes DAKOTA has been compiled with configure/make and installed to `PREFIX` using 'make install'. While the DAKOTA build system offers the most up-to-date guidance for what libraries are needed to link against a particular version of DAKOTA, a typical case is presented here. Note that depending on how you configured DAKOTA, some of the following libraries may not be available (for example NPSOL, DOT, NLPQL) -- check which appear in `$PREFIX/lib`.

As of DAKOTA 5.0, `-levidence` is no longer required and `-lgsl` is optional (discouraged due to GPL), depending on how DAKOTA was configured.

Post DAKOTA 5.0, `-lquadrature` has been renamed to `-lsparsegrid`. Also the DFFTPACK library should be integrated into libpecos, so `-ldfftpack` should not be needed and NKM should be integrated into libsurfpack, so `-lnkm` should not be needed.

Note that as of DAKOTA 5.2, `-lnewmat` is no longer required but additional Boost libraries are needed (`-lboost_regex -lboost_filesystem -lboost_system`) as a result of migration from legacy DAKOTA utilities to more modern Boost components. It should also be noted that DAKOTA relies on VERSION 2 of Boost.Filesystem which is provided in the source distribution under packages/boost (Boost.Filesystem VERSION 3 is NOT supported at this time).

```
DAKOTA_LIBS = -L${PREFIX}/lib -ldakota -lteuchos -lpecos -llhs         \
              -lsparsegrid -lsurfpack -lconmin -lddace -ldot -lfsudace \
              -ljega -lcport -lnlpql -lnpsol -loptpp -lpsuade          \
              -lncsuopt -lcolin -linterfaces -lmomh -lscolib -lpebbl   \
              -ltinyxml -lutilib -l3po -lhopspack -lnidr -lamplsolver  \
              -lboost_signals -lboost_regex -lboost_filesystem         \
              -lboost_system -llapack -lblas
```

You may also need `funcadd0.o`, `-lfl`, `-lexpat`, and, if linking with system-provided GSL, `-lgslcblas`. The AMPL solver library may require `-ldl`. System compiler and math libraries may also need to be included. If configuring with graphics, you will need to add `-lDGraphics` and system X libraries (partial list here):

```
-lXpm -lXm -lXt -lXmu -lXp -lXext -lX11 -lSM -lICE
```

CMake support for library users is experimental. At present library names and inclusion requirements differ slightly when using CMake, e.g, libpecos_src and libdakota_src instead of libpecos and libdakota, respectively. A representative (non-authoritative) set of necessary link libraries is shown here for illustration purposes only:

```
-ldakota_src -lteuchos -lnidr -lpecos -lpecos_src -llhs -lmods -lmod
```

```
-ldfftpack -lsparsegrid -lsurfpack -lsurfpack -lutilib -lcolin -linterfaces
-lscolib -l3po -lpebbl -ltinyxml -lconmin -ldace -lanalyzer -lrandom
-lsampling -lbose -ldot -lfsudace -lhopspack -ljega -ljega_fe -lmoga
-lsoga -leutils -lutilities -lncsuopt -lnlpql -lcport -lnpsol -loptpp
-lpsuade -lDGraphics -lamplsolver -lboost_regex -lboost_signals
-lboost_filesystem -lboost_system -lSM -lICE -lX11 -lXext -lXm -lXpm
-lXmu -lpthread -llapack -lcurl -ldl -lutilib -ltinyxml -lm -lboost_regex
-lboost_filesystem -lboost_system -lexpat -lboost_signals -ljega -lteuchos
-lblas -llapack
```

We have experienced problems with the creation of `libamplsolver.a` on some platforms. Please use the DAKOTA mailing lists for help with any problems.

Finally, it is important to use the same C++ compiler (possibly an MPI wrapper) for compiling DAKOTA and your application and declare the compiler define `-DHAVE_CONFIG_H` when including header files from DAKOTA. This ensures that the platform configuration settings are properly propagated.

Chapter 28

Summary

To utilize the DAKOTA library within a parent software application, the basic steps of main.C and the order of invocation of these steps should be mimicked from within the parent application. Of these steps, ParallelLibrary instantiation, ProblemDescDB::manage_inputs() and ParallelLibrary::specify_outputs_restart() require the use of overloaded forms in order to function in an environment without direct command line access and, potentially, without file parsing. Additional optional steps not performed in main.C include the extension/derivation of the direct interface and the retrieval of strategy results after a run.

DAKOTA's library mode is now in production use within several Sandia and external simulation codes/frameworks.

Chapter 29

Performing Function Evaluations

Performing function evaluations is one of the most critical functions of the DAKOTA software. It can also be one of the most complicated, as a variety of scheduling approaches and parallelism levels are supported. This complexity manifests itself in the code through a series of cascaded member functions, from the top level model evaluation functions, through various scheduling routines, to the low level details of performing a system call, fork, or direct function invocation. This section provides an overview of the primary classes and member functions involved.

Chapter 30

Synchronous function evaluations

For a synchronous (i.e., blocking) mapping of parameters to responses, an iterator invokes Model::compute_-response() to perform a function evaluation. This function is all that is seen from the iterator level, as underlying complexities are isolated. The binding of this top level function with lower level functions is as follows:

- Model::compute_response() utilizes Model::derived_compute_response() for portions of the response computation specific to derived model classes.
- Model::derived_compute_response() directly or indirectly invokes Interface::map().
- Interface::map() utilizes ApplicationInterface::derived_map() for portions of the mapping specific to derived application interface classes.

Chapter 31

Asynchronous function evaluations

For an asynchronous (i.e., nonblocking) mapping of parameters to responses, an iterator invokes Model::asynch_compute_response() multiple times to queue asynchronous jobs and then invokes either Model::synchronize() or Model::synchronize_nowait() to schedule the queued jobs in blocking or nonblocking fashion. Again, these functions are all that is seen from the iterator level, as underlying complexities are isolated. The binding of these top level functions with lower level functions is as follows:

- Model::asynch_compute_response() utilizes Model::derived_asynch_compute_response() for portions of the response computation specific to derived model classes.
- This derived model class function directly or indirectly invokes Interface::map() in asynchronous mode, which adds the job to a scheduling queue.
- Model::synchronize() or Model::synchronize_nowait() utilize Model::derived_synchronize() or Model::derived_synchronize_nowait() for portions of the scheduling process specific to derived model classes.
- These derived model class functions directly or indirectly invoke Interface::synch() or Interface::synch_nowait().
- For application interfaces, these interface synchronization functions are responsible for performing evaluation scheduling in one of the following modes:
  - asynchronous local mode (using ApplicationInterface::asynchronous_local_evaluations() or ApplicationInterface::asynchronous_local_evaluations_nowait())
  - message passing mode (using ApplicationInterface::self_schedule_evaluations() or ApplicationInterface::static_schedule_evaluations() on the iterator master and ApplicationInterface::serve_evaluations_synch() or ApplicationInterface::serve_evaluations_peer() on the servers)
  - hybrid mode (using ApplicationInterface::self_schedule_evaluations() or ApplicationInterface::static_schedule_evaluations() on the iterator master and ApplicationInterface::serve_evaluations_asynch() on the servers)
- These scheduling functions utilize ApplicationInterface::derived_map() and ApplicationInterface::derived_map_asynch() for portions of asynchronous job launching specific to derived application interface classes, as well as ApplicationInterface::derived_synch() and

ApplicationInterface::derived_synch_nowait() for portions of job capturing specific to derived application interface classes.

Chapter 32

Analyses within each function evaluation

The discussion above covers the parallelism level of concurrent function evaluations serving an iterator. For the parallelism level of concurrent analyses serving a function evaluation, similar schedulers are involved (ForkApplicInterface::synchronous_local_analyses(), ForkApplicInterface::asynchronous_local_analyses(), ApplicationInterface::self_schedule_analyses(), ApplicationInterface::serve_analyses_synch(), ForkApplicInterface::serve_analyses_asynch()) to support synchronous local, asynchronous local, message passing, and hybrid modes. Not all of the schedulers are elevated to the ApplicationInterface level since the system call and direct function interfaces do not yet support nonblocking local analyses (and therefore support synchronous local and message passing modes, but not asynchronous local or hybrid modes). Fork interfaces, however, support all modes of analysis parallelism.

Chapter 33

Working with Variable Containers and Views

Variable views control the subset of variable types that are active and inactive within a particular iterative study. For design optimization and uncertainty quantification (UQ), for example, the active variables view consists of design or uncertain types, respectively, and any other variable types are carried along invisible to the iterative algorithm being employed. For parameter studies and design of experiments, however, a variable subset view is not imposed and all variables are active. Selected UQ methods can also be toggled into an "All" view using the `all_variables` input specification. When not in an All view, finer gradations within the uncertain variable sets are also relevant: probabilistic methods (reliability, stochastic expansion) view aleatory uncertain variables as active, nonprobabilistic methods (interval, evidence) view epistemic uncertain variables as active, and a few UQ methods (sampling) view both as active. In a more advanced NestedModel use case such as optimization under uncertainty, design variables are active in the outer optimization context and the uncertain variables are active in the inner UQ context, with an additional requirement on the inner UQ level to return derivatives with respect to its "inactive" variables (i.e., the design variables) for use in the outer optimization loop.

For efficiency, contiguous arrays of data store variable information for each of the domain types (continuous, discrete integer, and discrete real), but active and inactive views into them permit selecting subsets in a given context. This management is encapsulated into the Variables and SharedVariablesData classes. This page clarifies concepts of merged vs. mixed, fine-grained vs. aggregated types, domain types, and views into contiguous arrays.

We begin with an overview of the storage and management concept, for which the following two sections describe the storage of variable values and meta-data about their organization, used in part to manage views. They are intended to communicate rationale to maintainers of Variables and SharedVariablesData classes. The final section provides a discussion of active and inactive views.

Chapter 34

Storage in Variables

As described in the Main Page Variables, a Variables object manages variable types (design, aleatory uncertain, epistemic uncertain, and state) and domain types (continuous, discrete integer, and discrete real) and supports different approaches to either distinguishing among these types or aggregating them. Two techniques are used in cooperation to accomplish this management: (1) class specialization (MergedVariables or MixedVariables) and (2) views into contiguous variable arrays. The latter technique is used whenever it can satisfy the requirement, with fallback to class specialization when it cannot. In particular, aggregation or separation of variable types can be accomplished with views, but for aggregation or separation of variable domains, we must resort to class specialization in order to relax discrete domain types. In this class specialization, a MergedVariables object combines continuous and discrete types (relaxing integers to reals) whereas a MixedVariables object maintains the integer/real distinction throughout.

The core data for a Variables instance is stored in a set of three continguous arrays, corresponding to the domain types: allContinuousVars, allDiscreteIntVars, and allDiscreteRealVars, unique to each Variables instance.

Within the core variable data arrays, data corresponding to different aggregated variable types are stored in sequence for each domain type:

- continuous: [design, aleatory uncertain, epistemic uncertain, state]
- discrete integer: [design, aleatory uncertain, (epistemic uncertain), state]
- discrete real: [design, aleatory uncertain, (epistemic uncertain), state]

Note there are currently no epistemic discrete variables. This domain type ordering (continuous, discrete integer, discrete real) and aggregated variable type ordering (design, aleatory uncertain, epistemic uncertain, state) is preserved whenever distinct types are flattened into single contiguous arrays. Note that the aleatory and epistemic uncertain variables contain sub-types for different distributions (e.g., normal, uniform, histogram, poisson), and discrete integer types include both integer ranges and integer set sub-types. All sub-types are ordered according to their order of appearance in dakota.input.nspec.

When relaxing in MixedVariables, the allContinuousVars will also aggregate the discrete types, such that they contain ALL design, then ALL uncertain, then ALL state variables, each in aggregated type order; the allDiscreteIntVars and allDiscreteRealVars arrays are empty.

Chapter 35

Storage in SharedVariablesData

Each Variables instance contains a reference-counted SharedVariablesData object that stores information on the variables configuration. This configuration data includes counts, types, IDs, and labels, which are often the same across many Variables instances. Thus, SharedVariablesData is intended to reduce the memory footprint by allowing the sharing of a single copy of redundant information among different Variables instances.

One of the purposes of this shared information is to support mappings between variable types, IDs, and indices into the storage arrays. Variable "types" refer to the fine-grained variable types a user would specify in an input file, as enumerated in DataVariables.H, e.g, CONTINUOUS_DESIGN, WEIBULL_UNCERTAIN, DISCRETE_-STATE_RANGE, etc. variablesComponents is a map from these variable types to counts of how many are present.

In contrast, the variablesCompsTotals array stores total counts of each "aggregated type" (design, aleatory uncertain, epistemic uncertain, state) which might be selected to be active in a given view. Thus this array has length 12 to track the combinations of three domain type storage arrays with four possible aggregated variable types: {continuous, discrete integer, discrete real} x {design, aleatory uncertain, epistemic uncertain, state}. For example, the first entry of this array stores the number of continuous design variables, the second the number of discrete integer design (including both discrete design range and discrete design set integer types), and the last the number of discrete real state variables.

The arrays allContinuousTypes, allDiscreteIntTypes, and allDiscreteRealTypes are sized to match the corresponding core domain type storage arrays. They track the fine-grained variable type stored in that entry of the data array (since when relaxed, the continuous array may be storing data corresponding to discrete data).

Finally allContinuousIds stores the 1-based IDs of the variables stored in the allContinuousVars array, i.e., the variable number of all the problem variables considered as a single contiguous set, in aggregate type order. For merged (relaxed) views, mergedDiscreteIds stores the 1-based IDs of the variables which have been relaxed into the continuous array.

These counts, types, and IDs are most commonly used within the Model classes for mappings between variables objects at different levels of a model recursion. See, for example, the variable mappings in the NestedModel constructor.

Chapter 36

Active and inactive views

The pair SharedVariablesDataRep::variablesView tracks the active and inactive views of the data, with values taken from the enum in DataVariables.H. The valid values include EMPTY and the combinations {merged, mixed} x {all, design, aleatory uncertain, epistemic uncertain, uncertain, state}. The ALL cases indicate aggregation of the design, aleatory uncertain, epistemic uncertain, and state types, whereas the DISTINCT cases indicate either no aggregation (design, aleatory uncertain, epistemic uncertain, state) or reduced aggregation (aleatory+epistemic uncertain). The active view is determined by the algorithm in use, managed in Variables::get_view(). Any inactive view is set based on higher level iteration within a model recursion (e.g., a NestedModel), which enables lower level iteration to return derivatives with respect to variables that are active at the higher level. In the case where there is no higher level iteration, then the inactive view will remain EMPTY. It is important to stress that "inactive" at one level corresponds to active at another, and therefore the inactive set of variables should not be interpreted as the strict complement of the active set of variables; rather, active and inactive are both subsets whose union may still be a subset of the total container (more precise terminology might involve "primary" active and "secondary" active or similar). An active complement view could potentially be supported in the future, should the need arise, although this view would require management of non-contiguous portions of the aggregated arrays.

Given these groupings (views), the active and inactive subsets of the allContinuousVars, allDiscreteIntVars, and allDiscreteRealVars arrays are always contiguous, permitting vector views of the underlying data using either Teuchos::View (for numerical vectors) or Boost.MultiArray (for book-keeping arrays) views.

When a Variables envelope is constructed, its letter is initialized to either a MergedVariables or MixedVariables object depending on the active view. The derived classes size the contiguous storage arrays to accomodate all the problem variables, and then initialize active views into them, which could involve either subsets (DISTINCT active views) or views of the full arrays (ALL active views). Inactive views, on the other hand, are initialized during construction of a model recursion (e.g., a call to Model::inactive_view() in the NestedModel constructor). Thus, active variable subsets are always available but inactive variable subsets will be EMPTY prior to them being initialized within a Model recursion.

Accessors for continuous variables include:

- continuous_variables(): returns the active view which might return all (ALL views) or a subset (DISTINCT views) such as design, uncertain, only aleatory uncertain, etc.
- inactive_continuous_variables(): returns the inactive view which which is either a subset or empty
- all_continuous_variables(): returns the full vector allContinuousVars

and this pattern is followed for active/inactive/all access to discrete_int_variables() and discrete_real_variables() as well as for labels, IDs, and types in SharedVariablesData and variable bounds in Constraints.

Chapter 37

Todo List

Chapter 38

Namespace Index

38.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Dakota (The primary namespace for DAKOTA) 107
SIM (A sample namespace for derived classes that use assign_rep() to plug facilities into DAKOTA) . . 263

Chapter 39

Class Index

39.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

ActiveSet . . . 265
AnalysisCode . . . 269
  ForkAnalysisCode . . . 467
  SysCallAnalysisCode . . . 907
Approximation . . . 294
  GaussProcApproximation . . . 478
  PecosApproximation . . . 779
  SurfpackApproximation . . . 881
  TANA3Approximation . . . 912
  TaylorApproximation . . . 915
APPSEvalMgr . . . 309
BaseConstructor . . . 316
BiStream . . . 317
BoStream . . . 320
COLINApplication . . . 323
CommandShell . . . 336
Constraints . . . 349
  MergedConstraints . . . 541
  MixedConstraints . . . 555
DataInterface . . . 375
DataMethod . . . 377
DataMethodRep . . . 379
DataModel . . . 394
DataModelRep . . . 396
DataResponses . . . 401
DataResponsesRep . . . 403
DataStrategy . . . 408
DataStrategyRep . . . 410

DataVariables . . . 413
DataVariablesRep . . . 415
DiscrepancyCorrection . . . 441
Driver . . . 451
Evaluator . . . 458
EvaluatorCreator . . . 465
GetLongOpt . . . 485
  CommandLineHandler . . . 334
Graphics . . . 489
Interface . . . 503
  ApplicationInterface . . . 280
    DirectApplicInterface . . . 431
      ParallelDirectApplicInterface . . . 749
      SerialDirectApplicInterface . . . 838
    ForkApplicInterface . . . 469
    GridApplicInterface . . . 493
    SysCallApplicInterface . . . 909
  ApproximationInterface . . . 303
Iterator . . . 513
  Analyzer . . . 274
    NonD . . . 630
      NonDCalibration . . . 646
        NonDBayesCalibration . . . 644
          NonDGPMSABayesCalibration . . . 671
          NonDQUESOBayesCalibration . . . 716
      NonDExpansion . . . 653
        NonDPolynomialChaos . . . 708
        NonDStochCollocation . . . 735
      NonDIntegration . . . 677
        NonDCubature . . . 649
        NonDQuadrature . . . 711
        NonDSparseGrid . . . 731
      NonDInterval . . . 680
        NonDGlobalInterval . . . 662
          NonDGlobalEvidence . . . 660
          NonDGlobalSingleInterval . . . 669
        NonDLHSInterval . . . 685
          NonDLHSEvidence . . . 683
          NonDLHSSingleInterval . . . 690
        NonDLocalInterval . . . 694
          NonDLocalEvidence . . . 692
          NonDLocalSingleInterval . . . 706
      NonDReliability . . . 719
        NonDGlobalReliability . . . 666
        NonDLocalReliability . . . 697
      NonDSampling . . . 724
        NonDAdaptImpSampling . . . 640

NonDIncremLHSSampling . . . 674
NonDLHSSampling . . . 687
PStudyDACE . . . 797
DDACEDesignCompExp . . . 426
FSUDesignCompExp . . . 473
ParamStudy . . . 772
PSUADEDesignCompExp . . . 800
Verification . . . 933
RichExtrapVerification . . . 828
Minimizer . . . 545
LeastSq . . . 536
NL2SOLLeastSq . . . 618
NLSSOLLeastSq . . . 627
SNLLLeastSq . . . 853
Optimizer . . . 741
APPSOptimizer . . . 312
COLINOptimizer . . . 327
CONMINOptimizer . . . 341
DOTOptimizer . . . 446
JEGAOptimizer . . . 527
NCSUOptimizer . . . 598
NLPQLPOptimizer . . . 621
NPSOLOptimizer . . . 737
SNLLOptimizer . . . 858
SurrBasedMinimizer . . . 896
EffGlobalMinimizer . . . 453
SurrBasedGlobalMinimizer . . . 885
SurrBasedLocalMinimizer . . . 887
Model . . . 559
NestedModel . . . 602
RecastModel . . . 805
SingleModel . . . 847
SurrogateModel . . . 902
DataFitSurrModel . . . 362
HierarchSurrModel . . . 496
MPIPackBuffer . . . 592
MPIUnpackBuffer . . . 595
NL2Res . . . 617
NoDBBaseConstructor . . . 629
ParallelConfiguration . . . 747
ParallelLevel . . . 750
ParallelLibrary . . . 754
ParamResponsePair . . . 768
partial_prp_equality . . . 777
partial_prp_hash . . . 778
ProblemDescDB . . . 786
NIDRProblemDescDB . . . 612
RecastBaseConstructor . . . 804

Response . . . 814
ResponseRep . . . 820
SensAnalysisGlobal . . . 831
SharedVariablesData . . . 839
SharedVariablesDataRep . . . 842
SNLLBase . . . 850
  SNLLLeastSq . . . 853
  SNLLOptimizer . . . 858
SOLBase . . . 866
  NLSSOLLeastSq . . . 627
  NPSOLOptimizer . . . 737
Strategy . . . 869
  ConcurrentStrategy . . . 338
  HybridStrategy . . . 501
    CollaborativeHybridStrategy . . . 332
    EmbeddedHybridStrategy . . . 456
    SequentialHybridStrategy . . . 833
  SingleMethodStrategy . . . 845
String . . . 878
TrackerHTTP . . . 917
Variables . . . 920
  MergedVariables . . . 543
  MixedVariables . . . 557

Chapter 40

Class Index

40.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

ActiveSet (Container class for active set tracking information. Contains the active set request vector and the derivative variables vector) . . . 265
AnalysisCode (Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations) . . . 269
Analyzer (Base class for NonD, DACE, and ParamStudy branches of the iterator hierarchy) . . . 274
ApplicationInterface (Derived class within the interface class hierarchy for supporting interfaces to simulation codes) . . . 280
Approximation (Base class for the approximation class hierarchy) . . . 294
ApproximationInterface (Derived class within the interface class hierarchy for supporting approximations to simulation-based results) . . . 303
APPSEvalMgr (Evaluation manager class for APPSPACK) . . . 309
APPSOptimizer (Wrapper class for APPSPACK) . . . 312
BaseConstructor (Dummy struct for overloading letter-envelope constructors) . . . 316
BiStream (The binary input stream class. Overloads the >> operator for all data types) . . . 317
BoStream (The binary output stream class. Overloads the << operator for all data types) . . . 320
COLINApplication . . . 323
COLINOptimizer (Wrapper class for optimizers defined using COLIN) . . . 327
CollaborativeHybridStrategy (Strategy for hybrid minimization using multiple collaborating optimization and nonlinear least squares methods) . . . 332
CommandLineHandler (Utility class for managing command line inputs to DAKOTA) . . . 334
CommandShell (Utility class which defines convenience operators for spawning processes with system calls) . . . 336
ConcurrentStrategy (Strategy for multi-start iteration or pareto set optimization) . . . 338
CONMINOptimizer (Wrapper class for the CONMIN optimization library) . . . 341
Constraints (Base class for the variable constraints class hierarchy) . . . 349
DataFitSurrModel (Derived model class within the surrogate model branch for managing data fit surrogates (global and local)) . . . 362
DataInterface (Handle class for interface specification data) . . . 375
DataMethod (Handle class for method specification data) . . . 377

DataMethodRep (Body class for method specification data) . . . 379
DataModel (Handle class for model specification data) . . . 394
DataModelRep (Body class for model specification data) . . . 396
DataResponses (Handle class for responses specification data) . . . 401
DataResponsesRep (Body class for responses specification data) . . . 403
DataStrategy (Handle class for strategy specification data) . . . 408
DataStrategyRep (Body class for strategy specification data) . . . 410
DataVariables (Handle class for variables specification data) . . . 413
DataVariablesRep (Body class for variables specification data) . . . 415
DDACEDesignCompExp (Wrapper class for the DDACE design of experiments library) . . . 426
DirectApplicInterface (Derived application interface class which spawns simulation codes and testers using direct procedure calls) . . . 431
DiscrepancyCorrection (Base class for discrepancy corrections) . . . 441
DOTOptimizer (Wrapper class for the DOT optimization library) . . . 446
Driver (A subclass of the JEGA front end driver that exposes the individual protected methods to execute the algorithm) . . . 451
EffGlobalMinimizer (Implementation of Efficient Global Optimization/Least Squares algorithms) . . . 453
EmbeddedHybridStrategy (Strategy for closely-coupled hybrid minimization, typically involving the embedding of local search methods within global search methods) . . . 456
Evaluator (An evaluator specialization that knows how to interact with Dakota) . . . 458
EvaluatorCreator (A specialization of the JEGA::FrontEnd::EvaluatorCreator that creates a new instance of a Evaluator) . . . 465
ForkAnalysisCode (Derived class in the AnalysisCode class hierarchy which spawns simulations using forks) . . . 467
ForkApplicInterface (Derived application interface class which spawns simulation codes using forks) . 469
FSUDesignCompExp (Wrapper class for the FSUDace QMC/CVT library) . . . 473
GaussProcApproximation (Derived approximation class for Gaussian Process implementation) . . . 478
GetLongOpt (GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France)) . . . 485
Graphics (Single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc) . . . 489
GridApplicInterface (Derived application interface class which spawns simulation codes using grid services such as Condor or Globus) . . . 493
HierarchSurrModel (Derived model class within the surrogate model branch for managing hierarchical surrogates (models of varying fidelity)) . . . 496
HybridStrategy (Base class for hybrid minimization strategies) . . . 501
Interface (Base class for the interface class hierarchy) . . . 503
Iterator (Base class for the iterator class hierarchy) . . . 513
JEGAOptimizer (A version of Dakota::Optimizer for instantiation of John Eddy's Genetic Algorithms (JEGA)) . . . 527
LeastSq (Base class for the nonlinear least squares branch of the iterator hierarchy) . . . 536
MergedConstraints (Derived class within the Constraints hierarchy which employs the merged data view)541
MergedVariables (Derived class within the Variables hierarchy which employs the merged data view) . 543
Minimizer (Base class for the optimizer and least squares branches of the iterator hierarchy) . . . 545
MixedConstraints (Derived class within the Constraints hierarchy which employs the default data view (no variable or domain type array merging)) . . . 555
MixedVariables (Derived class within the Variables hierarchy which employs the default data view (no variable or domain type array merging)) . . . 557
Model (Base class for the model class hierarchy) . . . 559

MPIPackBuffer (Class for packing MPI message buffers) 592
MPIUnpackBuffer (Class for unpacking MPI message buffers) 595
NCSUOptimizer (Wrapper class for the NCSU DIRECT optimization library) 598
NestedModel (Derived model class which performs a complete sub-iterator execution within every evaluation of the model) 602
NIDRProblemDescDB (The derived input file database utilizing the new IDR parser) 612
NL2Res (Auxiliary information passed to calcr and calcj via ur) 617
NL2SOLLeastSq (Wrapper class for the NL2SOL nonlinear least squares library) 618
NLPQLPOptimizer (Wrapper class for the NLPQLP optimization library, Version 2.0) 621
NLSSOLLeastSq (Wrapper class for the NLSSOL nonlinear least squares library) 627
NoDBBaseConstructor (Dummy struct for overloading constructors used in on-the-fly instantiations) . 629
NonD (Base class for all nondetermistic iterators (the DAKOTA/UQ branch)) 630
NonDAdaptImpSampling (Class for the Adaptive Importance Sampling methods within DAKOTA) . . 640
NonDBayesCalibration (Base class for Bayesian inference: generates posterior distribution on model parameters given experimental data) 644
NonDCalibration 646
NonDCubature (Derived nondeterministic class that generates N-dimensional numerical cubature points for evaluation of expectation integrals) 649
NonDExpansion (Base class for polynomial chaos expansions (PCE) and stochastic collocation (SC)) . 653
NonDGlobalEvidence (Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ) 660
NonDGlobalInterval (Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification) 662
NonDGlobalReliability (Class for global reliability methods within DAKOTA/UQ) 666
NonDGlobalSingleInterval (Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification) 669
NonDGPMSABayesCalibration (Generates posterior distribution on model parameters given experiment data) 671
NonDIncremLHSSampling (Performs icremental LHS sampling for uncertainty quantification) 674
NonDIntegration (Derived nondeterministic class that generates N-dimensional numerical integration points for evaluation of expectation integrals) 677
NonDInterval (Base class for interval-based methods within DAKOTA/UQ) 680
NonDLHSEvidence (Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ) . 683
NonDLHSInterval (Class for the LHS-based interval methods within DAKOTA/UQ) 685
NonDLHSSampling (Performs LHS and Monte Carlo sampling for uncertainty quantification) 687
NonDLHSSingleInterval (Class for pure interval propagation using LHS) 690
NonDLocalEvidence (Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ) 692
NonDLocalInterval (Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification) 694
NonDLocalReliability (Class for the reliability methods within DAKOTA/UQ) 697
NonDLocalSingleInterval (Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification) 706
NonDPolynomialChaos (Nonintrusive polynomial chaos expansion approaches to uncertainty quantification) 708
NonDQuadrature (Derived nondeterministic class that generates N-dimensional numerical quadrature points for evaluation of expectation integrals over uncorrelated standard normals/uniforms/exponentials/betas/gammas) 711
NonDQUESOBayesCalibration (Bayesian inference using the QUESO library from UT Austin) 716
NonDReliability (Base class for the reliability methods within DAKOTA/UQ) 719

NonDSampling (Base class for common code between NonDLHSSampling, NonDIncremLHSSampling, and NonDAdaptImpSampling) . . . 724
NonDSparseGrid (Derived nondeterministic class that generates N-dimensional Smolyak sparse grids for numerical evaluation of expectation integrals over independent standard random variables) 731
NonDStochCollocation (Nonintrusive stochastic collocation approaches to uncertainty quantification) . 735
NPSOLOptimizer (Wrapper class for the NPSOL optimization library) . . . 737
Optimizer (Base class for the optimizer branch of the iterator hierarchy) . . . 741
ParallelConfiguration (Container class for a set of ParallelLevel list iterators that collectively identify a particular multilevel parallel configuration) . . . 747
ParallelDirectApplicInterface (Sample derived interface class for testing parallel simulator plug-ins using assign_rep()) . . . 749
ParallelLevel (Container class for the data associated with a single level of communicator partitioning) 750
ParallelLibrary (Class for partitioning multiple levels of parallelism and managing message passing within these levels) . . . 754
ParamResponsePair (Container class for a variables object, a response object, and an evaluation id) . . 768
ParamStudy (Class for vector, list, centered, and multidimensional parameter studies) . . . 772
partial_prp_equality (Predicate for comparing ONLY the interfaceId and Vars attributes of PRPair) . . . 777
partial_prp_hash (Wrapper to delegate to the ParamResponsePair hash_value function) . . . 778
PecosApproximation (Derived approximation class for global basis polynomials) . . . 779
ProblemDescDB (The database containing information parsed from the DAKOTA input file) . . . 786
PStudyDACE (Base class for managing common aspects of parameter studies and design of experiments methods) . . . 797
PSUADEDesignCompExp (Wrapper class for the PSUADE library) . . . 800
RecastBaseConstructor (Dummy struct for overloading constructors used in on-the-fly Model instantiations) . . . 804
RecastModel (Derived model class which provides a thin wrapper around a sub-model in order to recast the form of its inputs and/or outputs) . . . 805
Response (Container class for response functions and their derivatives. Response provides the handle class) . . . 814
ResponseRep (Container class for response functions and their derivatives. ResponseRep provides the body class) . . . 820
RichExtrapVerification (Class for Richardson extrapolation for code and solution verification) . . . 828
SensAnalysisGlobal (Class for a utility class containing correlation calculations and variance-based decomposition) . . . 831
SequentialHybridStrategy (Strategy for sequential hybrid minimization using multiple optimization and nonlinear least squares methods on multiple models of varying fidelity) . . . 833
SerialDirectApplicInterface (Sample derived interface class for testing serial simulator plug-ins using assign_rep()) . . . 838
SharedVariablesData (Container class encapsulating variables data that can be shared among a set of Variables instances) . . . 839
SharedVariablesDataRep (The representation of a SharedVariablesData instance. This representation, or body, may be shared by multiple SharedVariablesData handle instances) . . . 842
SingleMethodStrategy (Simple fall-through strategy for running a single iterator on a single model) . . 845
SingleModel (Derived model class which utilizes a single interface to map variables into responses) . . 847
SNLLBase (Base class for OPT++ optimization and least squares methods) . . . 850
SNLLLeastSq (Wrapper class for the OPT++ optimization library) . . . 853
SNLLOptimizer (Wrapper class for the OPT++ optimization library) . . . 858
SOLBase (Base class for Stanford SOL software) . . . 866
Strategy (Base class for the strategy class hierarchy) . . . 869

String (Dakota::String class, used as main string class for Dakota) 878
SurfpackApproximation (Derived approximation class for Surfpack approximation classes. Interface between Surfpack and Dakota) 881
SurrBasedGlobalMinimizer (The global surrogate-based minimizer which sequentially minimizes and updates a global surrogate model without trust region controls) 885
SurrBasedLocalMinimizer (Class for provably-convergent local surrogate-based optimization and nonlinear least squares) 887
SurrBasedMinimizer (Base class for local/global surrogate-based optimization/least squares) 896
SurrogateModel (Base class for surrogate models (DataFitSurrModel and HierarchSurrModel)) 902
SysCallAnalysisCode (Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls) 907
SysCallApplicInterface (Derived application interface class which spawns simulation codes using system calls) 909
TANA3Approximation (Derived approximation class for TANA-3 two-point exponential approximation (a multipoint approximation)) 912
TaylorApproximation (Derived approximation class for first- or second-order Taylor series (a local approximation)) 915
TrackerHTTP (TrackerHTTP: a usage tracking module that uses HTTP/HTTPS via the curl library) . . 917
Variables (Base class for the variables class hierarchy) 920
Verification (Base class for managing common aspects of verification studies) 933

Chapter 41

File Index

41.1 File List

Here is a list of all documented files with brief descriptions:

dll_api.C (This file contains a DakotaRunner class, which launches DAKOTA) 935
dll_api.h (API for DLL interactions) 937
JEGAOptimizer.C (Contains the implementation of the JEGAOptimizer class) 939
JEGAOptimizer.H (Contains the definition of the JEGAOptimizer class) 940
library_mode.C (File containing a mock simulator main for testing DAKOTA in library mode) 941
library_split.C (File containing a mock simulator main for testing DAKOTA in library mode on a split communicator) 943
main.C (File containing the main program for DAKOTA) 944
restart_util.C (File containing the DAKOTA restart utility main program) 945

Chapter 42

Namespace Documentation

42.1 Dakota Namespace Reference

The primary namespace for DAKOTA.

Classes

- class AnalysisCode

  Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations.

- class ApplicationInterface

  Derived class within the interface class hierarchy for supporting interfaces to simulation codes.

- class ApproximationInterface

  Derived class within the interface class hierarchy for supporting approximations to simulation-based results.

- class APPSEvalMgr

  Evaluation manager class for APPSPACK.

- class APPSOptimizer

  Wrapper class for APPSPACK.

- class COLINApplication
- class COLINOptimizer

  Wrapper class for optimizers defined using COLIN.

- class CollaborativeHybridStrategy

  Strategy for hybrid minimization using multiple collaborating optimization and nonlinear least squares methods.

- class GetLongOpt

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France).

- class CommandLineHandler

  Utility class for managing command line inputs to DAKOTA.

- class CommandShell

  Utility class which defines convenience operators for spawning processes with system calls.

- class ConcurrentStrategy

  Strategy for multi-start iteration or pareto set optimization.

- class CONMINOptimizer

  Wrapper class for the CONMIN optimization library.

- class ActiveSet

  Container class for active set tracking information. Contains the active set request vector and the derivative variables vector.

- class Analyzer

  Base class for NonD, DACE, and ParamStudy branches of the iterator hierarchy.

- class Approximation

  Base class for the approximation class hierarchy.

- class BiStream

  The binary input stream class. Overloads the >> operator for all data types.

- class BoStream

  The binary output stream class. Overloads the << operator for all data types.

- class Constraints

  Base class for the variable constraints class hierarchy.

- class Graphics

  The Graphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.

- class Interface

  Base class for the interface class hierarchy.

- class Iterator

  Base class for the iterator class hierarchy.

- class LeastSq

  Base class for the nonlinear least squares branch of the iterator hierarchy.

- class Minimizer

  Base class for the optimizer and least squares branches of the iterator hierarchy.

- class Model

  Base class for the model class hierarchy.

- class NonD

  Base class for all nondetermistic iterators (the DAKOTA/UQ branch).

- class Optimizer

  Base class for the optimizer branch of the iterator hierarchy.

- class PStudyDACE

  Base class for managing common aspects of parameter studies and design of experiments methods.

- class ResponseRep

  Container class for response functions and their derivatives. ResponseRep provides the body class.

- class Response

  Container class for response functions and their derivatives. Response provides the handle class.

- class Strategy

  Base class for the strategy class hierarchy.

- class String

  Dakota::String class, used as main string class for Dakota.

- class Variables

  Base class for the variables class hierarchy.

- class Verification

  Base class for managing common aspects of verification studies.

- class DataFitSurrModel

  Derived model class within the surrogate model branch for managing data fit surrogates (global and local).

- class DataInterface

  Handle class for interface specification data.

- class DataMethodRep

  Body class for method specification data.

- class DataMethod

  Handle class for method specification data.

- class DataModelRep

Body class for model specification data.

- class DataModel

  Handle class for model specification data.

- class DataResponsesRep

  Body class for responses specification data.

- class DataResponses

  Handle class for responses specification data.

- class DataStrategyRep

  Body class for strategy specification data.

- class DataStrategy

  Handle class for strategy specification data.

- class DataVariablesRep

  Body class for variables specification data.

- class DataVariables

  Handle class for variables specification data.

- class DDACEDesignCompExp

  Wrapper class for the DDACE design of experiments library.

- class DirectApplicInterface

  Derived application interface class which spawns simulation codes and testers using direct procedure calls.

- class DiscrepancyCorrection

  Base class for discrepancy corrections.

- class DOTOptimizer

  Wrapper class for the DOT optimization library.

- class EffGlobalMinimizer

  Implementation of Efficient Global Optimization/Least Squares algorithms.

- class EmbeddedHybridStrategy

  Strategy for closely-coupled hybrid minimization, typically involving the embedding of local search methods within global search methods.

- class ForkAnalysisCode

  Derived class in the AnalysisCode class hierarchy which spawns simulations using forks.

- class ForkApplicInterface

Derived application interface class which spawns simulation codes using forks.

- class FSUDesignCompExp

  Wrapper class for the FSUDace QMC/CVT library.

- class GaussProcApproximation

  Derived approximation class for Gaussian Process implementation.

- struct BaseConstructor

  Dummy struct for overloading letter-envelope constructors.

- struct NoDBBaseConstructor

  Dummy struct for overloading constructors used in on-the-fly instantiations.

- struct RecastBaseConstructor

  Dummy struct for overloading constructors used in on-the-fly Model instantiations.

- class GridApplicInterface

  Derived application interface class which spawns simulation codes using grid services such as Condor or Globus.

- class HierarchSurrModel

  Derived model class within the surrogate model branch for managing hierarchical surrogates (models of varying fidelity).

- class HybridStrategy

  Base class for hybrid minimization strategies.

- class JEGAOptimizer

  A version of Dakota::Optimizer for instantiation of John Eddy's Genetic Algorithms (JEGA).

- class MergedConstraints

  Derived class within the Constraints hierarchy which employs the merged data view.

- class MergedVariables

  Derived class within the Variables hierarchy which employs the merged data view.

- class MixedConstraints

  Derived class within the Constraints hierarchy which employs the default data view (no variable or domain type array merging).

- class MixedVariables

  Derived class within the Variables hierarchy which employs the default data view (no variable or domain type array merging).

- class MPIPackBuffer

  Class for packing MPI message buffers.

- class MPIUnpackBuffer

  Class for unpacking MPI message buffers.

- class NCSUOptimizer

  Wrapper class for the NCSU DIRECT optimization library.

- class NestedModel

  Derived model class which performs a complete sub-iterator execution within every evaluation of the model.

- class NIDRProblemDescDB

  The derived input file database utilizing the new IDR parser.

- struct NL2Res

  Auxiliary information passed to calcr and calcj via ur.

- class NL2SOLLeastSq

  Wrapper class for the NL2SOL nonlinear least squares library.

- class NLPQLPOptimizer

  Wrapper class for the NLPQLP optimization library, Version 2.0.

- class NLSSOLLeastSq

  Wrapper class for the NLSSOL nonlinear least squares library.

- class NonDAdaptImpSampling

  Class for the Adaptive Importance Sampling methods within DAKOTA.

- class NonDBayesCalibration

  Base class for Bayesian inference: generates posterior distribution on model parameters given experimental data.

- class NonDCalibration
- class NonDCubature

  Derived nondeterministic class that generates N-dimensional numerical cubature points for evaluation of expectation integrals.

- class NonDExpansion

  Base class for polynomial chaos expansions (PCE) and stochastic collocation (SC).

- class NonDGlobalEvidence

  Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ.

- class NonDGlobalInterval

  Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification.

- class NonDGlobalReliability

Class for global reliability methods within DAKOTA/UQ.

- class NonDGlobalSingleInterval

  Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification.

- class NonDGPMSABayesCalibration

  Generates posterior distribution on model parameters given experiment data.

- class NonDIncremLHSSampling

  Performs icremental LHS sampling for uncertainty quantification.

- class NonDIntegration

  Derived nondeterministic class that generates N-dimensional numerical integration points for evaluation of expectation integrals.

- class NonDInterval

  Base class for interval-based methods within DAKOTA/UQ.

- class NonDLHSEvidence

  Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ.

- class NonDLHSInterval

  Class for the LHS-based interval methods within DAKOTA/UQ.

- class NonDLHSSampling

  Performs LHS and Monte Carlo sampling for uncertainty quantification.

- class NonDLHSSingleInterval

  Class for pure interval propagation using LHS.

- class NonDLocalEvidence

  Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ.

- class NonDLocalInterval

  Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification.

- class NonDLocalReliability

  Class for the reliability methods within DAKOTA/UQ.

- class NonDLocalSingleInterval

  Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification.

- class NonDPolynomialChaos

  Nonintrusive polynomial chaos expansion approaches to uncertainty quantification.

- class NonDQuadrature

  Derived nondeterministic class that generates N-dimensional numerical quadrature points for evaluation of expectation integrals over uncorrelated standard normals/uniforms/exponentials/betas/gammas.

- class NonDQUESOBayesCalibration

  Bayesian inference using the QUESO library from UT Austin.

- class NonDReliability

  Base class for the reliability methods within DAKOTA/UQ.

- class NonDSampling

  Base class for common code between NonDLHSSampling, NonDIncremLHSSampling, and NonDAdaptImpSampling.

- class NonDSparseGrid

  Derived nondeterministic class that generates N-dimensional Smolyak sparse grids for numerical evaluation of expectation integrals over independent standard random variables.

- class NonDStochCollocation

  Nonintrusive stochastic collocation approaches to uncertainty quantification.

- class NPSOLOptimizer

  Wrapper class for the NPSOL optimization library.

- class ParallelLevel

  Container class for the data associated with a single level of communicator partitioning.

- class ParallelConfiguration

  Container class for a set of ParallelLevel list iterators that collectively identify a particular multilevel parallel configuration.

- class ParallelLibrary

  Class for partitioning multiple levels of parallelism and managing message passing within these levels.

- class ParamResponsePair

  Container class for a variables object, a response object, and an evaluation id.

- class ParamStudy

  Class for vector, list, centered, and multidimensional parameter studies.

- class PecosApproximation

  Derived approximation class for global basis polynomials.

- class ProblemDescDB

  The database containing information parsed from the DAKOTA input file.

- struct partial_prp_hash

  wrapper to delegate to the ParamResponsePair hash_value function

- struct partial_prp_equality

  predicate for comparing ONLY the interfaceId and Vars attributes of PRPair

- class PSUADEDesignCompExp

  Wrapper class for the PSUADE library.

- class RecastModel

  Derived model class which provides a thin wrapper around a sub-model in order to recast the form of its inputs and/or outputs.

- class RichExtrapVerification

  Class for Richardson extrapolation for code and solution verification.

- class SensAnalysisGlobal

  Class for a utility class containing correlation calculations and variance-based decomposition.

- class SequentialHybridStrategy

  Strategy for sequential hybrid minimization using multiple optimization and nonlinear least squares methods on multiple models of varying fidelity.

- class SharedVariablesDataRep

  The representation of a SharedVariablesData instance. This representation, or body, may be shared by multiple SharedVariablesData handle instances.

- class SharedVariablesData

  Container class encapsulating variables data that can be shared among a set of Variables instances.

- class SingleMethodStrategy

  Simple fall-through strategy for running a single iterator on a single model.

- class SingleModel

  Derived model class which utilizes a single interface to map variables into responses.

- class SNLLBase

  Base class for OPT++ optimization and least squares methods.

- class SNLLLeastSq

  Wrapper class for the OPT++ optimization library.

- class SNLLOptimizer

  Wrapper class for the OPT++ optimization library.

- class SOLBase

Base class for Stanford SOL software.

- class SurfpackApproximation

  Derived approximation class for Surfpack approximation classes. Interface between Surfpack and Dakota.

- class SurrBasedGlobalMinimizer

  The global surrogate-based minimizer which sequentially minimizes and updates a global surrogate model without trust region controls.

- class SurrBasedLocalMinimizer

  Class for provably-convergent local surrogate-based optimization and nonlinear least squares.

- class SurrBasedMinimizer

  Base class for local/global surrogate-based optimization/least squares.

- class SurrogateModel

  Base class for surrogate models (DataFitSurrModel and HierarchSurrModel).

- class SysCallAnalysisCode

  Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls.

- class SysCallApplicInterface

  Derived application interface class which spawns simulation codes using system calls.

- class TANA3Approximation

  Derived approximation class for TANA-3 two-point exponential approximation (a multipoint approximation).

- class TaylorApproximation

  Derived approximation class for first- or second-order Taylor series (a local approximation).

- class TrackerHTTP

  TrackerHTTP: a usage tracking module that uses HTTP/HTTPS via the curl library.

Typedefs

- typedef double **Real**
- typedef Teuchos::SerialDenseVector< int, Real > **RealVector**
- typedef Teuchos::SerialDenseVector< int, int > **IntVector**
- typedef Teuchos::SerialDenseMatrix< int, Real > **RealMatrix**
- typedef Teuchos::SerialSymDenseMatrix< int, Real > **RealSymMatrix**
- typedef std::deque< bool > **BoolDeque**
- typedef std::vector< BoolDeque > **BoolDequeArray**
- typedef std::vector< Real > **RealArray**
- typedef std::vector< RealArray > **Real2DArray**
- typedef std::vector< int > **IntArray**

- typedef std::vector< IntArray > **Int2DArray**
- typedef std::vector< short > **ShortArray**
- typedef std::vector< unsigned short > **UShortArray**
- typedef std::vector< UShortArray > **UShort2DArray**
- typedef std::vector< UShort2DArray > **UShort3DArray**
- typedef std::vector< size_t > **SizetArray**
- typedef std::vector< SizetArray > **Sizet2DArray**
- typedef std::vector< String > **StringArray**
- typedef std::vector< StringArray > **String2DArray**
- typedef boost::multi_array_types::index_range **idx_range**
- typedef boost::multi_array< String, 1 > **StringMultiArray**
- typedef StringMultiArray::array_view< 1 >::type **StringMultiArrayView**
- typedef StringMultiArray::const_array_view< 1 >::type **StringMultiArrayConstView**
- typedef boost::multi_array< unsigned short, 1 > **UShortMultiArray**
- typedef UShortMultiArray::array_view< 1 >::type **UShortMultiArrayView**
- typedef UShortMultiArray::const_array_view< 1 >::type **UShortMultiArrayConstView**
- typedef boost::multi_array< size_t, 1 > **SizetMultiArray**
- typedef SizetMultiArray::array_view< 1 >::type **SizetMultiArrayView**
- typedef SizetMultiArray::const_array_view< 1 >::type **SizetMultiArrayConstView**
- typedef std::vector< RealVector > **RealVectorArray**
- typedef std::vector< RealVectorArray > **RealVector2DArray**
- typedef std::vector< RealMatrix > **RealMatrixArray**
- typedef std::vector< RealSymMatrix > **RealSymMatrixArray**
- typedef std::vector< IntVector > **IntVectorArray**
- typedef std::vector< Variables > **VariablesArray**
- typedef std::vector< Response > **ResponseArray**
- typedef std::vector< ParamResponsePair > **PRPArray**
- typedef std::vector< PRPArray > **PRP2DArray**
- typedef std::vector< Model > **ModelArray**
- typedef std::vector< Iterator > **IteratorArray**
- typedef std::list< bool > **BoolList**
- typedef std::list< int > **IntList**
- typedef std::list< size_t > **SizetList**
- typedef std::list< Real > **RealList**
- typedef std::list< String > **StringList**
- typedef std::list< Variables > **VariablesList**
- typedef std::list< Interface > **InterfaceList**
- typedef std::list< Response > **ResponseList**
- typedef std::list< Model > **ModelList**
- typedef std::list< Iterator > **IteratorList**
- typedef std::pair< int, String > **IntStringPair**
- typedef std::pair< Real, Real > **RealRealPair**
- typedef std::pair< int, Response > **IntResponsePair**
- typedef std::set< Real > **RealSet**
- typedef std::set< int > **IntSet**
- typedef std::set< size_t > **SizetSet**

- typedef std::vector< RealSet > **RealSetArray**
- typedef std::vector< IntSet > **IntSetArray**
- typedef std::map< int, int > **IntIntMap**
- typedef std::map< int, short > **IntShortMap**
- typedef std::map< int, RealVector > **IntRealVectorMap**
- typedef std::map< int, ActiveSet > **IntActiveSetMap**
- typedef std::map< int, Variables > **IntVariablesMap**
- typedef std::map< int, Response > **IntResponseMap**
- typedef std::map< IntArray, size_t > **IntArraySizetMap**
- typedef std::multimap< RealRealPair, ParamResponsePair > **RealPairPRPMultiMap**
- typedef IntList::iterator **ILIter**
- typedef IntList::const_iterator **ILCIter**
- typedef SizetList::iterator **StLIter**
- typedef SizetList::const_iterator **StLCIter**
- typedef RealList::iterator **RLIter**
- typedef RealList::const_iterator **RLCIter**
- typedef StringList::iterator **StringLIter**
- typedef StringList::const_iterator **StringLCIter**
- typedef VariablesList::iterator **VarsLIter**
- typedef InterfaceList::iterator **InterfLIter**
- typedef ResponseList::iterator **RespLIter**
- typedef ModelList::iterator **ModelLIter**
- typedef IteratorList::iterator **IterLIter**
- typedef std::list< ParallelLevel >::iterator **ParLevLIter**
- typedef std::list< ParallelConfiguration >::iterator **ParConfigLIter**
- typedef IntSet::iterator **ISIter**
- typedef IntSet::const_iterator **ISCIter**
- typedef IntIntMap::iterator **IntIntMIter**
- typedef IntIntMap::const_iterator **IntIntMCIter**
- typedef IntShortMap::iterator **IntShMIter**
- typedef IntRealVectorMap::iterator **IntRDVMIter**
- typedef IntRealVectorMap::const_iterator **IntRDVMCIter**
- typedef IntActiveSetMap::iterator **IntASMIter**
- typedef IntVariablesMap::iterator **IntVarsMIter**
- typedef IntVariablesMap::const_iterator **IntVarsMCIter**
- typedef IntResponseMap::iterator **IntRespMIter**
- typedef IntResponseMap::const_iterator **IntRespMCIter**
- typedef void(∗ **dl_find_optimum_t**)(void ∗, Optimizer1 ∗, char ∗)
- typedef void(∗ **dl_destructor_t**)(void ∗∗)
- typedef int(∗ **ftw_fn**)(const char ∗file, const struct stat ∗, int ftype, int depth, void ∗v)
- typedef struct dirent **dirent**
- typedef struct Dakota::Cbuf **Cbuf**
- typedef struct Dakota::Buf **Buf**
- typedef struct Dakota::Finfo **Finfo**
- typedef Teuchos::SerialDenseSolver< int, Real > **RealSolver**
- typedef Teuchos::SerialSpdDenseSolver< int, Real > **RealSpdSolver**

- typedef int(∗ start_grid_computing_t)(char ∗analysis_driver_script, char ∗params_file, char ∗results_file)

  definition of start grid computing type (function pointer)

- typedef int(∗ perform_analysis_t)(char ∗iteration_num)

  definition of perform analysis type (function pointer)

- typedef int ∗(∗ get_jobs_completed_t)()

  definition of get completed jobs type (function pointer)

- typedef int(∗ stop_grid_computing_t)()

  definition of stop grid computing type (function pointer)

- typedef unsigned char **u_char**
- typedef unsigned short **u_short**
- typedef unsigned int **u_int**
- typedef unsigned long **u_long**
- typedef long long **long_long**
- typedef unsigned long **UL**
- typedef void(∗ **Calcrj**)(int ∗n, int ∗p, Real ∗x, int ∗nf, Real ∗r, int ∗ui, void ∗ur, Vf vf)
- typedef void(∗ **Vf**)()
- typedef int **MPI_Comm**
- typedef void ∗ **MPI_Request**
- typedef bmi::multi_index_container< Dakota::ParamResponsePair, bmi::indexed_by< bmi::ordered_unique< bmi::tag< ordered >, bmi::const_mem_fun< Dakota::ParamResponsePair, const IntStringPair &,&Dakota::ParamResponsePair::eval_interface_ids > >, bmi::hashed_non_unique< bmi::tag< hashed >, bmi::identity< Dakota::ParamResponsePair >, partial_prp_hash, partial_prp_equality > > > PRPMultiIndexCache

  Boost Multi-Index Container for globally caching ParamResponsePairs.

- typedef PRPMultiIndexCache **PRPCache**
- typedef PRPCache::index_iterator< ordered >::type **PRPCacheOIter**
- typedef PRPCache::index_const_iterator< ordered >::type **PRPCacheOCIter**
- typedef PRPCache::index_iterator< hashed >::type **PRPCacheHIter**
- typedef PRPCache::index_const_iterator< hashed >::type **PRPCacheHCIter**
- typedef PRPCacheOIter **PRPCacheIter**
- typedef PRPCacheOCIter **PRPCacheCIter**
- typedef bmi::multi_index_container< Dakota::ParamResponsePair, bmi::indexed_by< bmi::ordered_unique< bmi::tag< ordered >, bmi::const_mem_fun< Dakota::ParamResponsePair, int,&Dakota::ParamResponsePair::eval_id > >, bmi::hashed_non_unique< bmi::tag< hashed >, bmi::identity< Dakota::ParamResponsePair >, partial_prp_hash, partial_prp_equality > > > PRPMultiIndexQueue

  Boost Multi-Index Container for locally queueing ParamResponsePairs.

- typedef PRPMultiIndexQueue **PRPQueue**
- typedef PRPQueue::index_iterator< ordered >::type **PRPQueueOIter**
- typedef PRPQueue::index_const_iterator< ordered >::type **PRPQueueOCIter**
- typedef PRPQueue::index_iterator< hashed >::type **PRPQueueHIter**

- typedef PRPQueue::index_const_iterator< hashed >::type **PRPQueueHCIter**
- typedef PRPQueueOIter **PRPQueueIter**
- typedef PRPQueueOCIter **PRPQueueCIter**

Enumerations

- enum {
  COBYLA, **DIRECT**, **EA**, **MS**,
  PS, **SW** }
- enum { **OBJECTIVE**, **INEQUALITY_CONSTRAINT**, **EQUALITY_CONSTRAINT** }
  define algebraic function types
- enum {
  SILENT_OUTPUT, **QUIET_OUTPUT**, **NORMAL_OUTPUT**, **VERBOSE_OUTPUT**,
  DEBUG_OUTPUT }
- enum { **STD_NORMAL_U**, **STD_UNIFORM_U**, **ASKEY_U**, **EXTENDED_U** }
- enum { **NODAL_INTERPOLANT**, **HIERARCHICAL_INTERPOLANT** }
- enum { **NO_INT_REFINE**, **IS**, **AIS**, **MMAIS** }
- enum { **PROBABILITIES**, **RELIABILITIES**, **GEN_RELIABILITIES** }
- enum {
  NO_EMULATOR, **POLYNOMIAL_CHAOS**, **STOCHASTIC_COLLOCATION**, **GAUSSIAN_-PROCESS**,
  KRIGING }
- enum { **IGNORE_RANKS**, **SET_RANKS**, **GET_RANKS**, **SET_GET_RANKS** }
- enum {
  UNCERTAIN, **UNCERTAIN_UNIFORM**, **ACTIVE**, **ACTIVE_UNIFORM**,
  ALL, **ALL_UNIFORM** }
- enum {
  MV, **AMV_X**, **AMV_U**, **AMV_PLUS_X**,
  AMV_PLUS_U, **TANA_X**, **TANA_U**, **NO_APPROX** }
- enum { **BREITUNG**, **HOHENRACK**, **HONG** }
- enum { **EGRA_X**, **EGRA_U** }
- enum { **ORIGINAL_PRIMARY**, **SINGLE_OBJECTIVE**, **LAGRANGIAN_OBJECTIVE**, **AUGMENTED_LAGRANGIAN_OBJECTIVE** }
- enum { **NO_CONSTRAINTS**, **LINEARIZED_CONSTRAINTS**, **ORIGINAL_CONSTRAINTS** }
- enum { **NO_RELAX**, **HOMOTOPY**, **COMPOSITE_STEP** }
- enum { **PENALTY_MERIT**, **ADAPTIVE_PENALTY_MERIT**, **LAGRANGIAN_MERIT**, **AUGMENTED_LAGRANGIAN_MERIT** }
- enum { **FILTER**, **TR_RATIO** }
- enum { **SCALE_NONE**, **SCALE_VALUE**, **SCALE_LOG** }
- enum { **CDV**, **LINEAR**, **NONLIN**, **FN_LSQ** }
- enum { **DISALLOW**, **TARGET**, **BOUNDS** }
- enum { **DEFAULT_POINTS**, **MINIMUM_POINTS**, **RECOMMENDED_POINTS**, **TOTAL_POINTS** }

define special values for pointsManagement

- enum {

  NO_SURROGATE = 0, **UNCORRECTED_SURROGATE**, **AUTO_CORRECTED_SURROGATE**, **BYPASS_SURROGATE**,

  MODEL_DISCREPANCY }

  define special values for SurrogateModel::responseMode

- enum { **NO_CORRECTION** = 0, **ADDITIVE_CORRECTION**, **MULTIPLICATIVE_CORRECTION**, **COMBINED_CORRECTION** }

  define special values for approxCorrectionType

- enum {

  EMPTY, **MERGED_ALL**, **MIXED_ALL**, **MERGED_DISTINCT_DESIGN**,

  MERGED_DISTINCT_UNCERTAIN, **MERGED_DISTINCT_ALEATORY_UNCERTAIN**, **MERGED_DISTINCT_EPISTEMIC_UNCERTAIN**, **MERGED_DISTINCT_STATE**,

  MIXED_DISTINCT_DESIGN, **MIXED_DISTINCT_UNCERTAIN**, **MIXED_DISTINCT_ALEATORY_UNCERTAIN**, **MIXED_DISTINCT_EPISTEMIC_UNCERTAIN**,

  MIXED_DISTINCT_STATE }
- enum {

  CONTINUOUS_DESIGN, **DISCRETE_DESIGN_RANGE**, **DISCRETE_DESIGN_SET_INT**, **DISCRETE_DESIGN_SET_REAL**,

  NORMAL_UNCERTAIN, **LOGNORMAL_UNCERTAIN**, **UNIFORM_UNCERTAIN**, **LOGUNIFORM_UNCERTAIN**,

  TRIANGULAR_UNCERTAIN, **EXPONENTIAL_UNCERTAIN**, **BETA_UNCERTAIN**, **GAMMA_UNCERTAIN**,

  GUMBEL_UNCERTAIN, **FRECHET_UNCERTAIN**, **WEIBULL_UNCERTAIN**, **HISTOGRAM_BIN_UNCERTAIN**,

  POISSON_UNCERTAIN, **BINOMIAL_UNCERTAIN**, **NEGATIVE_BINOMIAL_UNCERTAIN**, **GEOMETRIC_UNCERTAIN**,

  HYPERGEOMETRIC_UNCERTAIN, **HISTOGRAM_POINT_UNCERTAIN**, **INTERVAL_UNCERTAIN**, **CONTINUOUS_STATE**,

  DISCRETE_STATE_RANGE, **DISCRETE_STATE_SET_INT**, **DISCRETE_STATE_SET_REAL** }
- enum var_t {

  VAR_x1, **VAR_x2**, **VAR_x3**, **VAR_b**,

  VAR_h, **VAR_P**, **VAR_M**, **VAR_Y**,

  VAR_w, **VAR_t**, **VAR_R**, **VAR_E**,

  VAR_X, **VAR_Fs**, **VAR_P1**, **VAR_P2**,

  VAR_P3, **VAR_B**, **VAR_D**, **VAR_H**,

  VAR_F0, **VAR_d** }

  enumeration of possible variable types (to index to names)

- enum driver_t {

  NO_DRIVER = 0, **CANTILEVER_BEAM**, **MOD_CANTILEVER_BEAM**, **CYLINDER_HEAD**,

  EXTENDED_ROSENBROCK, **GENERALIZED_ROSENBROCK**, **LF_ROSENBROCK**, **ROSENBROCK**,

  GERSTNER, **SCALABLE_GERSTNER**, **LOGNORMAL_RATIO**, **MULTIMODAL**,

  PLUGIN_ROSENBROCK, **PLUGIN_TEXT_BOOK**, **SHORT_COLUMN**, **LF_SHORT_COLUMN**,

  SOBOL_RATIONAL, **SOBOL_G_FUNCTION**, **SOBOL_ISHIGAMI**, **STEEL_COLUMN_COST**,

  STEEL_COLUMN_PERFORMANCE, **TEXT_BOOK**, **TEXT_BOOK1**, **TEXT_BOOK2**,

  TEXT_BOOK3, **TEXT_BOOK_OUU**, **SCALABLE_TEXT_BOOK**, **HERBIE**,

  SMOOTH_HERBIE, **SHUBERT**, **SALINAS**, **MODELCENTER**,

  MATLAB, **PYTHON** }

  enumeration of possible driver types (to index to names)

- enum local_data_t { **VARIABLES_MAP** = 1, **VARIABLES_VECTOR** = 2 }

  enumeration for how local variables are stored (values must employ a bit representation)

- enum {

  sFTW_F, **sFTW_SL**, **sFTW_D**, **sFTW_DP**,

  sFTW_DNR, **sFTW_O**, **sFTW_NS** }
- enum {

  sFTWret_OK, **sFTWret_quit**, **sFTWret_skipdir**, **sFTWret_Follow**,

  sFTWret_mallocfailure }
- enum { **SETUP_MODEL**, **SETUP_USERFUNC** }
- enum {

  CAUVar_normal = 0, **CAUVar_lognormal** = 1, **CAUVar_uniform** = 2, **CAUVar_loguniform** = 3,

  CAUVar_triangular = 4, **CAUVar_exponential** = 5, **CAUVar_beta** = 6, **CAUVar_gamma** = 7,

  CAUVar_gumbel = 8, **CAUVar_frechet** = 9, **CAUVar_weibull** = 10, **CAUVar_histogram_bin** = 11,

  CAUVar_Nkinds = 12 }
- enum {

  DAUIVar_poisson = 0, **DAUIVar_binomial** = 1, **DAUIVar_negative_binomial** = 2, **DAUIVar_geometric** = 3,

  DAUIVar_hypergeometric = 4, **DAUIVar_Nkinds** = 5 }
- enum { **DAURVar_histogram_point** = 0, **DAURVar_Nkinds** = 1 }
- enum { **CEUVar_interval** = 0, **CEUVar_Nkinds** = 1 }
- enum {

  DiscSetVar_design_set_int = 0, **DiscSetVar_design_set_real** = 1, **DiscSetVar_state_set_int** = 2, **DiscSetVar_state_set_real** = 3,

  DiscSetVar_Nkinds = 4 }
- enum { **N_VLS** = 4 }

- enum CG_UPDATETYPE {

  CG_STEEPEST, **CG_FLETCHER_REEVES**, **CG_POLAK_RIBIERE**, **CG_POLAK_RIBIERE_-PLUS**,

  CG_HESTENES_STIEFEL }

  NonlinearCG update options.

- enum CG_LINESEARCHTYPE { **CG_FIXED_STEP**, **CG_LS_SIMPLE**, **CG_LS_BRENT**, **CG_LS_-WOLFE** }

  NonlinearCG linesearch options.

- enum {

  LIST = 1, **VECTOR_SV**, **VECTOR_FP**, **CENTERED**,

  MULTIDIM }
- enum { **ESTIMATE_ORDER** = 1, **CONVERGE_ORDER**, **CONVERGE_QOI** }
- enum EvalType { **NLFEvaluator**, **CONEvaluator** }

  enumeration for the type of evaluator function

- enum {

  TH_SILENT_OUTPUT, **TH_QUIET_OUTPUT**, **TH_NORMAL_OUTPUT**, **TH_VERBOSE_-OUTPUT**,

  TH_DEBUG_OUTPUT }

Functions

- static const char ∗ basename (const char ∗s)

  return name of s, stripped of any leading path information

- static void cleanup_and_abort (const std::string &resname)

  output error message about results file and call abort

- CommandShell & flush (CommandShell &shell)

  convenient shell manipulator function to "flush" the shell

- bool operator== (const ActiveSet &set1, const ActiveSet &set2)

  equality operator for ActiveSet

- std::istream & operator>> (std::istream &s, ActiveSet &set)

  std::istream extraction operator for ActiveSet. Calls read(std::istream&).

- std::ostream & operator<< (std::ostream &s, const ActiveSet &set)

  std::ostream insertion operator for ActiveSet. Calls write(std::istream&).

- BiStream & operator>> (BiStream &s, ActiveSet &set)

  BiStream extraction operator for ActiveSet. Calls read(BiStream&).

- BoStream & operator<< (BoStream &s, const ActiveSet &set)

  BoStream insertion operator for ActiveSet. Calls write(BoStream&).

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, ActiveSet &set)

  MPIUnpackBuffer extraction operator for ActiveSet. Calls read(MPIUnpackBuffer&).

- MPIPackBuffer & operator<< (MPIPackBuffer &s, const ActiveSet &set)

  MPIPackBuffer insertion operator for ActiveSet. Calls write(MPIPackBuffer&).

- bool operator!= (const ActiveSet &set1, const ActiveSet &set2)

  inequality operator for ActiveSet

- template<class ArrayT >
  void array_read (ArrayT &v, BiStream &s)

  Read an array from BiStream, s.

- template<class ArrayT >
  void array_write (const ArrayT &v, BoStream &s)

  Write an array to BoStream, s.

- template<class ArrayT >
  BiStream & operator>> (BiStream &s, ArrayT &data)

  global BiStream extraction operator for generic "array" container

- template<class ArrayT >
  BoStream & operator<< (BoStream &s, const ArrayT &data)

  global BoStream insertion operator for generic "array" container

- std::istream & operator>> (std::istream &s, Constraints &con)

  std::istream extraction operator for Constraints

- std::ostream & operator<< (std::ostream &s, const Constraints &con)

  std::ostream insertion operator for Constraints

- bool interface_id_compare (const Interface &interface, const void *id)

  global comparison function for Interface

- bool method_id_compare (const Iterator &iterator, const void *id)

  global comparison function for Iterator

- bool model_id_compare (const Model &model, const void *id)

  global comparison function for Model

- bool operator== (const ResponseRep &rep1, const ResponseRep &rep2)

  equality operator for ResponseRep

- bool responses_id_compare (const Response &resp, const void *id)
  global comparison function for Response
- std::istream & operator>> (std::istream &s, Response &response)
  std::istream extraction operator for Response. Calls read(std::istream&).
- std::ostream & operator<< (std::ostream &s, const Response &response)
  std::ostream insertion operator for Response. Calls write(std::ostream&).
- BiStream & operator>> (BiStream &s, Response &response)
  BiStream extraction operator for Response. Calls read(BiStream&).
- BoStream & operator<< (BoStream &s, const Response &response)
  BoStream insertion operator for Response. Calls write(BoStream&).
- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, Response &response)
  MPIUnpackBuffer extraction operator for Response. Calls read(MPIUnpackBuffer&).
- MPIPackBuffer & operator<< (MPIPackBuffer &s, const Response &response)
  MPIPackBuffer insertion operator for Response. Calls write(MPIPackBuffer&).
- bool operator== (const Response &resp1, const Response &resp2)
  equality operator for Response
- bool operator!= (const Response &resp1, const Response &resp2)
  inequality operator for Response
- std::string re_match (const std::string &token, const boost::regex &re)
  Global utility function to ease migration from CtelRegExp to Boost.Regex.
- MPIPackBuffer & operator<< (MPIPackBuffer &s, const String &data)
  Reads String from buffer.
- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, String &data)
  Writes String to buffer.
- String operator+ (const String &s1, const String &s2)
  Concatenate two Strings and return the resulting String.
- String operator+ (const char *s1, const String &s2)
  Append a String to a char and return the resulting String.*
- String operator+ (const String &s1, const char *s2)
  Append a char to a String and return the resulting String.*

- String operator+ (const std::string &s1, const String &s2)

  Append a String to a std::string and return the resulting String.

- String operator+ (const String &s1, const std::string &s2)

  Append a std::string to a String and return the resulting String.

- String toUpper (const String &str)

  Returns a String converted to upper case. Calls String::upper().

- String toLower (const String &str)

  Returns a String converted to lower case. Calls String::lower().

- bool operator== (const Variables &vars1, const Variables &vars2)

  equality operator for Variables

- bool binary_equal_to (const Variables &vars1, const Variables &vars2)

  binary_equal_to (since 'operator==' is not suitable for boost/hashed lookup)

- std::size_t hash_value (const Variables &vars)

  hash_value for Variables - required by the new BMI hash_set of PRPairs

- bool variables_id_compare (const Variables &vars, const void *id)

  global comparison function for Variables

- std::istream & operator>> (std::istream &s, Variables &vars)

  std::istream extraction operator for Variables.

- std::ostream & operator<< (std::ostream &s, const Variables &vars)

  std::ostream insertion operator for Variables.

- BiStream & operator>> (BiStream &s, Variables &vars)

  BiStream extraction operator for Variables.

- BoStream & operator<< (BoStream &s, const Variables &vars)

  BoStream insertion operator for Variables.

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, Variables &vars)

  MPIUnpackBuffer extraction operator for Variables.

- MPIPackBuffer & operator<< (MPIPackBuffer &s, const Variables &vars)

  MPIPackBuffer insertion operator for Variables.

- bool operator!= (const Variables &vars1, const Variables &vars2)

  inequality operator for Variables

- template<class T >
  std::ostream & operator<< (std::ostream &s, const std::set< T > &data)

  global std::ostream insertion operator for std::set

- template<class T >
  MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, std::set< T > &data)

  global MPIUnpackBuffer extraction operator for std::set

- template<class T >
  MPIPackBuffer & operator<< (MPIPackBuffer &s, const std::set< T > &data)

  global MPIPackBuffer insertion operator for std::set

- template<typename OrdinalType , typename ScalarType >
  MPIPackBuffer & operator<< (MPIPackBuffer &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &data)

  global MPIPackBuffer insertion operator for Teuchos::SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  MPIPackBuffer & operator<< (MPIPackBuffer &s, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &data)

  global MPIPackBuffer insertion operator for Teuchos::SerialDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  MPIPackBuffer & operator<< (MPIPackBuffer &s, const Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &data)

  global MPIPackBuffer insertion operator for Teuchos::SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &data)

  global MPIUnpackBuffer extraction operator for Teuchos::SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &data)

  global MPIUnpackBuffer extraction operator for Teuchos::SerialDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &data)

  global MPIUnpackBuffer extraction operator for Teuchos::SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void read_data (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  standard istream extraction operator for full SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void read_data (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArray &label_array)

  standard istream extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArrayView label_array)

  standard istream extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data_partial (std::istream &s, size_t start_index, size_t num_items, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  standard istream extraction operator for partial SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void read_data_partial (std::istream &s, size_t start_index, size_t num_items, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArray &label_array)

  standard istream extraction operator for partial SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data_partial (std::istream &s, size_t start_index, size_t num_items, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArrayView label_array)

  standard istream extraction operator for partial SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data_tabular (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  tabular istream extraction operator for full SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void read_data_partial_tabular (std::istream &s, size_t start_index, size_t num_items, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  tabular istream extraction operator for partial SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void read_data_annotated (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArray &label_array)

  annotated istream extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data_annotated (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArrayView label_array)

  annotated istream extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const ScalarType *v, OrdinalType len)

standard ostream insertion operator for full SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  standard ostream insertion operator for full SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  standard ostream insertion operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArrayConstView label_array)

  standard ostream insertion operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringArray &label_array)

  standard ostream insertion operator for full SerialDenseVector with alternate labels

- template<typename OrdinalType , typename ScalarType >
  void write_data_aprepro (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  aprepro ostream insertion operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data_partial (std::ostream &s, size_t start_index, size_t num_items, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  standard ostream insertion operator for partial SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void write_data_partial (std::ostream &s, size_t start_index, size_t num_items, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  standard ostream insertion operator for partial SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data_partial_aprepro (std::ostream &s, size_t start_index, size_t num_items, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  aprepro ostream insertion operator for partial SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data_annotated (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  annotated ostream insertion operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data_tabular (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  tabular ostream insertion operator for full SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void write_data_tabular (std::ostream &s, const ScalarType *ptr, OrdinalType num_items)

  tabular ostream insertion operator for full SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void write_data_partial_tabular (std::ostream &s, size_t start_index, size_t num_items, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v)

  tabular ostream insertion operator for partial SerialDenseVector

- void write_data_tabular (std::ostream &s, const StringArray &sa)

  tabular ostream insertion operator for vector of strings

- void write_data_tabular (std::ostream &s, StringMultiArrayConstView ma)

  tabular ostream insertion operator for view of StringMultiArray

- template<typename OrdinalType , typename ScalarType >
  std::istream & operator>> (std::istream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &data)

  global std::istream extraction operator for SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  std::ostream & operator<< (std::ostream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &data)

  global std::ostream insertion operator for SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void read_data (BiStream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArray &label_array)

  standard binary stream extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data (BiStream &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArrayView label_array)

  standard binary stream extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data (BoStream &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  standard binary stream insertion operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data (MPIUnpackBuffer &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArray &label_array)

standard MPI buffer extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data (MPIUnpackBuffer &s, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, StringMultiArrayView label_array)

  standard MPI buffer extraction operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void write_data (MPIPackBuffer &s, const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &v, const StringMultiArray &label_array)

  standard MPI buffer insertion operator for full SerialDenseVector with labels

- template<typename OrdinalType , typename ScalarType >
  void read_data (std::istream &s, std::vector< Teuchos::SerialDenseVector< OrdinalType, ScalarType > > &va)

  standard istream extraction operator for std::vector of SerialDenseVectors

- template<typename OrdinalType , typename ScalarType >
  void read_data (std::istream &s, Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &m)

  standard istream extraction operator for SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void read_data (BiStream &s, Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &m)

  standard binary stream extraction operator for SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void read_data (MPIUnpackBuffer &s, Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &m)

  standard MPI buffer extraction operator for SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &m, bool brackets, bool row_rtn, bool final_rtn)

  formatted ostream insertion operator for SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void write_data (BoStream &s, const Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &m)

  standard binary stream insertion operator for SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void write_data (MPIPackBuffer &s, const Teuchos::SerialSymDenseMatrix< OrdinalType, ScalarType > &m)

  standard MPI buffer insertion operator for SerialSymDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &m, bool brackets, bool row_rtn, bool final_rtn)

formatted ostream insertion operator for SerialDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void write_data (std::ostream &s, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &m, const StringArray &row_labels, const StringArray &col_labels)

  ostream insertion operator for SerialDenseMatrix with row/col labels

- template<typename OrdinalType , typename ScalarType >
  void write_col_vector_trans (std::ostream &s, OrdinalType col, OrdinalType num_items, bool brackets, bool break_line, bool final_rtn, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &sdm)

  ostream insertion operator for a column vector from a SerialDenseMatrix

- template<typename OrdinalType , typename ScalarType >
  void write_col_vector_trans (std::ostream &s, OrdinalType col, bool brackets, bool break_line, bool final_rtn, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &sdm)

  ostream insertion operator for a column vector from a SerialDenseMatrix

- template<typename OStreamType , typename OrdinalType , typename ScalarType >
  void write_col_vector_trans (OStreamType &s, OrdinalType col, OrdinalType num_items, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &sdm)

  ostream insertion operator for a column vector from a SerialDenseMatrix

- template<typename OStreamType , typename OrdinalType , typename ScalarType >
  void write_col_vector_trans (OStreamType &s, OrdinalType col, const Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &sdm)

  ostream insertion operator for a column vector from a SerialDenseMatrix

- template<typename IStreamType , typename OrdinalType , typename ScalarType >
  void read_col_vector_trans (IStreamType &s, OrdinalType col, Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &sdm)

  istream extraction operator for a column vector from a SerialDenseMatrix

- template<class ArrayT >
  void array_read (ArrayT &v, std::istream &s)

  read array from std::istream

- template<class ArrayT >
  void array_write (const ArrayT &v, std::ostream &s)

  write array to std::ostream

- template<class ListT >
  void list_write (const ListT &l, std::ostream &s)

  write list to std::ostream

- template<class T >
  std::istream & operator>> (std::istream &s, std::vector< T > &data)

  global std::istream extraction operator for std::vector

- template<class T >
  std::ostream & operator<< (std::ostream &s, const std::vector< T > &data)

  global std::ostream insertion operator for std::vector

- template<class T >
  std::ostream & operator<< (std::ostream &s, const std::list< T > &data)

  global std::ostream insertion operator for std::list

- template<class ArrayT >
  void array_write (std::ostream &s, const ArrayT &v, const std::vector< String > &label_array)

  write array to std::ostream with labels

- template<class ArrayT >
  void array_write_aprepro (std::ostream &s, const ArrayT &v, const std::vector< String > &label_array)

  write array to std::ostream (APREPRO format)

- template<class ArrayT >
  void array_write_annotated (const ArrayT &v, std::ostream &s, bool write_len)

  Write array to ostream as a row vector; precede with length if write_len = true.

- bool operator== (const ShortArray &dsa1, const ShortArray &dsa2)

  equality operator for ShortArray

- bool operator== (const StringArray &dsa1, const StringArray &dsa2)

  equality operator for StringArray

- bool operator== (const SizetArray &sa, SizetMultiArrayConstView smav)

  equality operator for SizetArray and SizetMultiArrayConstView

- bool operator== (const IntArray &dia1, const IntArray &dia2)

  equality operator for IntArray

- bool operator!= (const IntArray &dia1, const IntArray &dia2)

  inequality operator for IntArray

- bool operator!= (const ShortArray &dsa1, const ShortArray &dsa2)

  inequality operator for ShortArray

- bool operator!= (const StringArray &dsa1, const StringArray &dsa2)

  inequality operator for StringArray

- bool operator!= (const SizetArray &sa, SizetMultiArrayConstView smav)

  inequality operator for StringArray

- void build_label (String &label, const String &root_label, size_t tag)

create a label by appending a numerical tag to the root_label

- void build_labels (StringArray &label_array, const String &root_label)

  create an array of labels by tagging root_label for each entry in label_array. Uses build_label().

- void build_labels (StringMultiArray &label_array, const String &root_label)

  create an array of labels by tagging root_label for each entry in label_array. Uses build_label().

- void build_labels_partial (StringArray &label_array, const String &root_label, size_t start_index, size_t num_items)

  create a partial array of labels by tagging root_label for a subset of entries in label_array. Uses build_label().

- void copy_row_vector (const RealMatrix &m, RealMatrix::ordinalType i, std::vector< Real > &row)

  Copies a row of a Teuchos_SerialDenseMatrix<int,Real> to std::vector<Real>.

- template<class T >
  void copy_data (const std::vector< T > &dbv, T *ptr, const int ptr_len)

  *copy Array<T> to T**

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const std::vector< Teuchos::SerialDenseVector< OrdinalType, ScalarType > > &sdva, ScalarType *ptr, const OrdinalType ptr_len, const String &ptr_type)

  *copy Array<Teuchos::SerialDenseVector<OT,ST> > to ST**

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv, Teuchos::SerialDenseMatrix< OrdinalType, ScalarType > &sdm, OrdinalType nr, OrdinalType nc)

  copy Teuchos::SerialDenseVector<OT,ST> to Teuchos::SerialDenseMatrix<OT,ST>

- template<class T >
  void copy_data (const std::list< T > &dl, std::vector< T > &da)

  copy std::list<T> to std::vector<T>

- template<class T >
  void copy_data (const std::list< T > &dl, std::vector< std::vector< T > > &d2a, size_t num_a, size_t a_len)

  copy std::list<T> to std::vector<std::vector<T> >

- template<class T >
  void copy_data (const std::vector< std::vector< T > > &d2a, std::vector< T > &da)

  copy std::vector<vector<T> > to std::vector<T>(unroll vecOfvecs into vector)

- template<class T >
  void copy_data (const std::map< int, T > &im, std::vector< T > &da)

  copy map<int, T> to std::vector<T> (discard integer keys)

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv1, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv2)

  copy Teuchos::SerialDenseVector<OrdinalType, ScalarType> to same (used in place of operator= when a deep copy of a vector view is needed)

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv, std::vector< ScalarType > &da)

  copy Teuchos::SerialDenseVector<OrdinalType, ScalarType> to std::vector<ScalarType>

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const std::vector< ScalarType > &da, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv)

  copy Array<ScalarType> to Teuchos::SerialDenseVector<OrdinalType, ScalarType>

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const ScalarType *ptr, const OrdinalType ptr_len, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv)

  copy ScalarType to Teuchos::SerialDenseVector<OrdinalType, ScalarType>*

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv, ScalarType *ptr, const OrdinalType ptr_len)

  copy ScalarType to Teuchos::SerialDenseVector<OrdinalType, ScalarType>*

- template<typename OrdinalType , typename ScalarType >
  void copy_data (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv, std::vector< Teuchos::SerialDenseVector< OrdinalType, ScalarType > > &sdva, OrdinalType num_vec, OrdinalType vec_len)

  copy SerialDenseVector<> to Array<SerialDenseVector<> >

- template<typename OrdinalType , typename ScalarType >
  void copy_data_partial (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv1, OrdinalType start_index1, OrdinalType num_items, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv2)

  copy portion of first SerialDenseVector to all of second SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void copy_data_partial (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv1, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv2, OrdinalType start_index2)

  copy all of first SerialDenseVector to portion of second SerialDenseVector

- template<typename OrdinalType , typename ScalarType >
  void copy_data_partial (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv1, OrdinalType start_index1, OrdinalType num_items, Teuchos::SerialDenseVector< OrdinalType, ScalarType > &sdv2, OrdinalType start_index2)

  copy portion of first SerialDenseVector to portion of second SerialDenseVector

- template<class T >
  void copy_data_partial (const std::vector< T > &da1, size_t start_index1, size_t num_items, std::vector< T > &da2)

  copy portion of first Array<T> to all of second Array<T>

- template<class T >
  void copy_data_partial (const std::vector< T > &da1, std::vector< T > &da2, size_t start_index2)

  copy all of first Array<T> to portion of second Array<T>

- template<class T >
  void copy_data_partial (const std::vector< T > &da, boost::multi_array< T, 1 > &bma, size_t start_index_bma)

  copy all of first Array<T> to portion of boost::multi_array<T, 1>

- template<class T >
  void copy_data_partial (const std::vector< T > &da1, size_t start_index1, size_t num_items, std::vector< T > &da2, size_t start_index2)

  copy portion of first Array<T> to portion of second Array<T>

- void merge_data_partial (const IntVector &d_array, RealVector &m_array, size_t start_index_ma)

  aggregate continuous and discrete arrays into a single merged array

- template<typename OrdinalType , typename ScalarType >
  const ScalarType & set_index_to_value (OrdinalType index, const std::set< ScalarType > &values)

  retrieve the set value corresponding to the passed index

- template<typename ScalarType >
  size_t set_value_to_index (const ScalarType &value, const std::set< ScalarType > &values)

  calculate the set index corresponding to the passed value

- template<typename OrdinalType , typename ScalarType >
  void x_y_pairs_to_x_set (const Teuchos::SerialDenseVector< OrdinalType, ScalarType > &xy_pairs, std::set< ScalarType > &x_set)

  convert a SerialDenseVector of head-to-tail (x,y) pairs into a std::set of (x), discarding the y values

- template<typename MultiArrayType , typename DataType >
  size_t find_index (const MultiArrayType &a, const DataType &search_data)

  generic find_index (inactive)

- template<typename MultiArrayType , typename DakArrayType >
  void copy_data (const MultiArrayType &ma, DakArrayType &da)

  generic copy (inactive)

- template<class T >
  size_t find_index (const boost::multi_array< T, 1 > &bma, const T &search_data)

  compute the index of an entry within a boost::multi_array

- size_t find_index (SizetMultiArrayConstView bmacv, size_t search_data)

  compute the index of an entry within a boost::multi_array view

- size_t find_index (StringMultiArrayConstView bmacv, const String &search_data)

  compute the index of an entry within a boost::multi_array view

- template<class ListT >
  ListT::size_type find_index (const ListT &l, const typename ListT::value_type &val)

  compute the index of an entry within a std::list

- void copy_data (SizetMultiArrayConstView ma, SizetArray &da)

  copy boost::multi_array view to Array

- void copy_data (StringMultiArrayConstView ma, StringArray &da)

  copy boost::multi_array view to Array

- template<typename DakContainerType >
  bool contains (const DakContainerType &v, const typename DakContainerType::value_type &val)

  return true if the item val appears in container v

- template<class ListT >
  ListT::size_type count_if (const ListT &l, bool(∗test_fn)(const typename ListT::value_type &, const std::string &), const std::string &test_fn_data)

  count the number of elements in the list satisfying the predicate test_fn w.r.t. the passed test_fn_data

- template<class ListT >
  ListT::const_iterator find_if (const ListT &c, bool(∗test_fn)(const typename ListT::value_type &, const std::string &), const std::string &test_fn_data)

  return an iterator to the first list element satisfying the predicate test_fn w.r.t. the passed test_fn_data; end if not found

- Real rel_change_rv (const RealVector &curr_rv, const RealVector &prev_rv)

  Computes relative change between successive RealVectors using Euclidean norm.

- MPIPackBuffer & operator<< (MPIPackBuffer &s, const DataInterface &data)

  MPIPackBuffer insertion operator for DataInterface.

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, DataInterface &data)

  MPIUnpackBuffer extraction operator for DataInterface.

- std::ostream & operator<< (std::ostream &s, const DataInterface &data)

  std::ostream insertion operator for DataInterface

- MPIPackBuffer & operator<< (MPIPackBuffer &s, const DataMethod &data)

  MPIPackBuffer insertion operator for DataMethod.

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, DataMethod &data)
  MPIUnpackBuffer extraction operator for DataMethod.
- std::ostream & operator<< (std::ostream &s, const DataMethod &data)
  std::ostream insertion operator for DataMethod
- MPIPackBuffer & operator<< (MPIPackBuffer &s, const DataModel &data)
  MPIPackBuffer insertion operator for DataModel.
- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, DataModel &data)
  MPIUnpackBuffer extraction operator for DataModel.
- std::ostream & operator<< (std::ostream &s, const DataModel &data)
  std::ostream insertion operator for DataModel
- MPIPackBuffer & operator<< (MPIPackBuffer &s, const DataResponses &data)
  MPIPackBuffer insertion operator for DataResponses.
- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, DataResponses &data)
  MPIUnpackBuffer extraction operator for DataResponses.
- std::ostream & operator<< (std::ostream &s, const DataResponses &data)
  std::ostream insertion operator for DataResponses
- MPIPackBuffer & operator<< (MPIPackBuffer &s, const DataStrategy &data)
  MPIPackBuffer insertion operator for DataStrategy.
- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, DataStrategy &data)
  MPIUnpackBuffer extraction operator for DataStrategy.
- std::ostream & operator<< (std::ostream &s, const DataStrategy &data)
  std::ostream insertion operator for DataStrategy
- MPIPackBuffer & operator<< (MPIPackBuffer &s, const DataVariables &data)
  MPIPackBuffer insertion operator for DataVariables.
- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, DataVariables &data)
  MPIUnpackBuffer extraction operator for DataVariables.
- std::ostream & operator<< (std::ostream &s, const DataVariables &data)
  std::ostream insertion operator for DataVariables
- int salinas_main (int argc, char *argv[ ], MPI_Comm *comm)
  subroutine interface to SALINAS simulation code
- int **dlsolver_option** (Opt_Info *)

- void abort_handler (int code)

  global function which handles serial or parallel aborts

- RealVector const ∗ **continuous_lower_bounds** (Optimizer1 ∗o)
- RealVector const ∗ **continuous_upper_bounds** (Optimizer1 ∗o)
- RealVector const ∗ **nonlinear_ineq_constraint_lower_bounds** (Optimizer1 ∗o)
- RealVector const ∗ **nonlinear_ineq_constraint_upper_bounds** (Optimizer1 ∗o)
- RealVector const ∗ **nonlinear_eq_constraint_targets** (Optimizer1 ∗o)
- RealVector const ∗ **linear_ineq_constraint_lower_bounds** (Optimizer1 ∗o)
- RealVector const ∗ **linear_ineq_constraint_upper_bounds** (Optimizer1 ∗o)
- RealVector const ∗ **linear_eq_constraint_targets** (Optimizer1 ∗o)
- RealMatrix const ∗ **linear_ineq_constraint_coeffs** (Optimizer1 ∗o)
- RealMatrix const ∗ **linear_eq_constraint_coeffs** (Optimizer1 ∗o)
- void **ComputeResponses** (Optimizer1 ∗o, int mode, int n, double ∗x)
- void **GetFuncs** (Optimizer1 ∗o, int m0, int m1, double ∗f)
- void **GetGrads** (Optimizer1 ∗o, int m0, int m1, int n, int is, int js, double ∗g)
- void **GetContVars** (Optimizer1 ∗o, int n, double ∗x)
- void **SetBestContVars** (Optimizer1 ∗o, int n, double ∗x)
- void **SetBestRespFns** (Optimizer1 ∗o, int n, double ∗x)
- void ∗ **dl_constructor** (Optimizer1 ∗, Dakota_funcs ∗, dl_find_optimum_t ∗, dl_destructor_t ∗)
- static RealVector const ∗ **continuous_lower_bounds1** (Optimizer1 ∗o)
- static RealVector const ∗ **continuous_upper_bounds1** (Optimizer1 ∗o)
- static RealVector const ∗ **nonlinear_ineq_constraint_lower_bounds1** (Optimizer1 ∗o)
- static RealVector const ∗ **nonlinear_ineq_constraint_upper_bounds1** (Optimizer1 ∗o)
- static RealVector const ∗ **nonlinear_eq_constraint_targets1** (Optimizer1 ∗o)
- static RealVector const ∗ **linear_ineq_constraint_lower_bounds1** (Optimizer1 ∗o)
- static RealVector const ∗ **linear_ineq_constraint_upper_bounds1** (Optimizer1 ∗o)
- static RealVector const ∗ **linear_eq_constraint_targets1** (Optimizer1 ∗o)
- static RealMatrix const ∗ **linear_eq_constraint_coeffs1** (Optimizer1 ∗o)
- static RealMatrix const ∗ **linear_ineq_constraint_coeffs1** (Optimizer1 ∗o)
- static void **ComputeResponses1** (Optimizer1 ∗o, int mode, int n, double ∗x)
- static void **GetFuncs1** (Optimizer1 ∗o, int m0, int m1, double ∗f)
- static void **GetGrads1** (Optimizer1 ∗o, int m0, int m1, int n, int is, int js, double ∗g)
- static void **GetContVars1** (Optimizer1 ∗o, int n, double ∗x)
- static void **SetBestContVars1** (Optimizer1 ∗o, int n, double ∗x)
- static void **SetBestDiscVars1** (Optimizer1 ∗o, int n, int ∗x)
- static void **SetBestRespFns1** (Optimizer1 ∗o, int n, double ∗x)
- static double **Get_Real1** (Optimizer1 ∗o, const char ∗name)
- static int **Get_Int1** (Optimizer1 ∗o, const char ∗name)
- static bool **Get_Bool1** (Optimizer1 ∗o, const char ∗name)
- DOTOptimizer ∗ **new_DOTOptimizer** (Model &model)
- DOTOptimizer ∗ **new_DOTOptimizer** (NoDBBaseConstructor, Model &model)
- void **dak_sigcatch** (int sig)
- void start_dakota_heartbeat (int seconds)
- int **sftw** (const char ∗name, int(∗fn)(const char ∗file, const struct stat ∗, int ftype, int depth, void ∗v), void ∗v)

- static int **ftw1** (char ∗name, size_t namelen, size_t namemaxlen, ftw_fn fn, int, void ∗v)
- static int **compar** (const void ∗a, const void ∗b)
- static int **dodir** (DIR ∗dir, char ∗name, size_t namelen, size_t namemaxlen, ftw_fn fn, int depth, void ∗v, struct stat ∗sb)
- int **sftw** (const char ∗name, ftw_fn fn, void ∗v)
- static int **Symlink** (const char ∗from, const char ∗to)
- static int **my_recrm** (const char ∗file, const struct stat ∗sb, int ftype, int depth, void ∗v)
- int **rec_rmdir** (const char ∗name)
- static void **buf_incr** (Buf ∗b, size_t Lt)
- int my_cp (const char ∗file, const struct stat ∗sb, int ftype, int depth, void ∗v)
- int **rec_cp** (const char ∗from, const char ∗todir, int copy, int flatten, int replace)
- static char ∗ **pathsimp** (char ∗t0)
- void get_npath (int appdrive, std::string ∗pnpath)
- void **workdir_adjust** (const std::string &workdir)
- std::string get_cwd ()

  Portability adapter for getcwd.

- void putenv_impl (const char ∗name_and_value)

  Utility function from boost/test, not available in the DAKOTA snapshot.

- static HANDLE ∗ **wait_setup** (std::map< pid_t, int > ∗M, size_t ∗pn)
- static int **wait_for_one** (size_t n, HANDLE ∗h, int req1, size_t ∗pi)
- static void **pid_botch** ()
- template<class ListT >
  void **removeAt** (ListT &l, typename ListT::size_type index)
- Real getdist (const RealVector &x1, const RealVector &x2)
- Real mindist (const RealVector &x, const RealMatrix &xset, int except)
- Real mindistindx (const RealVector &x, const RealMatrix &xset, const IntArray &indx)
- Real getRmax (const RealMatrix &xset)
- ParallelLibrary dummy_lib ("dummy")

  dummy ParallelLibrary object used for < mandatory reference initialization when < a real ParallelLibrary instance is < unavailable

- template<typename T >
  T abort_handler_t (int code)
- int start_grid_computing (char ∗analysis_driver_script, char ∗params_file, char ∗results_file)
- int stop_grid_computing ()
- int perform_analysis (char ∗iteration_num)
- template<typename T >
  string asstring (const T &val)

  Creates a string from the argument val using an ostringstream.

- void run_dakota_data ()

  Function to encapsulate the DAKOTA object instantiations for mode 2: direct Data class instantiation.

- **PACKBUF** (int, MPI_INT)
- **UNPACKBUF** (int, MPI_INT)

- **PACKSIZE** (int, MPI_INT)
- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const int &data)

  insert an int

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const u_int &data)

  insert a u_int

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const long &data)

  insert a long

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const u_long &data)

  insert a u_long

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const short &data)

  insert a short

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const u_short &data)

  insert a u_short

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const char &data)

  insert a char

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const u_char &data)

  insert a u_char

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const double &data)

  insert a double

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const float &data)

  insert a float

- MPIPackBuffer & operator<< (MPIPackBuffer &buff, const bool &data)

  insert a bool

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, int &data)

  extract an int

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, u_int &data)

  extract a u_int

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, long &data)

  extract a long

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, u_long &data)

  extract a u_long

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, short &data)

  extract a short

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, u_short &data)

  extract a u_short

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, char &data)

  extract a char

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, u_char &data)

  extract a u_char

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, double &data)

  extract a double

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, float &data)

  extract a float

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &buff, bool &data)

  extract a bool

- template<class ContainerT >
  void container_read (ContainerT &c, MPIUnpackBuffer &s)

  Read a generic container (vector<T>, list<T>) from MPIUnpackBuffer, s.

- template<class ContainerT >
  void container_write (const ContainerT &c, MPIPackBuffer &s)

  Write a generic container to MPIPackBuffer, s.

- template<class ContainerT >
  MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, ContainerT &data)

  global MPIUnpackBuffer extraction operator for generic container

- template<class ContainerT >
  MPIPackBuffer & operator<< (MPIPackBuffer &s, const ContainerT &data)

  global MPIPackBuffer insertion operator for generic container

- int MPIPackSize (const int &data, const int num=1)

  return packed size of an int

- int MPIPackSize (const u_int &data, const int num=1)

  return packed size of a u_int

- int MPIPackSize (const long &data, const int num=1)

  return packed size of a long

- int MPIPackSize (const u_long &data, const int num=1)

  return packed size of a u_long

- int MPIPackSize (const short &data, const int num=1)

  return packed size of a short

- int MPIPackSize (const u_short &data, const int num=1)

  return packed size of a u_short

- int MPIPackSize (const char &data, const int num=1)

  return packed size of a char

- int MPIPackSize (const u_char &data, const int num=1)

  return packed size of a u_char

- int MPIPackSize (const double &data, const int num=1)

  return packed size of a double

- int MPIPackSize (const float &data, const int num=1)

  return packed size of a float

- int MPIPackSize (const bool &data, const int num=1)

  return packed size of a bool

- int **nidr_parse** (const char ∗, FILE ∗)
- int **not_executable** (const char ∗driver_name, const char ∗tdir)
- static void **scale_chk** (StringArray &ST, RealVector &S, const char ∗what, const char ∗∗univ)
- static void **BuildLabels** (StringArray ∗sa, size_t nsa, size_t n1, size_t n2, const char ∗stub)
- static int **flist_check** (IntList ∗L, int n, IntArray ∗iv, const char ∗what)
- static void **flist_check2** (size_t n, IntArray ∗iv, const char ∗what)
- static int **wronglen** (size_t n, RealVector ∗V, const char ∗what)
- static int **wronglen2** (size_t n, IntVector ∗V, const char ∗what)
- static void **Vcopyup** (RealVector ∗V, RealVector ∗M, size_t i, size_t n)
- static void **Set_rdv** (RealVector ∗V, double d, size_t n)
- static void **Vadj_Normal** (DataVariablesRep ∗dv, size_t i0, Var_Info ∗vi)
- static void **Vbgen_normalUnc** (DataVariablesRep ∗dv, size_t i0)
- static void **Vadj_Lognormal** (DataVariablesRep ∗dv, size_t i0, Var_Info ∗vi)
- static void **Vbgen_lognormalUnc** (DataVariablesRep ∗dv, size_t i0)
- static void **Vadj_Uniform** (DataVariablesRep ∗dv, size_t i0, Var_Info ∗vi)
- static void **Vbgen_uniformUnc** (DataVariablesRep ∗dv, size_t i0)
- static void **Vadj_Loguniform** (DataVariablesRep ∗dv, size_t i0, Var_Info ∗vi)
- static void **Vbgen_loguniformUnc** (DataVariablesRep ∗dv, size_t i0)
- static void **Vadj_Triangular** (DataVariablesRep ∗dv, size_t i0, Var_Info ∗vi)
- static void **Vbgen_triangularUnc** (DataVariablesRep ∗dv, size_t i0)
- static void **Vadj_Exponential** (DataVariablesRep ∗dv, size_t i0, Var_Info ∗vi)
- static void **Vbgen_Exponential** (DataVariablesRep ∗dv, size_t i0)

- static void **Vadj_Beta** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_betaUnc** (DataVariablesRep *dv, size_t i0)
- static void **Vadj_Gamma** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_Gamma** (DataVariablesRep *dv, size_t i0)
- static void **Vadj_Gumbel** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_Gumbel** (DataVariablesRep *dv, size_t i0)
- static void **Vadj_Frechet** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_Frechet** (DataVariablesRep *dv, size_t i0)
- static void **Vadj_Weibull** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_Weibull** (DataVariablesRep *dv, size_t i0)
- static void **Vadj_HistogramBin** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_Poisson** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_Binomial** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_NegBinomial** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_Geometric** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_HyperGeom** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_HistogramPt** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_HistogramBin** (DataVariablesRep *dv, size_t i0)
- static void **Vbgen_HistogramPt** (DataVariablesRep *dv, size_t i0)
- static void **Vadj_Interval** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vbgen_Interval** (DataVariablesRep *dv, size_t i0)
- static void **Vbgen_Poisson** (DataVariablesRep *dv, size_t i0)
- static void **Vbgen_Binomial** (DataVariablesRep *dv, size_t i0)
- static void **Vbgen_NegBinomial** (DataVariablesRep *dv, size_t i0)
- static void **Vbgen_Geometric** (DataVariablesRep *dv, size_t i0)
- static void **Vbgen_HyperGeom** (DataVariablesRep *dv, size_t i0)
- static void **DIset** (size_t n, IntSetArray *a, IntVector *L, IntVector *U, IntVector *V)
- static void **DRset** (size_t n, RealSetArray *a, RealVector *L, RealVector *U, RealVector *V)
- static void **Vbgen_DDSI** (DataVariablesRep *dv, size_t n)
- static void **Vbgen_DDSR** (DataVariablesRep *dv, size_t n)
- static void **Vbgen_DSSI** (DataVariablesRep *dv, size_t n)
- static void **Vbgen_DSSR** (DataVariablesRep *dv, size_t n)
- static void **not_div** (const char *kind, size_t nsv, size_t m)
- static void **wrong_number** (const char *what, const char *kind, size_t nsv, size_t m)
- static void **too_small** (const char *kind)
- static void **suppressed** (const char *kind, int ndup, int *ip, Real *rp)
- static void **bad_initial_ivalue** (const char *kind, int val)
- static void **bad_initial_rvalue** (const char *kind, Real val)
- static void **Vadj_DiscreteDesSetReal** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_DiscreteDesSetInt** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_DiscreteStateSetReal** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Vadj_DiscreteStateSetInt** (DataVariablesRep *dv, size_t i0, Var_Info *vi)
- static void **Rdv_copy** (RealVector **prdv, RealVectorArray *rdva)
- static void **var_iulbl** (const char *keyname, Values *val, VarLabel *vl)
- static Iface_mp_Rlit **MP3** (failAction, recoveryFnVals, recover)
- static Iface_mp_ilit **MP3** (failAction, retryLimit, retry)

- static Iface_mp_lit **MP2** (analysisScheduling, self)
- static Iface_mp_lit **MP2** (analysisScheduling, static)
- static Iface_mp_lit **MP2** (evalScheduling, self)
- static Iface_mp_lit **MP2** (evalScheduling, static)
- static Iface_mp_lit **MP2** (failAction, abort)
- static Iface_mp_lit **MP2** (failAction, continuation)
- static Iface_mp_lit **MP2** (interfaceSynchronization, asynchronous)
- static Iface_mp_lit **MP2** (interfaceType, direct)
- static Iface_mp_lit **MP2** (interfaceType, fork)
- static Iface_mp_lit **MP2** (interfaceType, grid)
- static Iface_mp_lit **MP2** (interfaceType, system)
- static Iface_mp_lit **MP2** (asynchLocalEvalScheduling, self)
- static Iface_mp_lit **MP2** (asynchLocalEvalScheduling, static)
- static String **MP_** (idInterface)
- static String **MP_** (inputFilter)
- static String **MP_** (outputFilter)
- static String **MP_** (parametersFile)
- static String **MP_** (resultsFile)
- static String **MP_** (templateDir)
- static String **MP_** (workDir)
- static String2DArray **MP_** (analysisComponents)
- static StringArray **MP_** (analysisDrivers)
- static StringArray **MP_** (templateFiles)
- static bool **MP_** (activeSetVectorFlag)
- static bool **MP_** (allowExistingResultsFlag)
- static bool **MP_** (apreproFlag)
- static bool **MP_** (dirSave)
- static bool **MP_** (dirTag)
- static bool **MP_** (evalCacheFlag)
- static bool **MP_** (fileSaveFlag)
- static bool **MP_** (fileTagFlag)
- static bool **MP_** (restartFileFlag)
- static bool **MP_** (templateCopy)
- static bool **MP_** (templateReplace)
- static bool **MP_** (useWorkdir)
- static bool **MP_** (verbatimFlag)
- static int **MP_** (analysisServers)
- static int **MP_** (asynchLocalAnalysisConcurrency)
- static int **MP_** (asynchLocalEvalConcurrency)
- static int **MP_** (evalServers)
- static int **MP_** (procsPerAnalysis)
- static IntVector **MP_** (primeBase)
- static IntVector **MP_** (sequenceLeap)
- static IntVector **MP_** (sequenceStart)
- static IntVector **MP_** (stepsPerVariable)
- static Method_mp_ilit2 **MP3** (replacementType, numberRetained, chc)

- static Method_mp_ilit2 **MP3** (replacementType, numberRetained, elitist)
- static Method_mp_ilit2 **MP3** (replacementType, numberRetained, random)
- static Method_mp_ilit2z **MP3** (crossoverType, numCrossPoints, multi_point_binary)
- static Method_mp_ilit2z **MP3** (crossoverType, numCrossPoints, multi_point_parameterized_binary)
- static Method_mp_ilit2z **MP3** (crossoverType, numCrossPoints, multi_point_real)
- static Method_mp_lit **MP2** (boxDivision, all_dimensions)
- static Method_mp_lit **MP2** (boxDivision, major_dimension)
- static Method_mp_lit **MP2** (collocPtReuse, all)
- static Method_mp_lit **MP2** (convergenceType, average_fitness_tracker)
- static Method_mp_lit **MP2** (convergenceType, best_fitness_tracker)
- static Method_mp_lit **MP2** (convergenceType, metric_tracker)
- static Method_mp_lit **MP2** (crossoverType, blend)
- static Method_mp_lit **MP2** (crossoverType, two_point)
- static Method_mp_lit **MP2** (crossoverType, uniform)
- static Method_mp_lit **MP2** (distributionType, complementary)
- static Method_mp_lit **MP2** (distributionType, cumulative)
- static Method_mp_lit **MP2** (evalSynchronization, blocking)
- static Method_mp_lit **MP2** (evalSynchronization, nonblocking)
- static Method_mp_lit **MP2** (evalSynchronize, blocking)
- static Method_mp_lit **MP2** (evalSynchronize, nonblocking)
- static Method_mp_lit **MP2** (expansionSampleType, incremental_lhs)
- static Method_mp_lit **MP2** (exploratoryMoves, adaptive)
- static Method_mp_lit **MP2** (exploratoryMoves, multi_step)
- static Method_mp_lit **MP2** (exploratoryMoves, simple)
- static Method_mp_lit **MP2** (fitnessType, domination_count)
- static Method_mp_lit **MP2** (fitnessType, layer_rank)
- static Method_mp_lit **MP2** (fitnessType, linear_rank)
- static Method_mp_lit **MP2** (fitnessType, merit_function)
- static Method_mp_lit **MP2** (fitnessType, proportional)
- static Method_mp_lit **MP2** (initializationType, random)
- static Method_mp_lit **MP2** (initializationType, unique_random)
- static Method_mp_lit **MP2** (integrationRefine, ais)
- static Method_mp_lit **MP2** (integrationRefine, is)
- static Method_mp_lit **MP2** (integrationRefine, mmais)
- static Method_mp_lit **MP2** (meritFunction, merit_max)
- static Method_mp_lit **MP2** (meritFunction, merit_max_smooth)
- static Method_mp_lit **MP2** (meritFunction, merit1)
- static Method_mp_lit **MP2** (meritFunction, merit1_smooth)
- static Method_mp_lit **MP2** (meritFunction, merit2)
- static Method_mp_lit **MP2** (meritFunction, merit2_smooth)
- static Method_mp_lit **MP2** (meritFunction, merit2_squared)
- static Method_mp_lit **MP2** (methodName, asynch_pattern_search)
- static Method_mp_lit **MP2** (methodName, coliny_cobyla)
- static Method_mp_lit **MP2** (methodName, coliny_direct)
- static Method_mp_lit **MP2** (methodName, coliny_pattern_search)
- static Method_mp_lit **MP2** (methodName, coliny_solis_wets)

- static Method_mp_lit **MP2** (methodName, conmin_frcg)
- static Method_mp_lit **MP2** (methodName, conmin_mfd)
- static Method_mp_lit **MP2** (methodName, dace)
- static Method_mp_lit **MP2** (methodName, dot_bfgs)
- static Method_mp_lit **MP2** (methodName, dot_frcg)
- static Method_mp_lit **MP2** (methodName, dot_mmfd)
- static Method_mp_lit **MP2** (methodName, dot_slp)
- static Method_mp_lit **MP2** (methodName, dot_sqp)
- static Method_mp_lit **MP2** (methodName, efficient_global)
- static Method_mp_lit **MP2** (methodName, fsu_cvt)
- static Method_mp_lit **MP2** (methodName, fsu_halton)
- static Method_mp_lit **MP2** (methodName, fsu_hammersley)
- static Method_mp_lit **MP2** (methodName, ncsu_direct)
- static Method_mp_lit **MP2** (methodName, nl2sol)
- static Method_mp_lit **MP2** (methodName, nlpql_sqp)
- static Method_mp_lit **MP2** (methodName, nlssol_sqp)
- static Method_mp_lit **MP2** (methodName, nond_bayes_calibration)
- static Method_mp_lit **MP2** (methodName, nond_global_evidence)
- static Method_mp_lit **MP2** (methodName, nond_global_interval_est)
- static Method_mp_lit **MP2** (methodName, nond_global_reliability)
- static Method_mp_lit **MP2** (methodName, nond_importance_sampling)
- static Method_mp_lit **MP2** (methodName, nond_local_evidence)
- static Method_mp_lit **MP2** (methodName, nond_local_interval_est)
- static Method_mp_lit **MP2** (methodName, nond_polynomial_chaos)
- static Method_mp_lit **MP2** (methodName, nond_sampling)
- static Method_mp_lit **MP2** (methodName, nond_stoch_collocation)
- static Method_mp_lit **MP2** (methodName, nonlinear_cg)
- static Method_mp_lit **MP2** (methodName, npsol_sqp)
- static Method_mp_lit **MP2** (methodName, optpp_cg)
- static Method_mp_lit **MP2** (methodName, optpp_fd_newton)
- static Method_mp_lit **MP2** (methodName, optpp_g_newton)
- static Method_mp_lit **MP2** (methodName, optpp_newton)
- static Method_mp_lit **MP2** (methodName, optpp_pds)
- static Method_mp_lit **MP2** (methodName, optpp_q_newton)
- static Method_mp_lit **MP2** (methodName, psuade_moat)
- static Method_mp_lit **MP2** (methodName, richardson_extrap)
- static Method_mp_lit **MP2** (methodName, surrogate_based_global)
- static Method_mp_lit **MP2** (methodName, surrogate_based_local)
- static Method_mp_lit **MP2** (methodName, vector_parameter_study)
- static Method_mp_lit **MP2** (methodName, list_parameter_study)
- static Method_mp_lit **MP2** (methodName, centered_parameter_study)
- static Method_mp_lit **MP2** (methodName, multidim_parameter_study)
- static Method_mp_lit **MP2** (metropolisType, adaptive)
- static Method_mp_lit **MP2** (metropolisType, hastings)
- static Method_mp_lit **MP2** (minMaxType, maximize)
- static Method_mp_lit **MP2** (minMaxType, minimize)

- static Method_mp_lit **MP2** (mutationType, bit_random)
- static Method_mp_lit **MP2** (mutationType, offset_cauchy)
- static Method_mp_lit **MP2** (mutationType, offset_normal)
- static Method_mp_lit **MP2** (mutationType, offset_uniform)
- static Method_mp_lit **MP2** (mutationType, replace_uniform)
- static Method_mp_lit **MP2** (nondOptAlgorithm, nip)
- static Method_mp_lit **MP2** (nondOptAlgorithm, sqp)
- static Method_mp_lit **MP2** (nondOptAlgorithm, lhs)
- static Method_mp_lit **MP2** (nondOptAlgorithm, ego)
- static Method_mp_lit **MP2** (patternBasis, coordinate)
- static Method_mp_lit **MP2** (patternBasis, simplex)
- static Method_mp_lit **MP2** (rejectionType, standard)
- static Method_mp_lit **MP2** (rejectionType, delayed)
- static Method_mp_lit **MP2** (reliabilityIntegration, first_order)
- static Method_mp_lit **MP2** (reliabilityIntegration, second_order)
- static Method_mp_lit **MP2** (reliabilitySearchType, amv_plus_u)
- static Method_mp_lit **MP2** (reliabilitySearchType, amv_plus_x)
- static Method_mp_lit **MP2** (reliabilitySearchType, amv_u)
- static Method_mp_lit **MP2** (reliabilitySearchType, amv_x)
- static Method_mp_lit **MP2** (reliabilitySearchType, egra_u)
- static Method_mp_lit **MP2** (reliabilitySearchType, egra_x)
- static Method_mp_lit **MP2** (reliabilitySearchType, no_approx)
- static Method_mp_lit **MP2** (reliabilitySearchType, tana_u)
- static Method_mp_lit **MP2** (reliabilitySearchType, tana_x)
- static Method_mp_lit **MP2** (replacementType, elitist)
- static Method_mp_lit **MP2** (replacementType, favor_feasible)
- static Method_mp_lit **MP2** (replacementType, roulette_wheel)
- static Method_mp_lit **MP2** (replacementType, unique_roulette_wheel)
- static Method_mp_lit **MP2** (responseLevelMappingType, gen_reliabilities)
- static Method_mp_lit **MP2** (responseLevelMappingType, probabilities)
- static Method_mp_lit **MP2** (responseLevelMappingType, reliabilities)
- static Method_mp_lit **MP2** (rngName, mt19937)
- static Method_mp_lit **MP2** (rngName, rnum2)
- static Method_mp_lit **MP2** (sampleType, incremental_lhs)
- static Method_mp_lit **MP2** (sampleType, incremental_random)
- static Method_mp_lit **MP2** (sampleType, lhs)
- static Method_mp_lit **MP2** (sampleType, random)
- static Method_mp_lit **MP2** (searchMethod, gradient_based_line_search)
- static Method_mp_lit **MP2** (searchMethod, tr_pds)
- static Method_mp_lit **MP2** (searchMethod, trust_region)
- static Method_mp_lit **MP2** (searchMethod, value_based_line_search)
- static Method_mp_lit **MP2** (subMethodName, box_behnken)
- static Method_mp_lit **MP2** (subMethodName, central_composite)
- static Method_mp_lit **MP2** (subMethodName, gpmsa)
- static Method_mp_lit **MP2** (subMethodName, grid)
- static Method_mp_lit **MP2** (subMethodName, lhs)

- static Method_mp_lit **MP2** (subMethodName, oa_lhs)
- static Method_mp_lit **MP2** (subMethodName, oas)
- static Method_mp_lit **MP2** (subMethodName, queso)
- static Method_mp_lit **MP2** (subMethodName, random)
- static Method_mp_lit **MP2** (subMethodName, converge_order)
- static Method_mp_lit **MP2** (subMethodName, converge_qoi)
- static Method_mp_lit **MP2** (subMethodName, estimate_order)
- static Method_mp_lit **MP2** (trialType, grid)
- static Method_mp_lit **MP2** (trialType, halton)
- static Method_mp_lit **MP2** (trialType, random)
- static Method_mp_lit2 **MP4** (methodName, reliabilitySearchType, nond_local_reliability,"mv")
- static Method_mp_litc **MP3** (crossoverType, crossoverRate, shuffle_random)
- static Method_mp_litc **MP3** (crossoverType, crossoverRate, null_crossover)
- static Method_mp_litc **MP3** (mutationType, mutationRate, null_mutation)
- static Method_mp_litc **MP3** (mutationType, mutationRate, offset_cauchy)
- static Method_mp_litc **MP3** (mutationType, mutationRate, offset_normal)
- static Method_mp_litc **MP3** (mutationType, mutationRate, offset_uniform)
- static Method_mp_litc **MP3** (replacementType, fitnessLimit, below_limit)
- static Method_mp_litrv **MP3** (nichingType, nicheVector, distance)
- static Method_mp_litrv **MP3** (nichingType, nicheVector, radial)
- static Method_mp_litrv **MP3** (postProcessorType, distanceVector, distance_postprocessor)
- static Method_mp_slit2 **MP3** (initializationType, flatFile, flat_file)
- static Method_mp_slit2 **MP3** (methodName, dlDetails, dl_solver)
- static Real **MP_** (absConvTol)
- static Real **MP_** (centeringParam)
- static Real **MP_** (collocationRatio)
- static Real **MP_** (collocRatioTermsOrder)
- static Real **MP_** (constraintPenalty)
- static Real **MP_** (constrPenalty)
- static Real **MP_** (constraintTolerance)
- static Real **MP_** (contractFactor)
- static Real **MP_** (contractStepLength)
- static Real **MP_** (convergenceTolerance)
- static Real **MP_** (crossoverRate)
- static Real **MP_** (falseConvTol)
- static Real **MP_** (functionPrecision)
- static Real **MP_** (globalBalanceParam)
- static Real **MP_** (gradientTolerance)
- static Real **MP_** (initDelta)
- static Real **MP_** (initStepLength)
- static Real **MP_** (initTRRadius)
- static Real **MP_** (likelihoodScale)
- static Real **MP_** (lineSearchTolerance)
- static Real **MP_** (localBalanceParam)
- static Real **MP_** (maxBoxSize)
- static Real **MP_** (maxStep)

- static Real **MP_** (minBoxSize)
- static Real **MP_** (mutationRate)
- static Real **MP_** (mutationScale)
- static Real **MP_** (proposalCovScale)
- static Real **MP_** (refinementRate)
- static Real **MP_** (shrinkagePercent)
- static Real **MP_** (singConvTol)
- static Real **MP_** (singRadius)
- static Real **MP_** (smoothFactor)
- static Real **MP_** (solnTarget)
- static Real **MP_** (stepLenToBoundary)
- static Real **MP_** (surrBasedLocalTRContract)
- static Real **MP_** (surrBasedLocalTRContractTrigger)
- static Real **MP_** (surrBasedLocalTRExpand)
- static Real **MP_** (surrBasedLocalTRExpandTrigger)
- static Real **MP_** (surrBasedLocalTRInitSize)
- static Real **MP_** (surrBasedLocalTRMinSize)
- static Real **MP_** (threshDelta)
- static Real **MP_** (threshStepLength)
- static Real **MP_** (vbdDropTolerance)
- static Real **MP_** (volBoxSize)
- static Real **MP_** (xConvTol)
- static RealVector **MP_** (anisoGridDimPref)
- static RealVector **MP_** (finalPoint)
- static RealVector **MP_** (linearEqConstraintCoeffs)
- static RealVector **MP_** (linearEqScales)
- static RealVector **MP_** (linearEqTargets)
- static RealVector **MP_** (linearIneqConstraintCoeffs)
- static RealVector **MP_** (linearIneqLowerBnds)
- static RealVector **MP_** (linearIneqUpperBnds)
- static RealVector **MP_** (linearIneqScales)
- static RealVector **MP_** (listOfPoints)
- static RealVector **MP_** (stepVector)
- static RealVectorArray **MP_** (genReliabilityLevels)
- static RealVectorArray **MP_** (probabilityLevels)
- static RealVectorArray **MP_** (reliabilityLevels)
- static RealVectorArray **MP_** (responseLevels)
- static unsigned short **MP_** (cubIntOrder)
- static UShortArray **MP_** (expansionOrder)
- static UShortArray **MP_** (quadratureOrder)
- static UShortArray **MP_** (sparseGridLevel)
- static UShortArray **MP_** (varPartitions)
- static String **MP_** (centralPath)
- static String **MP_** (expansionImportFile)
- static String **MP_** (idMethod)
- static String **MP_** (logFile)

- static String **MP_** (meritFn)
- static String **MP_** (modelPointer)
- static String **MP_** (subMethodName)
- static String **MP_** (subMethodPointer)
- static StringArray **MP_** (linearEqScaleTypes)
- static StringArray **MP_** (linearIneqScaleTypes)
- static StringArray **MP_** (miscOptions)
- static bool **MP_** (allVarsFlag)
- static bool **MP_** (constantPenalty)
- static bool **MP_** (expansionFlag)
- static bool **MP_** (fixedSeedFlag)
- static bool **MP_** (fixedSequenceFlag)
- static bool **MP_** (latinizeFlag)
- static bool **MP_** (mainEffectsFlag)
- static bool **MP_** (methodScaling)
- static bool **MP_** (methodUseDerivsFlag)
- static bool **MP_** (mutationAdaptive)
- static bool **MP_** (printPopFlag)
- static bool **MP_** (probCollocFlag)
- static bool **MP_** (randomizeOrderFlag)
- static bool **MP_** (regressDiag)
- static bool **MP_** (showMiscOptions)
- static bool **MP_** (speculativeFlag)
- static bool **MP_** (surrBasedGlobalReplacePts)
- static bool **MP_** (surrBasedLocalLayerBypass)
- static bool **MP_** (vbdFlag)
- static bool **MP_** (volQualityFlag)
- static short **MP_** (expansionType)
- static short **MP_** (nestingOverride)
- static short **MP_** (refinementType)
- static int **MP_** (collocationPoints)
- static int **MP_** (contractAfterFail)
- static int **MP_** (covarianceType)
- static int **MP_** (emulatorSamples)
- static int **MP_** (expandAfterSuccess)
- static int **MP_** (expansionSamples)
- static int **MP_** (expansionTerms)
- static int **MP_** (maxFunctionEvaluations)
- static int **MP_** (maxIterations)
- static int **MP_** (mutationRange)
- static int **MP_** (newSolnsGenerated)
- static int **MP_** (numSamples)
- static int **MP_** (numSteps)
- static int **MP_** (numSymbols)
- static int **MP_** (numTrials)
- static int **MP_** (populationSize)

- static int **MP_** (previousSamples)
- static int **MP_** (randomSeed)
- static int **MP_** (searchSchemeSize)
- static int **MP_** (surrBasedLocalSoftConvLimit)
- static int **MP_** (totalPatternSize)
- static int **MP_** (verifyLevel)
- static size_t **MP_** (numFinalSolutions)
- static size_t **MP_** (numGenerations)
- static size_t **MP_** (numOffspring)
- static size_t **MP_** (numParents)
- static Method_mp_type **MP2s** (emulatorType, GAUSSIAN_PROCESS)
- static Method_mp_type **MP2s** (emulatorType, KRIGING)
- static Method_mp_type **MP2s** (emulatorType, POLYNOMIAL_CHAOS)
- static Method_mp_type **MP2s** (emulatorType, STOCHASTIC_COLLOCATION)
- static Method_mp_type **MP2s** (expansionType, ASKEY_U)
- static Method_mp_type **MP2s** (expansionType, STD_NORMAL_U)
- static Method_mp_type **MP2p** (growthOverride, RESTRICTED)
- static Method_mp_type **MP2p** (growthOverride, UNRESTRICTED)
- static Method_mp_type **MP2s** (methodOutput, DEBUG_OUTPUT)
- static Method_mp_type **MP2s** (methodOutput, NORMAL_OUTPUT)
- static Method_mp_type **MP2s** (methodOutput, QUIET_OUTPUT)
- static Method_mp_type **MP2s** (methodOutput, SILENT_OUTPUT)
- static Method_mp_type **MP2s** (methodOutput, VERBOSE_OUTPUT)
- static Method_mp_type **MP2p** (nestingOverride, NESTED)
- static Method_mp_type **MP2p** (nestingOverride, NON_NESTED)
- static Method_mp_type **MP2s** (piecewiseBasisType, NODAL_INTERPOLANT)
- static Method_mp_type **MP2s** (piecewiseBasisType, HIERARCHICAL_INTERPOLANT)
- static Method_mp_type **MP2p** (refinementControl, DIMENSION_ADAPTIVE_GENERALIZED_-SPARSE)
- static Method_mp_type **MP2p** (refinementControl, DIMENSION_ADAPTIVE_SPECTRAL_DECAY)
- static Method_mp_type **MP2p** (refinementControl, DIMENSION_ADAPTIVE_TOTAL_SOBOL)
- static Method_mp_type **MP2p** (refinementControl, UNIFORM_CONTROL)
- static Method_mp_type **MP2p** (refinementType, P_REFINEMENT)
- static Method_mp_type **MP2p** (refinementType, H_REFINEMENT)
- static Method_mp_type **MP2s** (surrBasedLocalAcceptLogic, FILTER)
- static Method_mp_type **MP2s** (surrBasedLocalAcceptLogic, TR_RATIO)
- static Method_mp_type **MP2s** (surrBasedLocalConstrRelax, HOMOTOPY)
- static Method_mp_type **MP2s** (surrBasedLocalMeritFn, ADAPTIVE_PENALTY_MERIT)
- static Method_mp_type **MP2s** (surrBasedLocalMeritFn, AUGMENTED_LAGRANGIAN_MERIT)
- static Method_mp_type **MP2s** (surrBasedLocalMeritFn, LAGRANGIAN_MERIT)
- static Method_mp_type **MP2s** (surrBasedLocalMeritFn, PENALTY_MERIT)
- static Method_mp_type **MP2s** (surrBasedLocalSubProbCon, LINEARIZED_CONSTRAINTS)
- static Method_mp_type **MP2s** (surrBasedLocalSubProbCon, NO_CONSTRAINTS)
- static Method_mp_type **MP2s** (surrBasedLocalSubProbCon, ORIGINAL_CONSTRAINTS)
- static Method_mp_type **MP2s** (surrBasedLocalSubProbObj, AUGMENTED_LAGRANGIAN_-OBJECTIVE)

- static Method_mp_type **MP2s** (surrBasedLocalSubProbObj, LAGRANGIAN_OBJECTIVE)
- static Method_mp_type **MP2s** (surrBasedLocalSubProbObj, ORIGINAL_PRIMARY)
- static Method_mp_type **MP2s** (surrBasedLocalSubProbObj, SINGLE_OBJECTIVE)
- static Method_mp_type **MP2p** (vbdControl, UNIVARIATE_VBD)
- static IntSet **MP_** (surrogateFnIndices)
- static Model_mp_lit **MP2** (approxPointReuse, all)
- static Model_mp_lit **MP2** (approxPointReuse, none)
- static Model_mp_lit **MP2** (approxPointReuse, region)
- static Model_mp_lit **MP2** (marsInterpolation, linear)
- static Model_mp_lit **MP2** (marsInterpolation, cubic)
- static Model_mp_lit **MP2** (modelType, nested)
- static Model_mp_lit **MP2** (modelType, single)
- static Model_mp_lit **MP2** (modelType, surrogate)
- static Model_mp_lit **MP2** (surrogateType, hierarchical)
- static Model_mp_lit **MP2** (surrogateType, global_gaussian)
- static Model_mp_lit **MP2** (surrogateType, global_kriging)
- static Model_mp_lit **MP2** (surrogateType, global_mars)
- static Model_mp_lit **MP2** (surrogateType, global_moving_least_squares)
- static Model_mp_lit **MP2** (surrogateType, global_neural_network)
- static Model_mp_lit **MP2** (surrogateType, global_polynomial)
- static Model_mp_lit **MP2** (surrogateType, global_radial_basis)
- static Model_mp_lit **MP2** (surrogateType, local_taylor)
- static Model_mp_lit **MP2** (surrogateType, multipoint_tana)
- static Model_mp_lit **MP2** (trendOrder, constant)
- static Model_mp_lit **MP2** (trendOrder, linear)
- static Model_mp_lit **MP2** (trendOrder, reduced_quadratic)
- static Model_mp_lit **MP2** (trendOrder, quadratic)
- static Model_mp_ord **MP2s** (approxCorrectionOrder, 0)
- static Model_mp_ord **MP2s** (approxCorrectionOrder, 1)
- static Model_mp_ord **MP2s** (approxCorrectionOrder, 2)
- static Model_mp_ord **MP2s** (polynomialOrder, 1)
- static Model_mp_ord **MP2s** (polynomialOrder, 2)
- static Model_mp_ord **MP2s** (polynomialOrder, 3)
- static Model_mp_type **MP2s** (approxCorrectionType, ADDITIVE_CORRECTION)
- static Model_mp_type **MP2s** (approxCorrectionType, COMBINED_CORRECTION)
- static Model_mp_type **MP2s** (approxCorrectionType, MULTIPLICATIVE_CORRECTION)
- static Model_mp_type **MP2s** (pointsManagement, MINIMUM_POINTS)
- static Model_mp_type **MP2s** (pointsManagement, RECOMMENDED_POINTS)
- static Real **MP_** (annRange)
- static RealVector **MP_** (krigingCorrelations)
- static RealVector **MP_** (primaryRespCoeffs)
- static RealVector **MP_** (secondaryRespCoeffs)
- static String **MP_** (approxPointReuseFile)
- static String **MP_** (idModel)
- static String **MP_** (interfacePointer)
- static String **MP_** (krigingOptMethod)

- static String **MP_** (lowFidelityModelPointer)
- static String **MP_** (optionalInterfRespPointer)
- static String **MP_** (responsesPointer)
- static String **MP_** (truthModelPointer)
- static String **MP_** (variablesPointer)
- static StringArray **MP_** (primaryVarMaps)
- static StringArray **MP_** (secondaryVarMaps)
- static StringArray **MP_** (diagMetrics)
- static bool **MP_** (approxPointFileAnnotated)
- static bool **MP_** (modelUseDerivsFlag)
- static bool **MP_** (pointSelection)
- static short **MP_** (annNodes)
- static short **MP_** (annRandomWeight)
- static short **MP_** (krigingMaxTrials)
- static short **MP_** (marsMaxBases)
- static short **MP_** (mlsPolyOrder)
- static short **MP_** (mlsWeightFunction)
- static short **MP_** (rbfBases)
- static short **MP_** (rbfMaxPts)
- static short **MP_** (rbfMaxSubsets)
- static short **MP_** (rbfMinPartition)
- static int **MP_** (pointsTotal)
- static IntList **MP_** (idAnalyticGrads)
- static IntList **MP_** (idAnalyticHessians)
- static IntList **MP_** (idNumericalGrads)
- static IntList **MP_** (idNumericalHessians)
- static IntList **MP_** (idQuasiHessians)
- static RealVector **MP_** (expConfigVars)
- static RealVector **MP_** (expObservations)
- static RealVector **MP_** (expStdDeviations)
- static RealVector **MP_** (primaryRespFnWeights)
- static RealVector **MP_** (nonlinearEqTargets)
- static RealVector **MP_** (nonlinearIneqLowerBnds)
- static RealVector **MP_** (nonlinearIneqUpperBnds)
- static RealVector **MP_** (fdGradStepSize)
- static RealVector **MP_** (fdHessStepSize)
- static RealVector **MP_** (primaryRespFnScales)
- static RealVector **MP_** (nonlinearEqScales)
- static RealVector **MP_** (nonlinearIneqScales)
- static Resp_mp_lit **MP2** (gradientType, analytic)
- static Resp_mp_lit **MP2** (gradientType, mixed)
- static Resp_mp_lit **MP2** (gradientType, none)
- static Resp_mp_lit **MP2** (gradientType, numerical)
- static Resp_mp_lit **MP2** (hessianType, analytic)
- static Resp_mp_lit **MP2** (hessianType, mixed)
- static Resp_mp_lit **MP2** (hessianType, none)

- static Resp_mp_lit **MP2** (hessianType, numerical)
- static Resp_mp_lit **MP2** (hessianType, quasi)
- static Resp_mp_lit **MP2** (intervalType, central)
- static Resp_mp_lit **MP2** (intervalType, forward)
- static Resp_mp_lit **MP2** (methodSource, dakota)
- static Resp_mp_lit **MP2** (methodSource, vendor)
- static Resp_mp_lit **MP2** (quasiHessianType, bfgs)
- static Resp_mp_lit **MP2** (quasiHessianType, damped_bfgs)
- static Resp_mp_lit **MP2** (quasiHessianType, sr1)
- static String **MP_** (expDataFileName)
- static String **MP_** (idResponses)
- static StringArray **MP_** (primaryRespFnScaleTypes)
- static StringArray **MP_** (nonlinearEqScaleTypes)
- static StringArray **MP_** (nonlinearIneqScaleTypes)
- static StringArray **MP_** (responseLabels)
- static bool **MP_** (centralHess)
- static bool **MP_** (expDataFileAnnotated)
- static bool **MP_** (ignoreBounds)
- static size_t **MP_** (numExpStdDeviations)
- static size_t **MP_** (numExpConfigVars)
- static size_t **MP_** (numExperiments)
- static size_t **MP_** (numLeastSqTerms)
- static size_t **MP_** (numNonlinearEqConstraints)
- static size_t **MP_** (numNonlinearIneqConstraints)
- static size_t **MP_** (numObjectiveFunctions)
- static size_t **MP_** (numResponseFunctions)
- static Real **MP_** (hybridLSProb)
- static RealVector **MP_** (concurrentParameterSets)
- static Strategy_mp_lit **MP2** (hybridType, collaborative)
- static Strategy_mp_lit **MP2** (hybridType, embedded)
- static Strategy_mp_lit **MP2** (hybridType, sequential)
- static Strategy_mp_lit **MP2** (iteratorScheduling, self)
- static Strategy_mp_lit **MP2** (iteratorScheduling, static)
- static Strategy_mp_lit **MP2** (strategyType, hybrid)
- static Strategy_mp_lit **MP2** (strategyType, multi_start)
- static Strategy_mp_lit **MP2** (strategyType, pareto_set)
- static Strategy_mp_lit **MP2** (strategyType, single_method)
- static String **MP_** (hybridGlobalMethodPointer)
- static String **MP_** (hybridLocalMethodPointer)
- static String **MP_** (methodPointer)
- static String **MP_** (tabularDataFile)
- static StringArray **MP_** (hybridMethodList)
- static bool **MP_** (graphicsFlag)
- static bool **MP_** (tabularDataFlag)
- static int **MP_** (concurrentRandomJobs)
- static int **MP_** (concurrentSeed)

- static int **MP_** (iteratorServers)
- static int **MP_** (outputPrecision)
- static size_t **MP_** (numBetaUncVars)
- static size_t **MP_** (numBinomialUncVars)
- static size_t **MP_** (numContinuousDesVars)
- static size_t **MP_** (numContinuousStateVars)
- static size_t **MP_** (numDiscreteDesRangeVars)
- static size_t **MP_** (numDiscreteDesSetIntVars)
- static size_t **MP_** (numDiscreteDesSetRealVars)
- static size_t **MP_** (numDiscreteStateRangeVars)
- static size_t **MP_** (numDiscreteStateSetIntVars)
- static size_t **MP_** (numDiscreteStateSetRealVars)
- static size_t **MP_** (numExponentialUncVars)
- static size_t **MP_** (numFrechetUncVars)
- static size_t **MP_** (numGammaUncVars)
- static size_t **MP_** (numGeometricUncVars)
- static size_t **MP_** (numGumbelUncVars)
- static size_t **MP_** (numHistogramBinUncVars)
- static size_t **MP_** (numHistogramPtUncVars)
- static size_t **MP_** (numHyperGeomUncVars)
- static size_t **MP_** (numIntervalUncVars)
- static size_t **MP_** (numLognormalUncVars)
- static size_t **MP_** (numLoguniformUncVars)
- static size_t **MP_** (numNegBinomialUncVars)
- static size_t **MP_** (numNormalUncVars)
- static size_t **MP_** (numPoissonUncVars)
- static size_t **MP_** (numTriangularUncVars)
- static size_t **MP_** (numUniformUncVars)
- static size_t **MP_** (numWeibullUncVars)
- static IntVector **MP_** (binomialUncNumTrials)
- static IntVector **MP_** (hyperGeomUncTotalPop)
- static IntVector **MP_** (hyperGeomUncSelectedPop)
- static IntVector **MP_** (hyperGeomUncNumDrawn)
- static IntVector **MP_** (negBinomialUncNumTrials)
- static IntVector **MP_** (discreteDesignRangeLowerBnds)
- static IntVector **MP_** (discreteDesignRangeUpperBnds)
- static IntVector **MP_** (discreteDesignRangeVars)
- static IntVector **MP_** (discreteDesignSetIntVars)
- static IntVector **MP_** (discreteStateRangeLowerBnds)
- static IntVector **MP_** (discreteStateRangeUpperBnds)
- static IntVector **MP_** (discreteStateRangeVars)
- static IntVector **MP_** (discreteStateSetIntVars)
- static IntArray **VP_** (dsvi)
- static IntArray **VP_** (ndsvi)
- static IntArray **VP_** (ndsvr)
- static IntArray **VP_** (nIv)

- static IntArray **VP_** (nbp)
- static IntArray **VP_** (npp)
- static IntArray **VP_** (nssvi)
- static IntArray **VP_** (nssvr)
- static IntArray **VP_** (ssvi)
- static RealVector **MP_** (betaUncLowerBnds)
- static RealVector **MP_** (betaUncUpperBnds)
- static RealVector **MP_** (binomialUncProbPerTrial)
- static RealVector **MP_** (continuousDesignLowerBnds)
- static RealVector **MP_** (continuousDesignUpperBnds)
- static RealVector **MP_** (continuousDesignVars)
- static RealVector **MP_** (continuousDesignScales)
- static RealVector **MP_** (continuousStateLowerBnds)
- static RealVector **MP_** (continuousStateUpperBnds)
- static RealVector **MP_** (continuousStateVars)
- static RealVector **MP_** (discreteDesignSetRealVars)
- static RealVector **MP_** (discreteStateSetRealVars)
- static RealVector **MP_** (frechetUncBetas)
- static RealVector **MP_** (geometricUncProbPerTrial)
- static RealVector **MP_** (gumbelUncBetas)
- static RealVector **MP_** (negBinomialUncProbPerTrial)
- static RealVector **MP_** (normalUncLowerBnds)
- static RealVector **MP_** (normalUncMeans)
- static RealVector **MP_** (normalUncUpperBnds)
- static RealVector **MP_** (poissonUncLambdas)
- static RealVector **MP_** (triangularUncModes)
- static RealVector **VP_** (dsvr)
- static RealVector **VP_** (Ivb)
- static RealVector **VP_** (Ivp)
- static RealVector **VP_** (ba)
- static RealVector **VP_** (bo)
- static RealVector **VP_** (bc)
- static RealVector **VP_** (pa)
- static RealVector **VP_** (pc)
- static RealVector **VP_** (ucm)
- static RealVector **VP_** (ssvr)
- static String **MP_** (idVariables)
- static StringArray **MP_** (continuousDesignLabels)
- static StringArray **MP_** (continuousDesignScaleTypes)
- static StringArray **MP_** (continuousStateLabels)
- static StringArray **MP_** (discreteDesignRangeLabels)
- static StringArray **MP_** (discreteDesignSetIntLabels)
- static StringArray **MP_** (discreteDesignSetRealLabels)
- static StringArray **MP_** (discreteStateRangeLabels)
- static StringArray **MP_** (discreteStateSetIntLabels)
- static StringArray **MP_** (discreteStateSetRealLabels)

- static Var_brv **MP2s** (betaUncAlphas, 0.)
- static Var_brv **MP2s** (betaUncBetas, 0.)
- static Var_brv **MP2s** (exponentialUncBetas, 0.)
- static Var_brv **MP2s** (frechetUncAlphas, 2.)
- static Var_brv **MP2s** (gammaUncAlphas, 0.)
- static Var_brv **MP2s** (gammaUncBetas, 0.)
- static Var_brv **MP2s** (gumbelUncAlphas, 0.)
- static Var_brv **MP2s** (lognormalUncErrFacts, 1.)
- static Var_brv **MP2s** (lognormalUncLambdas, 0.)
- static Var_brv **MP2s** (lognormalUncLowerBnds, 0.)
- static Var_brv **MP2s** (lognormalUncMeans, 0.)
- static Var_brv **MP2s** (lognormalUncStdDevs, 0.)
- static Var_brv **MP2s** (lognormalUncUpperBnds, DBL_MAX)
- static Var_brv **MP2s** (lognormalUncZetas, 0.)
- static Var_brv **MP2s** (loguniformUncLowerBnds, 0.)
- static Var_brv **MP2s** (loguniformUncUpperBnds, DBL_MAX)
- static Var_brv **MP2s** (normalUncStdDevs, 0.)
- static Var_brv **MP2s** (triangularUncLowerBnds,-DBL_MAX)
- static Var_brv **MP2s** (triangularUncUpperBnds, DBL_MAX)
- static Var_brv **MP2s** (uniformUncLowerBnds,-DBL_MAX)
- static Var_brv **MP2s** (uniformUncUpperBnds, DBL_MAX)
- static Var_brv **MP2s** (weibullUncAlphas, 0.)
- static Var_brv **MP2s** (weibullUncBetas, 0.)
- static const char ∗ **Var_Name** (StringArray ∗sa, char ∗buf, size_t i)
- void **dn2f_** (int ∗n, int ∗p, Real ∗x, Calcrj, int ∗iv, int ∗liv, int ∗lv, Real ∗v, int ∗ui, void ∗ur, Vf)
- void **dn2fb_** (int ∗n, int ∗p, Real ∗x, Real ∗b, Calcrj, int ∗iv, int ∗liv, int ∗lv, Real ∗v, int ∗ui, void ∗ur, Vf)
- void **dn2g_** (int ∗n, int ∗p, Real ∗x, Calcrj, Calcrj, int ∗iv, int ∗liv, int ∗lv, Real ∗v, int ∗ui, void ∗ur, Vf)
- void **dn2gb_** (int ∗n, int ∗p, Real ∗x, Real ∗b, Calcrj, Calcrj, int ∗iv, int ∗liv, int ∗lv, Real ∗v, int ∗ui, void ∗ur, Vf)
- void **divset_** (int ∗, int ∗, int ∗, int ∗, Real ∗)
- double **dr7mdc_** (int ∗)
- static void **Rswapchk** (Nl2Misc ∗q)
- static int **hasnaninf** (const double ∗d, int n)
- NLPQLPOptimizer ∗ **new_NLPQLPOptimizer** (Model &model)
- NLPQLPOptimizer ∗ **new_NLPQLPOptimizer** (NoDBBaseConstructor, Model &model)
- NPSOLOptimizer ∗ **new_NPSOLOptimizer** (Model &model)
- NPSOLOptimizer ∗ **new_NPSOLOptimizer1** (NoDBBaseConstructor, Model &model)
- NPSOLOptimizer ∗ **new_NPSOLOptimizer2** (Model &model, const int &derivative_level, const Real &conv_tol)
- NPSOLOptimizer ∗ **new_NPSOLOptimizer3** (const RealVector &initial_point, const RealVector &var_lower_bnds, const RealVector &var_upper_bnds, const RealMatrix &lin_ineq_coeffs, const RealVector &lin_ineq_lower_bnds, const RealVector &lin_ineq_upper_bnds, const RealMatrix &lin_eq_coeffs, const RealVector &lin_eq_targets, const RealVector &nonlin_ineq_lower_bnds, const RealVector &nonlin_ineq_upper_bnds, const RealVector &nonlin_eq_targets, void(∗user_obj_eval)(int &, int &, double ∗, double &, double ∗, int &), void(∗user_con_eval)(int &, int &, int &, int &, int ∗, double ∗, double ∗, double ∗, int &), const int &derivative_level, const Real &conv_tol)
- NPSOLOptimizer ∗ **new_NPSOLOptimizer** (NoDBBaseConstructor dummy, Model &model)

- NPSOLOptimizer * **new_NPSOLOptimizer** (Model &model, const int &, const Real &)
- NPSOLOptimizer * **new_NPSOLOptimizer** (const RealVector &initial_point, const RealVector &var_lower_bnds, const RealVector &var_upper_bnds, const RealMatrix &lin_ineq_coeffs, const RealVector &lin_ineq_lower_bnds, const RealVector &lin_ineq_upper_bnds, const RealMatrix &lin_eq_coeffs, const RealVector &lin_eq_targets, const RealVector &nonlin_ineq_lower_bnds, const RealVector &nonlin_ineq_upper_bnds, const RealVector &nonlin_eq_targets, void(*user_obj_eval)(int &, int &, double *, double &, double *, int &), void(*user_con_eval)(int &, int &, int &, int &, int *, double *, double *, double *, int &), const int &derivative_level, const Real &conv_tol)
- std::istream & operator>> (std::istream &s, ParamResponsePair &pair)

  std::istream extraction operator for ParamResponsePair

- std::ostream & operator<< (std::ostream &s, const ParamResponsePair &pair)

  std::ostream insertion operator for ParamResponsePair

- BiStream & operator>> (BiStream &s, ParamResponsePair &pair)

  BiStream extraction operator for ParamResponsePair.

- BoStream & operator<< (BoStream &s, const ParamResponsePair &pair)

  BoStream insertion operator for ParamResponsePair.

- MPIUnpackBuffer & operator>> (MPIUnpackBuffer &s, ParamResponsePair &pair)

  MPIUnpackBuffer extraction operator for ParamResponsePair.

- MPIPackBuffer & operator<< (MPIPackBuffer &s, const ParamResponsePair &pair)

  MPIPackBuffer insertion operator for ParamResponsePair.

- bool operator== (const ParamResponsePair &pair1, const ParamResponsePair &pair2)

  equality operator for ParamResponsePair

- bool operator!= (const ParamResponsePair &pair1, const ParamResponsePair &pair2)

  inequality operator for ParamResponsePair

- static void * **binsearch** (void *kw, size_t kwsize, size_t n, const char *key)
- static const char * **Begins** (const String &entry_name, const char *s)
- static void **Bad_name** (String entry_name, const char *where)
- static void **Locked_db** ()
- static void **Null_rep** (const char *who)
- static void **Null_rep1** (const char *who)
- bool set_compare (const ParamResponsePair &database_pr, const ActiveSet &search_set)

  search function for a particular ParamResponsePair within a PRPList based on ActiveSet content (request vector and derivative variables vector)

- bool id_vars_exact_compare (const ParamResponsePair &database_pr, const ParamResponsePair &search_pr)

  search function for a particular ParamResponsePair within a PRPMultiIndex

- std::size_t hash_value (const ParamResponsePair &prp)

hash_value for ParamResponsePairs stored in a PRPMultiIndex

- PRPCacheHIter hashedCacheBegin (PRPCache &prp_cache)

  hashed definition of cache begin

- PRPCacheHIter hashedCacheEnd (PRPCache &prp_cache)

  hashed definition of cache end

- PRPQueueHIter hashedQueueBegin (PRPQueue &prp_queue)

  hashed definition of queue begin

- PRPQueueHIter hashedQueueEnd (PRPQueue &prp_queue)

  hashed definition of queue end

- PRPCacheHIter lookup_by_val (PRPMultiIndexCache &prp_cache, const ParamResponsePair &search_pr)

  find a ParamResponsePair based on the interface id, variables, and ActiveSet search data within search_pr.

- bool lookup_by_val (PRPMultiIndexCache &prp_cache, const ParamResponsePair &search_pr, ParamResponsePair &found_pr)

  alternate overloaded form returns bool and sets found_pr by wrapping lookup_by_val(PRPMultiIndexCache&, ParamResponsePair&)

- PRPCacheHIter lookup_by_val (PRPMultiIndexCache &prp_cache, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set)

  find the evaluation id of a ParamResponsePair within a PRPMultiIndexCache based on interface id, variables, and ActiveSet search data

- bool lookup_by_val (PRPMultiIndexCache &prp_cache, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set, ParamResponsePair &found_pr)

  find a ParamResponsePair within a PRPMultiIndexCache based on interface id, variables, and ActiveSet search data

- bool lookup_by_val (PRPMultiIndexCache &prp_cache, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set, int &found_eval_id)

  find the evaluation id of a ParamResponsePair within a PRPMultiIndexCache based on interface id, variables, and ActiveSet search data

- bool lookup_by_val (PRPMultiIndexCache &prp_cache, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set, Response &found_resp)

  find the response of a ParamResponsePair within a PRPMultiIndexCache based on interface id, variables, and ActiveSet search data

- PRPCacheOIter lookup_by_ids (PRPMultiIndexCache &prp_cache, const IntStringPair &search_ids)

  find a ParamResponsePair within a PRPMultiIndexCache based on search_ids (i.e. std::pair<eval_id,interface_id>) search data

- bool lookup_by_ids (PRPMultiIndexCache &prp_cache, const IntStringPair &search_eval_interface_ids, ParamResponsePair &found_pr)

  find a ParamResponsePair within a PRPMultiIndexCache based on eval_interface_ids

- bool lookup_by_ids (PRPMultiIndexCache &prp_cache, const ParamResponsePair &search_pr, ParamResponsePair &found_pr)

  find a ParamResponsePair within a PRPMultiIndexCache based on eval_interface_ids from the ParamResponsePair search data

- PRPQueueHIter lookup_by_val (PRPMultiIndexQueue &prp_queue, const ParamResponsePair &search_pr)

  find a ParamResponsePair based on the interface id, variables, and ActiveSet search data within search_pr.

- bool lookup_by_val (PRPMultiIndexQueue &prp_queue, const ParamResponsePair &search_pr, ParamResponsePair &found_pr)

  alternate overloaded form returns bool and sets found_pr by wrapping lookup_by_val(PRPMultiIndexQueue&, ParamResponsePair&)

- PRPQueueHIter lookup_by_val (PRPMultiIndexQueue &prp_queue, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set)

  find the evaluation id of a ParamResponsePair within a PRPMultiIndexQueue based on interface id, variables, and ActiveSet search data

- bool lookup_by_val (PRPMultiIndexQueue &prp_queue, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set, ParamResponsePair &found_pr)

  find a ParamResponsePair within a PRPMultiIndexQueue based on interface id, variables, and ActiveSet search data

- bool lookup_by_val (PRPMultiIndexQueue &prp_queue, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set, int &found_eval_id)

  find the evaluation id of a ParamResponsePair within a PRPMultiIndexQueue based on interface id, variables, and ActiveSet search data

- bool lookup_by_val (PRPMultiIndexQueue &prp_queue, const String &search_interface_id, const Variables &search_vars, const ActiveSet &search_set, Response &found_resp)

  find the response of a ParamResponsePair within a PRPMultiIndexQueue based on interface id, variables, and ActiveSet search data

- PRPQueueOIter lookup_by_eval_id (PRPMultiIndexQueue &prp_queue, const int &search_id)

  find a ParamResponsePair within a PRPMultiIndexQueue based on search_id (i.e. integer eval_id) search data

- bool lookup_by_eval_id (PRPMultiIndexQueue &prp_queue, const int &search_id, ParamResponsePair &found_pr)

  find a ParamResponsePair within a PRPMultiIndexQueue based on eval_id

- bool lookup_by_eval_id (PRPMultiIndexQueue &prp_queue, const ParamResponsePair &search_pr, ParamResponsePair &found_pr)

find a ParamResponsePair within a PRPMultiIndexQueue based on eval_id from the ParamResponsePair search data

- void print_restart (int argc, char **argv, String print_dest)

  print a restart file

- void print_restart_tabular (int argc, char **argv, String print_dest)

  print a restart file (tabular format)

- void read_neutral (int argc, char **argv)

  read a restart file (neutral file format)

- void repair_restart (int argc, char **argv, String identifier_type)

  repair a restart file by removing corrupted evaluations

- void concatenate_restart (int argc, char **argv)

  concatenate multiple restart files

- void **find_env_token** (const char *s0, const char **s1, const char **s2, const char **s3)
- const char ** arg_list_adjust (const char **, void **)

  Utility function from legacy, "not_executable" module -- DO NOT TOUCH!

- bool path_is_absolute (const bfs::path &p)

  Utilities to manage API differences in BFS_VERSION_2 vs BFS_VERSION_3.

- std::string filename (const bfs::path &file_path)

  Same as file_path.filename() in latest boost::filesystem.

- bfs::path parent_path (const bfs::path &fp)

  Should be same as fp.parent_path() in latest boost::filesystem.

- bool contains (const bfs::path &dir_path, const std::string &file_name, boost::filesystem::path &complete_-filepath)

  Helper for "which" - sets complete_filepath from dir_path/file_name combo.

Variables

- ParallelLibrary * Dak_pl

  set by ParallelLibrary, for use in CLH

- ProblemDescDB dummy_db

  dummy ProblemDescDB object used for < mandatory reference initialization when a < real ProblemDescDB instance is unavailable

- ParallelLibrary dummy_lib

dummy ParallelLibrary object used for < mandatory reference initialization when < a real ParallelLibrary instance is < unavailable

- Graphics dakota_graphics

  the global Dakota::Graphics object used by < strategies, models, and approximations

- Interface dummy_interface

  dummy Interface object used for mandatory < reference initialization or default virtual < function return by reference when a real < Interface instance is unavailable

- Model dummy_model

  dummy Model object used for mandatory reference < initialization or default virtual function < return by reference when a real Model instance < is unavailable

- Iterator dummy_iterator

  dummy Iterator object used for mandatory < reference initialization or default virtual < function return by reference when a real < Iterator instance is unavailable

- const char ∗ FIELD_NAMES [ ]
- const int NUMBER_OF_FIELDS = 25
- PRPCache data_pairs

  contains all parameter/response pairs.

- Dakota_funcs ∗ **DF**
- Dakota_funcs **DakFuncs0**
- static time_t **start_time**
- static int **dakdrive**
- static char **slmap** [256]
- std::ostream ∗ dakota_cout = &cout

  DAKOTA stdout initially points to cout, < but may be redirected to a tagged ofstream if < there are concurrent iterators.

- std::ostream ∗ dakota_cerr = &cerr

  DAKOTA stderr initially points to cerr, < but may be redirected to a tagged ofstream if < there are concurrent iterators.

- BoStream write_restart

  the restart binary output stream (doesn't < really need to be global anymore except for < abort_handler()).

- int write_precision = 10

  used in ostream data output functions < (restart_util.C overrides this default value)

- int mc_ptr_int = 0

  global pointer for ModelCenter API

- int dc_ptr_int = 0

  global pointer for ModelCenter eval DB

- ProblemDescDB * Dak_pddb

  set by ProblemDescDB, for use in parsing

- const size_t _NPOS = ~(size_t)0

  special value returned by index() when entry not found

- static GuiKeyWord kw_1 [3]
- static GuiKeyWord **kw_2** [1]
- static GuiKeyWord **kw_3** [4]
- static GuiKeyWord **kw_4** [2]
- static GuiKeyWord **kw_5** [7]
- static GuiKeyWord **kw_6** [8]
- static GuiKeyWord **kw_7** [9]
- static GuiKeyWord **kw_8** [4]
- static GuiKeyWord **kw_9** [10]
- static GuiKeyWord **kw_10** [7]
- static GuiKeyWord **kw_11** [2]
- static GuiKeyWord **kw_12** [18]
- static GuiKeyWord **kw_13** [1]
- static GuiKeyWord **kw_14** [2]
- static GuiKeyWord **kw_15** [1]
- static GuiKeyWord **kw_16** [1]
- static GuiKeyWord **kw_17** [1]
- static GuiKeyWord **kw_18** [4]
- static GuiKeyWord **kw_19** [1]
- static GuiKeyWord **kw_20** [2]
- static GuiKeyWord **kw_21** [2]
- static GuiKeyWord **kw_22** [10]
- static GuiKeyWord **kw_23** [3]
- static GuiKeyWord **kw_24** [7]
- static GuiKeyWord **kw_25** [2]
- static GuiKeyWord **kw_26** [11]
- static GuiKeyWord **kw_27** [3]
- static GuiKeyWord **kw_28** [2]
- static GuiKeyWord **kw_29** [3]
- static GuiKeyWord **kw_30** [2]
- static GuiKeyWord **kw_31** [5]
- static GuiKeyWord **kw_32** [4]
- static GuiKeyWord **kw_33** [14]
- static GuiKeyWord **kw_34** [3]
- static GuiKeyWord **kw_35** [2]
- static GuiKeyWord **kw_36** [2]
- static GuiKeyWord **kw_37** [17]
- static GuiKeyWord **kw_38** [13]
- static GuiKeyWord **kw_39** [9]

- static GuiKeyWord **kw_40** [1]
- static GuiKeyWord **kw_41** [14]
- static GuiKeyWord **kw_42** [2]
- static GuiKeyWord **kw_43** [15]
- static GuiKeyWord **kw_44** [10]
- static GuiKeyWord **kw_45** [2]
- static GuiKeyWord **kw_46** [4]
- static GuiKeyWord **kw_47** [3]
- static GuiKeyWord **kw_48** [1]
- static GuiKeyWord **kw_49** [8]
- static GuiKeyWord **kw_50** [1]
- static GuiKeyWord **kw_51** [10]
- static GuiKeyWord **kw_52** [2]
- static GuiKeyWord **kw_53** [1]
- static GuiKeyWord **kw_54** [1]
- static GuiKeyWord **kw_55** [2]
- static GuiKeyWord **kw_56** [2]
- static GuiKeyWord **kw_57** [3]
- static GuiKeyWord **kw_58** [2]
- static GuiKeyWord **kw_59** [2]
- static GuiKeyWord **kw_60** [9]
- static GuiKeyWord **kw_61** [2]
- static GuiKeyWord **kw_62** [3]
- static GuiKeyWord **kw_63** [2]
- static GuiKeyWord **kw_64** [5]
- static GuiKeyWord **kw_65** [2]
- static GuiKeyWord **kw_66** [1]
- static GuiKeyWord **kw_67** [1]
- static GuiKeyWord **kw_68** [2]
- static GuiKeyWord **kw_69** [2]
- static GuiKeyWord **kw_70** [2]
- static GuiKeyWord **kw_71** [2]
- static GuiKeyWord **kw_72** [12]
- static GuiKeyWord **kw_73** [2]
- static GuiKeyWord **kw_74** [2]
- static GuiKeyWord **kw_75** [7]
- static GuiKeyWord **kw_76** [1]
- static GuiKeyWord **kw_77** [2]
- static GuiKeyWord **kw_78** [1]
- static GuiKeyWord **kw_79** [1]
- static GuiKeyWord **kw_80** [2]
- static GuiKeyWord **kw_81** [2]
- static GuiKeyWord **kw_82** [6]
- static GuiKeyWord **kw_83** [2]
- static GuiKeyWord **kw_84** [5]
- static GuiKeyWord **kw_85** [3]

- static GuiKeyWord **kw_86** [9]
- static GuiKeyWord **kw_87** [1]
- static GuiKeyWord **kw_88** [3]
- static GuiKeyWord **kw_89** [2]
- static GuiKeyWord **kw_90** [7]
- static GuiKeyWord **kw_91** [2]
- static GuiKeyWord **kw_92** [5]
- static GuiKeyWord **kw_93** [3]
- static GuiKeyWord **kw_94** [1]
- static GuiKeyWord **kw_95** [6]
- static GuiKeyWord **kw_96** [3]
- static GuiKeyWord **kw_97** [2]
- static GuiKeyWord **kw_98** [2]
- static GuiKeyWord **kw_99** [1]
- static GuiKeyWord **kw_100** [2]
- static GuiKeyWord **kw_101** [4]
- static GuiKeyWord **kw_102** [21]
- static GuiKeyWord **kw_103** [1]
- static GuiKeyWord **kw_104** [4]
- static GuiKeyWord **kw_105** [9]
- static GuiKeyWord **kw_106** [1]
- static GuiKeyWord **kw_107** [3]
- static GuiKeyWord **kw_108** [2]
- static GuiKeyWord **kw_109** [7]
- static GuiKeyWord **kw_110** [2]
- static GuiKeyWord **kw_111** [3]
- static GuiKeyWord **kw_112** [3]
- static GuiKeyWord **kw_113** [2]
- static GuiKeyWord **kw_114** [3]
- static GuiKeyWord **kw_115** [3]
- static GuiKeyWord **kw_116** [2]
- static GuiKeyWord **kw_117** [5]
- static GuiKeyWord **kw_118** [2]
- static GuiKeyWord **kw_119** [23]
- static GuiKeyWord **kw_120** [1]
- static GuiKeyWord **kw_121** [4]
- static GuiKeyWord **kw_122** [1]
- static GuiKeyWord **kw_123** [12]
- static GuiKeyWord **kw_124** [2]
- static GuiKeyWord **kw_125** [2]
- static GuiKeyWord **kw_126** [2]
- static GuiKeyWord **kw_127** [3]
- static GuiKeyWord **kw_128** [2]
- static GuiKeyWord **kw_129** [2]
- static GuiKeyWord **kw_130** [2]
- static GuiKeyWord **kw_131** [24]

- static GuiKeyWord **kw_132** [1]
- static GuiKeyWord **kw_133** [12]
- static GuiKeyWord **kw_134** [11]
- static GuiKeyWord **kw_135** [10]
- static GuiKeyWord **kw_136** [4]
- static GuiKeyWord **kw_137** [16]
- static GuiKeyWord **kw_138** [5]
- static GuiKeyWord **kw_139** [3]
- static GuiKeyWord **kw_140** [4]
- static GuiKeyWord **kw_141** [2]
- static GuiKeyWord **kw_142** [2]
- static GuiKeyWord **kw_143** [2]
- static GuiKeyWord **kw_144** [2]
- static GuiKeyWord **kw_145** [4]
- static GuiKeyWord **kw_146** [19]
- static GuiKeyWord **kw_147** [14]
- static GuiKeyWord **kw_148** [3]
- static GuiKeyWord **kw_149** [2]
- static GuiKeyWord **kw_150** [7]
- static GuiKeyWord **kw_151** [1]
- static GuiKeyWord **kw_152** [4]
- static GuiKeyWord **kw_153** [6]
- static GuiKeyWord **kw_154** [18]
- static GuiKeyWord **kw_155** [3]
- static GuiKeyWord **kw_156** [75]
- static GuiKeyWord **kw_157** [1]
- static GuiKeyWord **kw_158** [4]
- static GuiKeyWord **kw_159** [2]
- static GuiKeyWord **kw_160** [1]
- static GuiKeyWord **kw_161** [6]
- static GuiKeyWord **kw_162** [3]
- static GuiKeyWord **kw_163** [2]
- static GuiKeyWord **kw_164** [4]
- static GuiKeyWord **kw_165** [4]
- static GuiKeyWord **kw_166** [2]
- static GuiKeyWord **kw_167** [2]
- static GuiKeyWord **kw_168** [2]
- static GuiKeyWord **kw_169** [2]
- static GuiKeyWord **kw_170** [3]
- static GuiKeyWord **kw_171** [2]
- static GuiKeyWord **kw_172** [3]
- static GuiKeyWord **kw_173** [4]
- static GuiKeyWord **kw_174** [3]
- static GuiKeyWord **kw_175** [18]
- static GuiKeyWord **kw_176** [6]
- static GuiKeyWord **kw_177** [3]

- static GuiKeyWord **kw_178** [2]
- static GuiKeyWord **kw_179** [2]
- static GuiKeyWord **kw_180** [5]
- static GuiKeyWord **kw_181** [6]
- static GuiKeyWord **kw_182** [5]
- static GuiKeyWord **kw_183** [3]
- static GuiKeyWord **kw_184** [4]
- static GuiKeyWord **kw_185** [12]
- static GuiKeyWord **kw_186** [1]
- static GuiKeyWord **kw_187** [10]
- static GuiKeyWord **kw_188** [2]
- static GuiKeyWord **kw_189** [1]
- static GuiKeyWord **kw_190** [2]
- static GuiKeyWord **kw_191** [5]
- static GuiKeyWord **kw_192** [3]
- static GuiKeyWord **kw_193** [4]
- static GuiKeyWord **kw_194** [7]
- static GuiKeyWord **kw_195** [8]
- static GuiKeyWord **kw_196** [4]
- static GuiKeyWord **kw_197** [1]
- static GuiKeyWord **kw_198** [2]
- static GuiKeyWord **kw_199** [18]
- static GuiKeyWord **kw_200** [1]
- static GuiKeyWord **kw_201** [3]
- static GuiKeyWord **kw_202** [1]
- static GuiKeyWord **kw_203** [5]
- static GuiKeyWord **kw_204** [1]
- static GuiKeyWord **kw_205** [3]
- static GuiKeyWord **kw_206** [1]
- static GuiKeyWord **kw_207** [5]
- static GuiKeyWord **kw_208** [1]
- static GuiKeyWord **kw_209** [1]
- static GuiKeyWord **kw_210** [10]
- static GuiKeyWord **kw_211** [10]
- static GuiKeyWord **kw_212** [3]
- static GuiKeyWord **kw_213** [12]
- static GuiKeyWord **kw_214** [8]
- static GuiKeyWord **kw_215** [8]
- static GuiKeyWord **kw_216** [4]
- static GuiKeyWord **kw_217** [4]
- static GuiKeyWord **kw_218** [8]
- static GuiKeyWord **kw_219** [4]
- static GuiKeyWord **kw_220** [4]
- static GuiKeyWord **kw_221** [4]
- static GuiKeyWord **kw_222** [6]
- static GuiKeyWord **kw_223** [6]

- static GuiKeyWord **kw_224** [2]
- static GuiKeyWord **kw_225** [6]
- static GuiKeyWord **kw_226** [10]
- static GuiKeyWord **kw_227** [8]
- static GuiKeyWord **kw_228** [4]
- static GuiKeyWord **kw_229** [8]
- static GuiKeyWord **kw_230** [2]
- static GuiKeyWord **kw_231** [4]
- static GuiKeyWord **kw_232** [10]
- static GuiKeyWord **kw_233** [6]
- static GuiKeyWord **kw_234** [3]
- static GuiKeyWord **kw_235** [10]
- static GuiKeyWord **kw_236** [2]
- static GuiKeyWord **kw_237** [8]
- static GuiKeyWord **kw_238** [6]
- static GuiKeyWord **kw_239** [6]
- static GuiKeyWord **kw_240** [29]
- static GuiKeyWord **kw_241** [6]
- static KeyWord **kw_242** [4]
- static KeyWord **kw_243** [8]
- static KeyWord **kw_244** [2]
- static KeyWord **kw_245** [4]
- static KeyWord **kw_246** [10]
- static KeyWord **kw_247** [6]
- static KeyWord **kw_248** [3]
- static KeyWord **kw_249** [10]
- static KeyWord **kw_250** [2]
- static KeyWord **kw_251** [8]
- static KeyWord **kw_252** [6]
- static KeyWord **kw_253** [6]
- static KeyWord **kw_254** [29]
- static KeyWord **kw_255** [6]
- FILE ∗ **nidrin**
- static const char ∗ **aln_scaletypes** [ ] = { "auto", "log", "none", 0 }
- static Var_uinfo **CAUVLbl** [CAUVar_Nkinds]
- static Var_uinfo **DAUIVLbl** [DAUIVar_Nkinds]
- static Var_uinfo **DAURVLbl** [DAURVar_Nkinds]
- static Var_uinfo **CEUVLbl** [CEUVar_Nkinds]
- static Var_uinfo **DiscSetLbl** [DiscSetVar_Nkinds]
- static VarLabelChk **Vlch** [ ]
- static VLstuff **VLS** [N_VLS]
- static int **VL_aleatory** [N_VLS] = { 1, 0, 1, 1 }
- static String **MP_** (algebraicMappings)
- static Var_bgen **var_mp_bgen** [ ]
- static Var_bgen **var_mp_bgen_audr** [ ]
- static Var_bgen **var_mp_bgen_audi** [ ]

- static Var_bgen **var_mp_bgen_eu** [ ]
- static Var_bgen **var_mp_bgen_dis** [ ]
- static VarBgen **Bgen** [ ]
- static Var_bchk **var_mp_bndchk** [ ]
- static Var_ibchk **var_mp_ibndchk** [ ]
- const int LARGE_SCALE = 100

  a (perhaps arbitrary) definition of large scale; choose a large-scale algorithm if numVars >= LARGE_SCALE

42.1.1 Detailed Description

The primary namespace for DAKOTA. Utility functions for reading and writing tabular data files.

The COLINOptimizer class wraps COLIN, a Sandia-developed C++ optimization interface library. A variety of COLIN optimizers are defined in COLIN and its associated libraries, including SCOLIB which contains the optimization components from the old COLINY (formerly SGOPT) library. COLIN contains optimizers such as genetic algorithms, pattern search methods, and other nongradient-based techniques. COLINOptimizer uses a COLINApplication object to perform the function evaluations.

The user input mappings are as follows: `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, and `solution_accuracy` are mapped into COLIN's `max_iterations`, `max_function_evaluations_this_trial`, `function_value_tolerance`, `sufficient_objective_value` properties. An `outputLevel` is mapped to COLIN's `output_level` property and a setting of `debug` activates output of method initialization and sets the COLIN `debug` attribute to 10000 for the DEBUG output level. Refer to [Hart, W.E., 2006] for additional information on COLIN objects and controls.

The Dakota namespace encapsulates the core classes of the DAKOTA framework and prevents name clashes with third-party libraries from methods and packages. The C++ source files defining these core classes reside in Dakota/src as ∗.[CH].

Emerging utilities for tabular file I/O. For now, just extraction of capability from separate contexts to faciliate rework. These augment (and leverage) those in data_util.h

Design/capability goals: Ability to read / write data with row/col headers or in free-form Detect premature end of file, report if extra data More consistent and reliable checks for file open errors Require right number of cols in header mode; only total data checking in free-form (likely) Allow comment character for header rows or even in data? variables vs. variables/responses for both read and write Should we support CSV? delimiter = ',’; other? Verify treatment of trailing newline without reading a zero Allow reading into the transpose of the data structure

42.1.2 Typedef Documentation

42.1.2.1 typedef bmi::multi_index_container<Dakota::ParamResponsePair, bmi::indexed_by< bmi::ordered_unique<bmi::tag<ordered>, bmi::const_mem_fun<Dakota::ParamResponsePair, const IntStringPair&, &Dakota::ParamResponsePair::eval_interface_ids> >, bmi::hashed_non_unique<bmi::tag<hashed>, bmi::identity<Dakota::ParamResponsePair>, partial_prp_hash, partial_prp_equality> > > PRPMultiIndexCache

Boost Multi-Index Container for globally caching ParamResponsePairs. For a global cache, both evaluation and interface id's are used for tagging ParamResponsePair records.

42.1.2.2 typedef bmi::multi_index_container<Dakota::ParamResponsePair, bmi::indexed_by< bmi::ordered_unique<bmi::tag<ordered>, bmi::const_mem_-fun<Dakota::ParamResponsePair, int, &Dakota::ParamResponsePair::eval_id> >, bmi::hashed_non_unique<bmi::tag<hashed>, bmi::identity<Dakota::ParamResponsePair>, partial_prp_hash, partial_prp_equality> > > PRPMultiIndexQueue

Boost Multi-Index Container for locally queueing ParamResponsePairs. For a local queue, interface id's are expected to be consistent, such that evaluation id's are sufficient for tracking particular evaluations.

42.1.3 Function Documentation

42.1.3.1 CommandShell & flush (CommandShell & *shell*)

convenient shell manipulator function to "flush" the shell global convenience function for manipulating the shell; invokes the class member flush function.

Referenced by SysCallAnalysisCode::spawn_analysis(), SysCallAnalysisCode::spawn_evaluation(), SysCallAnalysisCode::spawn_input_filter(), and SysCallAnalysisCode::spawn_output_filter().

42.1.3.2 Real Dakota::rel_change_rv (const RealVector & *curr_rv*, const RealVector & *prev_rv*) `[inline]`

Computes relative change between successive RealVectors using Euclidean norm. Removes the first instance matching object a from the list (and therefore differs from the STL list::remove() which removes all instances). Uses the STL find() algorithm to find the object and the list.erase() method to perform the remove.// WJB: Only used in ProblemDescDB.C! template <class listt>=""> void remove(ListT& l, typename ListT::value_type a) { typename ListT::iterator it = std::find(l.begin(), l.end(), a); if (it != l.end()) { l.erase(it); } }

Referenced by EffGlobalMinimizer::minimize_surrogates_on_model(), and NonDGlobalInterval::post_process_-gp_results().

42.1.3.3 void start_dakota_heartbeat (int)

Heartbeat function provided by not_executable.C; pass output interval in seconds, or -1 to use $DAKOTA_-HEARTBEAT

Referenced by ParallelLibrary::init_mpi_comm(), and ParallelLibrary::ParallelLibrary().

42.1.3.4 int Dakota::my_cp (const char ∗ *file*, const struct stat ∗ *sb*, int *ftype*, int *depth*, void ∗ *v*)

my_cp is a wrapper around 'cp -r'. The extra layer allows for symlink to be used instead of file copy.

42.1.3.5 void Dakota::get_npath (int *appdrive*, std::string ∗ *pnpath*)

get_npath "shuffles" the string representing the current $PATH variable definition so that '.' is first in the $PATH. It then returns the new string as the result (last arg in the call).

References get_cwd().

42.1.3.6 std::string get_cwd ()

Portability adapter for getcwd. Portability adapter for getcwd

Referenced by get_npath().

42.1.3.7 Real Dakota::getdist (const RealVector & *x1*, const RealVector & *x2*)

Gets the Euclidean distance between x1 and x2

Referenced by mindist(), and mindistindx().

42.1.3.8 Real Dakota::mindist (const RealVector & *x*, const RealMatrix & *xset*, int *except*)

Returns the minimum distance between the point x and the points in the set xset (compares against all points in xset except point "except"): if except is not needed, pass 0.

References getdist().

Referenced by getRmax().

42.1.3.9 Real Dakota::mindistindx (const RealVector & *x*, const RealMatrix & *xset*, const IntArray & *indx*)

Gets the min distance between x and points in the set xset defined by the nindx values in indx.

References getdist().

Referenced by GaussProcApproximation::pointsel_add_sel().

42.1.3.10 Real Dakota::getRmax (const RealMatrix & *xset*)

Gets the maximum of the min distance between each point and the rest of the set.

References mindist().

Referenced by GaussProcApproximation::pointsel_add_sel().

42.1.3.11 T Dakota::abort_handler_t (int *code*) `[inline]`

Templatized abort_handler_t method that allows for convenient return from methods that otherwise have no sensible return from error clauses. Usage: MyType& method() { return abort_handler<MyType&>(-1); }

References abort_handler().

42.1.3.12 int Dakota::start_grid_computing (char * *analysis_driver_script*, char * *params_file*, char * *results_file*)

sample function prototype for launching grid computing

42.1.3.13 int Dakota::stop_grid_computing ()

sample function prototype for terminating grid computing

42.1.3.14 int Dakota::perform_analysis (char * *iteration_num*)

sample function prototype for submitting a grid evaluation

References SysCallAnalysisCode::spawn_evaluation().

42.1.3.15 string Dakota::asstring (const T & *val*) `[inline]`

Creates a string from the argument *val* using an ostringstream. This only gets used in this file and is only ever called with ints so no error checking is in place.

Parameters:

val The value of type T to convert to a string.

Returns:

The string representation of *val* created using an ostringstream.

Referenced by JEGAOptimizer::LoadTheConstraints(), and JEGAOptimizer::LoadTheObjectiveFunctions().

42.1.3.16 void run_dakota_data ()

Function to encapsulate the DAKOTA object instantiations for mode 2: direct Data class instantiation. Rather than parsing from an input file, this function populates Data class objects directly using a minimal specification and relies on constructor defaults and post-processing in post_process() to fill in the rest.

References ProblemDescDB::broadcast(), ProblemDescDB::check_input(), DataInterface::dataIfaceRep, DataMethod::dataMethodRep, DataResponses::dataRespRep, DataVariables::dataVarsRep, DataResponsesRep::gradientType, DataResponsesRep::hessianType, ProblemDescDB::insert_node(), ProblemDescDB::lock(), DataMethodRep::methodName, model_interface_plugins(), ParallelLibrary::mpirun_flag(), DataVariablesRep::numContinuousDesVars, DataResponsesRep::numNonlinearIneqConstraints, DataResponsesRep::numObjectiveFunctions, ProblemDescDB::post_process(), Strategy::run_strategy(), and ParallelLibrary::world_rank().

Referenced by main().

42.1.3.17 bool Dakota::set_compare (const ParamResponsePair & *database_pr*, const ActiveSet & *search_set*) `[inline]`

search function for a particular ParamResponsePair within a PRPList based on ActiveSet content (request vector and derivative variables vector) a global function to compare the ActiveSet of a particular database_pr (presumed to be in the global history list) with a passed in ActiveSet (search_set).

References ParamResponsePair::active_set(), ActiveSet::derivative_vector(), and ActiveSet::request_vector().

Referenced by lookup_by_val().

42.1.3.18 bool Dakota::id_vars_exact_compare (const ParamResponsePair & *database_pr*, const ParamResponsePair & *search_pr*) [inline]

search function for a particular ParamResponsePair within a PRPMultiIndex a global function to compare the interface id and variables of a particular database_pr (presumed to be in the global history list) with a passed in key of interface id and variables provided by search_pr.

References binary_equal_to(), ParamResponsePair::interface_id(), and ParamResponsePair::prp_parameters().

Referenced by partial_prp_equality::operator()().

42.1.3.19 PRPCacheHIter Dakota::lookup_by_val (PRPMultiIndexCache & *prp_cache*, const ParamResponsePair & *search_pr*) [inline]

find a ParamResponsePair based on the interface id, variables, and ActiveSet search data within search_pr. Lookup occurs in two steps: (1) PRPMultiIndexCache lookup based on strict equality in interface id and variables, and (2) set_compare() post-processing based on ActiveSet subset logic.

References ParamResponsePair::active_set(), and set_compare().

Referenced by ApplicationInterface::duplication_detect(), Model::estimate_derivatives(), SurrBasedLocalMinimizer::find_center_approx(), Optimizer::local_objective_recast_retrieve(), lookup_-by_val(), SNLLLeastSq::post_run(), COLINOptimizer::post_run(), SurrBasedMinimizer::print_-results(), Optimizer::print_results(), LeastSq::print_results(), DiscrepancyCorrection::search_db(), and NonDLocalReliability::update_mpp_search_data().

42.1.3.20 PRPQueueHIter Dakota::lookup_by_val (PRPMultiIndexQueue & *prp_queue*, const ParamResponsePair & *search_pr*) [inline]

find a ParamResponsePair based on the interface id, variables, and ActiveSet search data within search_pr. Lookup occurs in two steps: (1) PRPMultiIndexQueue lookup based on strict equality in interface id and variables, and (2) set_compare() post-processing based on ActiveSet subset logic.

References ParamResponsePair::active_set(), and set_compare().

42.1.3.21 void print_restart (int *argc*, char ** *argv*, String *print_dest*)

print a restart file **Usage:** "dakota_restart_util print dakota.rst"

"dakota_restart_util to_neutral dakota.rst dakota.neu"

Prints all evals. in full precision to either stdout or a neutral file. The former is useful for ensuring that duplicate detection is successful in a restarted run (e.g., starting a new method from the previous best), and the latter is used for translating binary files between platforms.

References ParamResponsePair::eval_id(), ParamResponsePair::write_annotated(), and write_precision.

Referenced by main().

42.1.3.22 void print_restart_tabular (int *argc*, char ** *argv*, String *print_dest*)

print a restart file (tabular format) **Usage:** "dakota_restart_util to_pdb dakota.rst dakota.pdb"

"dakota_restart_util to_tabular dakota.rst dakota.txt"

Unrolls all data associated with a particular tag for all evaluations and then writes this data in a tabular format (e.g., to a PDB database or MATLAB/TECPLOT data file).

References Variables::continuous_variables(), String::data(), data_pairs, Variables::discrete_int_variables(), Variables::discrete_real_variables(), and Response::function_values().

Referenced by main().

42.1.3.23 void read_neutral (int *argc*, char ** *argv*)

read a restart file (neutral file format) **Usage:** "dakota_restart_util from_neutral dakota.neu dakota.rst"

Reads evaluations from a neutral file. This is used for translating binary files between platforms.

References ParamResponsePair::read_annotated(), and write_restart.

Referenced by main().

42.1.3.24 void repair_restart (int *argc*, char ** *argv*, String *identifier_type*)

repair a restart file by removing corrupted evaluations **Usage:** "dakota_restart_util remove 0.0 dakota_old.rst dakota_new.rst"

"dakota_restart_util remove_ids 2 7 13 dakota_old.rst dakota_new.rst"

Repairs a restart file by removing corrupted evaluations. The identifier for evaluation removal can be either a double precision number (all evaluations having a matching response function value are removed) or a list of integers (all evaluations with matching evaluation ids are removed).

References Response::active_set_request_vector(), contains(), ParamResponsePair::eval_id(), Response::function_values(), ParamResponsePair::prp_response(), and write_restart.

Referenced by main().

42.1.3.25 void concatenate_restart (int *argc*, char ** *argv*)

concatenate multiple restart files **Usage:** "dakota_restart_util cat dakota_1.rst ... dakota_n.rst dakota_new.rst"

Combines multiple restart files into a single restart database.

References write_restart.

Referenced by main().

42.1.3.26 bool Dakota::path_is_absolute (const bfs::path & *p*) `[inline]`

Utilities to manage API differences in BFS_VERSION_2 vs BFS_VERSION_3. Same as path.is_absolute() in latest boost::filesystem

42.1.4 Variable Documentation

42.1.4.1 const char* FIELD_NAMES[ ]

Initial value:

```
{ "numFns", "numVars", "numACV", "numADIV",
                              "numADRV", "numDerivVars", "xC", "xDI",
                              "xDR", "xCLabels", "xDILabels",
                              "xDRLabels", "directFnASV", "directFnASM",
                              "directFnDVV", "directFnDVV_bool",
                              "fnFlag", "gradFlag", "hessFlag",
                              "fnVals",  "fnGrads",  "fnHessians",
                              "fnLabels", "failure", "currEvalId" }
```

fields to pass to Matlab in Dakota structure

Referenced by DirectApplicInterface::DirectApplicInterface(), and DirectApplicInterface::matlab_engine_run().

42.1.4.2 const int NUMBER_OF_FIELDS = 25

number of fields in above structure

Referenced by DirectApplicInterface::DirectApplicInterface(), and DirectApplicInterface::matlab_engine_run().

42.1.4.3 Dakota_funcs DakFuncs0

Initial value:

```
{
        fprintf,
        abort_handler,
        dlsolver_option,
        continuous_lower_bounds1,
        continuous_upper_bounds1,
        nonlinear_ineq_constraint_lower_bounds1,
        nonlinear_ineq_constraint_upper_bounds1,
        nonlinear_eq_constraint_targets1,
        linear_ineq_constraint_lower_bounds1,
        linear_ineq_constraint_upper_bounds1,
        linear_eq_constraint_targets1,
        linear_ineq_constraint_coeffs1,
        linear_eq_constraint_coeffs1,
        ComputeResponses1,
        GetFuncs1,
        GetGrads1,
        GetContVars1,
        SetBestContVars1,
        SetBestDiscVars1,
        SetBestRespFns1,
        Get_Real1,
        Get_Int1,
        Get_Bool1
        }
```

42.1.4.4 char slmap[256] [static]

Initial value:

```
{


0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x
     1d, 0x1e, 0x1f,
0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x
     2d, 0x2e, 0x2f,
0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x
     3d, 0x3e, 0x3f,
0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x
     4d, 0x4e, 0x4f,
0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, '/',  0x
     5d, 0x5e, 0x5f,
0x60, 'A',  'B',  'C',  'D',  'E',  'F',  'G',  'H',  'I',  'J',  'K',  'L',  'M
     ',  'N',  'O',
'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',  'X',  'Y',  'Z',  0x7b, 0x7c, 0x
     7d, 0x7e, 0x7f,
0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x
     8d, 0x8e, 0x8f,
0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x
     9d, 0x9e, 0x9f,
0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xab, 0xac, 0x
     ad, 0xae, 0xaf,
0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0x
     bd, 0xbe, 0xbf,
0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0x
     cd, 0xce, 0xcf,
0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0x
     dd, 0xde, 0xdf,
0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0x
     ed, 0xee, 0xef,
0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0x
     fd, 0xfe, 0xff}
```

42.1.4.5 static KeyWord kw_1 [static]

Initial value:

```
{
                {"active_set_vector",8,0,1,0,1655},
                {"evaluation_cache",8,0,2,0,1657},
                {"restart_file",8,0,3,0,1659}
                }
```

911 distinct keywords (plus 121 aliases)

42.1.4.6 static KeyWord kw_2 () [static]

Initial value:

```
{
```

```
                {"processors_per_analysis",9,0,1,0,1639,0,0.,0.,0.,0,"{Number of
     processors per analysis} http://dakota.sandia.gov/licensing/votd/html-ref/InterfC
     ommands.html#InterfApplicDF"}
                }
```

42.1.4.7 static KeyWord kw_3 () [static]

Initial value:

```
{
                {"abort",8,0,1,1,1645,0,0.,0.,0.,0,"@[CHOOSE failure mitigation]"
     },
                {"continuation",8,0,1,1,1651},
                {"recover",14,0,1,1,1649},
                {"retry",9,0,1,1,1647}
                }
```

42.1.4.8 static KeyWord kw_4 [static]

Initial value:

```
{
                {"copy",8,0,1,0,1633,0,0.,0.,0.,0,"{Copy template files} http://d
     akota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplicSC"},
                {"replace",8,0,2,0,1635,0,0.,0.,0.,0,"{Replace existing files} ht
     tp://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplicSC
     "}
                }
```

42.1.4.9 static KeyWord kw_5 () [static]

Initial value:

```
{
                {"dir_save",0,0,3,0,1626},
                {"dir_tag",0,0,2,0,1624},
                {"directory_save",8,0,3,0,1627,0,0.,0.,0.,0,"{Save work directory
     } http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfAppl
     icSC"},
                {"directory_tag",8,0,2,0,1625,0,0.,0.,0.,0,"{Tag work directory}
     http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplic
     SC"},
                {"named",11,0,1,0,1623,0,0.,0.,0.,0,"{Name of work directory} htt
     p://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplicSC"
     },
                {"template_directory",11,2,4,0,1629,kw_4,0.,0.,0.,0,"{Template di
     rectory} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#Int
     erfApplicSC"},
                {"template_files",15,2,4,0,1631,kw_4,0.,0.,0.,0,"{Template files}
      http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfAppli
     cSC"}
                }
```

42.1.4.10 static KeyWord kw_6 () [static]

Initial value:

```
{
            {"allow_existing_results",8,0,3,0,1611,0,0.,0.,0.,0,"{Allow exist
ing results files} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommand
s.html#InterfApplicSC"},
            {"aprepro",8,0,5,0,1615,0,0.,0.,0.,0,"{Aprepro parameters file fo
rmat} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#Interf
ApplicSC"},
            {"file_save",8,0,7,0,1619,0,0.,0.,0.,0,"{Parameters and results f
ile saving} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#
InterfApplicSC"},
            {"file_tag",8,0,6,0,1617,0,0.,0.,0.,0,"{Parameters and results fi
le tagging} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#
InterfApplicSC"},
            {"parameters_file",11,0,1,0,1607,0,0.,0.,0.,0,"{Parameters file n
ame} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfA
pplicSC"},
            {"results_file",11,0,2,0,1609,0,0.,0.,0.,0,"{Results file name} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplicS
C"},
            {"verbatim",8,0,4,0,1613,0,0.,0.,0.,0,"{Verbatim driver/filter in
vocation syntax} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.
html#InterfApplicSC"},
            {"work_directory",8,7,8,0,1621,kw_5,0.,0.,0.,0,"{Create work dire
ctory} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#Inter
fApplicSC"}
            }
```

42.1.4.11 static KeyWord kw_7 [static]

Initial value:

```
{
            {"analysis_components",15,0,1,0,1597,0,0.,0.,0.,0,"{Additional id
entifiers for use by the analysis_drivers} http://dakota.sandia.gov/licensing/vot
d/html-ref/InterfCommands.html#InterfApplic"},
            {"deactivate",8,3,6,0,1653,kw_1,0.,0.,0.,0,"{Feature deactivation
} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfAppl
ic"},
            {"direct",8,1,4,1,1637,kw_2,0.,0.,0.,0,"[CHOOSE interface type]{D
irect function interface } http://dakota.sandia.gov/licensing/votd/html-ref/Inter
fCommands.html#InterfApplicDF"},
            {"failure_capture",8,4,5,0,1643,kw_3,0.,0.,0.,0,"{Failure capturi
ng} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfAp
plic"},
            {"fork",8,8,4,1,1605,kw_6,0.,0.,0.,0,"@"},
            {"grid",8,0,4,1,1641,0,0.,0.,0.,0,"{Grid interface } http://dakot
a.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplicG"},
            {"input_filter",11,0,2,0,1599,0,0.,0.,0.,0,"{Input filter} http:/
/dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplic"},
            {"output_filter",11,0,3,0,1601,0,0.,0.,0.,0,"{Output filter} http
://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplic"},
            {"system",8,8,4,1,1603,kw_6,0.,0.,0.,0,"{System call interface }
http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfApplic
SC"}
            }
```

42.1.4.12 static KeyWord kw_8 () [static]

Initial value:

```
{
            {"analysis_concurrency",9,0,3,0,1669,0,0.,0.,0.,0,"{Asynchronous
    analysis concurrency} http://dakota.sandia.gov/licensing/votd/html-ref/InterfComm
    ands.html#InterfIndControl"},
            {"evaluation_concurrency",9,0,1,0,1663,0,0.,0.,0.,0,"{Asynchronou
    s evaluation concurrency} http://dakota.sandia.gov/licensing/votd/html-ref/Interf
    Commands.html#InterfIndControl"},
            {"local_evaluation_self_scheduling",8,0,2,0,1665,0,0.,0.,0.,0,"{S
    elf-schedule local evals} http://dakota.sandia.gov/licensing/votd/html-ref/Interf
    Commands.html#InterfIndControl"},
            {"local_evaluation_static_scheduling",8,0,2,0,1667,0,0.,0.,0.,0,"
    {Static-schedule local evals} http://dakota.sandia.gov/licensing/votd/html-ref/In
    terfCommands.html#InterfIndControl"}
            }
```

42.1.4.13 static KeyWord kw_9 () [static]

Initial value:

```
{
            {"algebraic_mappings",11,0,2,0,1593,0,0.,0.,0.,0,"{Algebraic mapp
    ings file} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#I
    nterfAlgebraic"},
            {"analysis_drivers",15,9,3,0,1595,kw_7,0.,0.,0.,0,"{Analysis driv
    ers} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#InterfA
    pplic"},
            {"analysis_self_scheduling",8,0,8,0,1679,0,0.,0.,0.,0,"[CHOOSE an
    alysis sched.]{Self scheduling of analyses} http://dakota.sandia.gov/licensing/vo
    td/html-ref/InterfCommands.html#InterfIndControl"},
            {"analysis_servers",9,0,7,0,1677,0,0.,0.,0.,0,"{Number of analysi
    s servers} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#I
    nterfIndControl"},
            {"analysis_static_scheduling",8,0,8,0,1681,0,0.,0.,0.,0,"{Static
    scheduling of analyses} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCo
    mmands.html#InterfIndControl"},
            {"asynchronous",8,4,4,0,1661,kw_8,0.,0.,0.,0,"{Asynchronous inter
    face usage} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#
    InterfIndControl"},
            {"evaluation_self_scheduling",8,0,6,0,1673,0,0.,0.,0.,0,"[CHOOSE
    evaluation sched.]{Self scheduling of evaluations} http://dakota.sandia.gov/licen
    sing/votd/html-ref/InterfCommands.html#InterfIndControl"},
            {"evaluation_servers",9,0,5,0,1671,0,0.,0.,0.,0,"{Number of evalu
    ation servers} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.ht
    ml#InterfIndControl"},
            {"evaluation_static_scheduling",8,0,6,0,1675,0,0.,0.,0.,0,"{Stati
    c scheduling of evaluations} http://dakota.sandia.gov/licensing/votd/html-ref/Int
    erfCommands.html#InterfIndControl"},
            {"id_interface",11,0,1,0,1591,0,0.,0.,0.,0,"{Interface set identi
    fier} http://dakota.sandia.gov/licensing/votd/html-ref/InterfCommands.html#Interf
    IndControl"}
            }
```

42.1.4.14 static KeyWord kw_10 () [static]

Initial value:

```
{
                {"merit1",8,0,1,1,261,0,0.,0.,0.,0,"[CHOOSE merit function]"},
                {"merit1_smooth",8,0,1,1,263},
                {"merit2",8,0,1,1,265},
                {"merit2_smooth",8,0,1,1,267,0,0.,0.,0.,0,"@"},
                {"merit2_squared",8,0,1,1,269},
                {"merit_max",8,0,1,1,257},
                {"merit_max_smooth",8,0,1,1,259}
                }
```

42.1.4.15 static KeyWord kw_11 () [static]

Initial value:

```
{
                {"blocking",8,0,1,1,251,0,0.,0.,0.,0,"[CHOOSE synchronization]"},

                {"nonblocking",8,0,1,1,253,0,0.,0.,0.,0,"@"}
                }
```

42.1.4.16 static KeyWord kw_12 () [static]

Initial value:

```
{
                {"constraint_penalty",10,0,7,0,271,0,0.,0.,0.,0,"{Constraint pena
     lty} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodA
     PPSDC"},
                {"contraction_factor",10,0,2,0,243,0,0.,0.,0.,0,"{Pattern contrac
     tion factor} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
     #MethodAPPSDC"},
                {"initial_delta",10,0,1,0,241,0,0.,0.,0.,0,"{Initial offset value
     } http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodAPPS
     DC"},
                {"linear_equality_constraint_matrix",14,0,14,0,403,0,0.,0.,0.,0,"
     {Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html
     -ref/MethodCommands.html#MethodIndControl"},
                {"linear_equality_scale_types",15,0,16,0,407,0,0.,0.,0.,0,"{Linea
     r equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Method
     Commands.html#MethodIndControl"},
                {"linear_equality_scales",14,0,17,0,409,0,0.,0.,0.,0,"{Linear equ
     ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
     l#MethodIndControl"},
                {"linear_equality_targets",14,0,15,0,405,0,0.,0.,0.,0,"{Linear eq
     uality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
     tml#MethodIndControl"},
                {"linear_inequality_constraint_matrix",14,0,9,0,393,0,0.,0.,0.,0,
     "{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
     tml-ref/MethodCommands.html#MethodIndControl"},
                {"linear_inequality_lower_bounds",14,0,10,0,395,0,0.,0.,0.,0,"{Li
     near inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Me
     thodCommands.html#MethodIndControl"},
```

```
          {"linear_inequality_scale_types",15,0,12,0,399,0,0.,0.,0.,0,"{Lin
ear inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"linear_inequality_scales",14,0,13,0,401,0,0.,0.,0.,0,"{Linear i
nequality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands
.html#MethodIndControl"},
          {"linear_inequality_upper_bounds",14,0,11,0,397,0,0.,0.,0.,0,"{Li
near inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"merit_function",8,7,6,0,255,kw_10,0.,0.,0.,0,"{Merit function}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodAPPSDC
"},
          {"smoothing_factor",10,0,8,0,273,0,0.,0.,0.,0,"{Smoothing factor}
 http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodAPPSD
C"},
          {"solution_accuracy",2,0,4,0,246},
          {"solution_target",10,0,4,0,247,0,0.,0.,0.,0,"{Solution target} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodAPPSDC"
},
          {"synchronization",8,2,5,0,249,kw_11,0.,0.,0.,0,"{Evaluation sync
hronization} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodAPPSDC"},
          {"threshold_delta",10,0,3,0,245,0,0.,0.,0.,0,"{Threshold for offs
et values} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodAPPSDC"}
          }
```

42.1.4.17 static KeyWord kw_13 () [static]

Initial value:

```
{
          {"emulator_samples",9,0,1,1,859}
          }
```

42.1.4.18 static KeyWord kw_14 () [static]

Initial value:

```
{
          {"adaptive",8,0,1,1,871},
          {"hastings",8,0,1,1,869}
          }
```

42.1.4.19 static KeyWord kw_15 () [static]

Initial value:

```
{
          {"emulator_samples",9,0,1,0,847}
          }
```

42.1.4.20 static KeyWord kw_16 () [static]

Initial value:

```
{
                {"sparse_grid_level",13,0,1,0,851}
                }
```

42.1.4.21 static KeyWord kw_17 () [static]

Initial value:

```
{
                {"sparse_grid_level",13,0,1,0,855}
                }
```

42.1.4.22 static KeyWord kw_18 [static]

Initial value:

```
{
                {"gaussian_process",8,1,1,1,845,kw_15},
                {"gp",0,1,1,1,844,kw_15},
                {"pce",8,1,1,1,849,kw_16},
                {"sc",8,1,1,1,853,kw_17}
                }
```

42.1.4.23 static KeyWord kw_19 [static]

Initial value:

```
{
                {"emulator",8,4,1,0,843,kw_18}
                }
```

42.1.4.24 static KeyWord kw_20 () [static]

Initial value:

```
{
                {"delayed",8,0,1,1,865},
                {"standard",8,0,1,1,863}
                }
```

42.1.4.25 static KeyWord kw_21 () [static]

Initial value:

```
{
                {"mt19937",8,0,1,1,877},
                {"rnum2",8,0,1,1,879}
                }
```

42.1.4.26 static KeyWord kw_22 () [static]

Initial value:

```
{
                {"gpmsa",8,1,1,1,857,kw_13},
                {"likelihood_scale",10,0,7,0,883},
                {"metropolis",8,2,3,0,867,kw_14},
                {"proposal_covariance_scale",10,0,6,0,881},
                {"queso",8,1,1,1,841,kw_19},
                {"rejection",8,2,2,0,861,kw_20},
                {"rng",8,2,5,0,875,kw_21},
                {"samples",9,0,9,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"seed",9,0,8,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
                {"use_derivatives",8,0,4,0,873}
                }
```

42.1.4.27 static KeyWord kw_23 () [static]

Initial value:

```
{
                {"deltas_per_variable",5,0,2,2,1128},
                {"step_vector",14,0,1,1,1127,0,0.,0.,0.,0,"{Step vector} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodPSCPS"},
                {"steps_per_variable",13,0,2,2,1129,0,0.,0.,0.,0,"{Number of step
s per variable} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodPSCPS"}
                }
```

42.1.4.28 static KeyWord kw_24 () [static]

Initial value:

```
{
                {"initial_delta",10,0,5,1,459,0,0.,0.,0.,0,"{Initial offset value
} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOL
IBPS"},
                {"misc_options",15,0,4,0,545,0,0.,0.,0.,0,"{Specify miscellaneous
 options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodSCOLIBDC"},
                {"seed",9,0,2,0,541,0,0.,0.,0.,0,"{Random seed for stochastic pat
```

```
tern search} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodSCOLIBPS"},
          {"show_misc_options",8,0,3,0,543,0,0.,0.,0.,0,"{Show miscellaneou
s options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBDC"},
          {"solution_accuracy",2,0,1,0,538},
          {"solution_target",10,0,1,0,539,0,0.,0.,0.,0,"{Desired solution t
arget} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dSCOLIBDC"},
          {"threshold_delta",10,0,6,2,461,0,0.,0.,0.,0,"{Threshold for offs
et values} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBPS"}
          }
```

42.1.4.29 static KeyWord kw_25 () [static]

Initial value:

```
{
          {"all_dimensions",8,0,1,1,469},
          {"major_dimension",8,0,1,1,467}
          }
```

42.1.4.30 static KeyWord kw_26 () [static]

Initial value:

```
{
          {"constraint_penalty",10,0,6,0,479,0,0.,0.,0.,0,"{Constraint pena
lty} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodS
COLIBDIR"},
          {"division",8,2,1,0,465,kw_25,0.,0.,0.,0,"{Box subdivision approa
ch} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSC
OLIBDIR"},
          {"global_balance_parameter",10,0,2,0,471,0,0.,0.,0.,0,"{Global se
arch balancing parameter} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodSCOLIBDIR"},
          {"local_balance_parameter",10,0,3,0,473,0,0.,0.,0.,0,"{Local sear
ch balancing parameter} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCo
mmands.html#MethodSCOLIBDIR"},
          {"max_boxsize_limit",10,0,4,0,475,0,0.,0.,0.,0,"{Maximum boxsize
limit} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dSCOLIBDIR"},
          {"min_boxsize_limit",10,0,5,0,477,0,0.,0.,0.,0,"{Minimum boxsize
limit} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dSCOLIBDIR"},
          {"misc_options",15,0,10,0,545,0,0.,0.,0.,0,"{Specify miscellaneou
s options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBDC"},
          {"seed",9,0,8,0,541,0,0.,0.,0.,0,"{Random seed for stochastic pat
tern search} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodSCOLIBPS"},
          {"show_misc_options",8,0,9,0,543,0,0.,0.,0.,0,"{Show miscellaneou
s options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBDC"},
          {"solution_accuracy",2,0,7,0,538},
          {"solution_target",10,0,7,0,539,0,0.,0.,0.,0,"{Desired solution t
```

```
arget} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dSCOLIBDC"}
          }
```

42.1.4.31 static KeyWord kw_27 () [static]

Initial value:

```
{
               {"blend",8,0,1,1,515},
               {"two_point",8,0,1,1,513},
               {"uniform",8,0,1,1,517}
               }
```

42.1.4.32 static KeyWord kw_28 () [static]

Initial value:

```
{
               {"linear_rank",8,0,1,1,495},
               {"merit_function",8,0,1,1,497}
               }
```

42.1.4.33 static KeyWord kw_29 [static]

Initial value:

```
{
               {"flat_file",11,0,1,1,491},
               {"simple_random",8,0,1,1,487},
               {"unique_random",8,0,1,1,489}
               }
```

42.1.4.34 static KeyWord kw_30 () [static]

Initial value:

```
{
               {"mutation_range",9,0,2,0,533,0,0.,0.,0.,0,"{Mutation range} http
:/dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"}
,
               {"mutation_scale",10,0,1,0,531,0,0.,0.,0.,0,"{Mutation scale} htt
p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"
}
               }
```

42.1.4.35 static KeyWord kw_31 () [static]

Initial value:

```
{
                {"non_adaptive",8,0,2,0,535,0,0.,0.,0.,0,"{Non-adaptive mutation
     flag} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
     SCOLIBEA"},
                {"offset_cauchy",8,2,1,1,527,kw_30},
                {"offset_normal",8,2,1,1,525,kw_30},
                {"offset_uniform",8,2,1,1,529,kw_30},
                {"replace_uniform",8,0,1,1,523}
                }
```

42.1.4.36 static KeyWord kw_32 () [static]

Initial value:

```
{
                {"chc",9,0,1,1,503,0,0.,0.,0.,0,"{CHC replacement type} http://da
     kota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"},
                {"elitist",9,0,1,1,505,0,0.,0.,0.,0,"{Elitist replacement type} h
     ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBE
     A"},
                {"new_solutions_generated",9,0,2,0,507,0,0.,0.,0.,0,"{New solutio
     ns generated} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
     l#MethodSCOLIBEA"},
                {"random",9,0,1,1,501,0,0.,0.,0.,0,"{Random replacement type} htt
     p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"
     }
                }
```

42.1.4.37 static KeyWord kw_33 () [static]

Initial value:

```
{
                {"constraint_penalty",10,0,9,0,537},
                {"crossover_rate",10,0,5,0,509,0,0.,0.,0.,0,"{Crossover rate} htt
     p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"
     },
                {"crossover_type",8,3,6,0,511,kw_27,0.,0.,0.,0,"{Crossover type}
     http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIB
     EA"},
                {"fitness_type",8,2,3,0,493,kw_28,0.,0.,0.,0,"{Fitness type} http
     ://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"}
     ,
                {"initialization_type",8,3,2,0,485,kw_29,0.,0.,0.,0,"{Initializat
     ion type} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
     thodSCOLIBEA"},
                {"misc_options",15,0,13,0,545,0,0.,0.,0.,0,"{Specify miscellaneou
     s options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
     ethodSCOLIBDC"},
                {"mutation_rate",10,0,7,0,519,0,0.,0.,0.,0,"{Mutation rate} http:
     //dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA"},
```

```
            {"mutation_type",8,5,8,0,521,kw_31,0.,0.,0.,0,"{Mutation type} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBEA
"},
            {"population_size",9,0,1,0,483,0,0.,0.,0.,0,"{Number of populatio
n members} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBEA"},
            {"replacement_type",8,4,4,0,499,kw_32,0.,0.,0.,0,"{Replacement ty
pe} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSC
OLIBEA"},
            {"seed",9,0,11,0,541,0,0.,0.,0.,0,"{Random seed for stochastic pa
ttern search} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodSCOLIBPS"},
            {"show_misc_options",8,0,12,0,543,0,0.,0.,0.,0,"{Show miscellaneo
us options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#
MethodSCOLIBDC"},
            {"solution_accuracy",2,0,10,0,538},
            {"solution_target",10,0,10,0,539,0,0.,0.,0.,0,"{Desired solution
target} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Meth
odSCOLIBDC"}
            }
```

42.1.4.38 static KeyWord kw_34 () [static]

Initial value:

```
{
            {"adaptive_pattern",8,0,1,1,433},
            {"basic_pattern",8,0,1,1,435},
            {"multi_step",8,0,1,1,431}
            }
```

42.1.4.39 static KeyWord kw_35 () [static]

Initial value:

```
{
            {"coordinate",8,0,1,1,421},
            {"simplex",8,0,1,1,423}
            }
```

42.1.4.40 static KeyWord kw_36 () [static]

Initial value:

```
{
            {"blocking",8,0,1,1,439},
            {"nonblocking",8,0,1,1,441}
            }
```

42.1.4.41 static KeyWord kw_37 () [static]

Initial value:

```
{
            {"constant_penalty",8,0,1,0,413,0,0.,0.,0.,0,"{Control of dynamic
 penalty} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodSCOLIBPS"},
            {"constraint_penalty",10,0,16,0,455,0,0.,0.,0.,0,"{Constraint pen
alty} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
SCOLIBPS"},
            {"contraction_factor",10,0,15,0,453,0,0.,0.,0.,0,"{Pattern contra
ction factor} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodSCOLIBPS"},
            {"expand_after_success",9,0,3,0,417,0,0.,0.,0.,0,"{Number of cons
ecutive improvements before expansion} http://dakota.sandia.gov/licensing/votd/ht
ml-ref/MethodCommands.html#MethodSCOLIBPS"},
            {"exploratory_moves",8,3,7,0,429,kw_34,0.,0.,0.,0,"{Exploratory m
oves selection} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodSCOLIBPS"},
            {"initial_delta",10,0,13,1,459,0,0.,0.,0.,0,"{Initial offset valu
e} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCO
LIBPS"},
            {"misc_options",15,0,12,0,545,0,0.,0.,0.,0,"{Specify miscellaneou
s options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBDC"},
            {"no_expansion",8,0,2,0,415,0,0.,0.,0.,0,"{No expansion flag} htt
p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBPS"
},
            {"pattern_basis",8,2,4,0,419,kw_35,0.,0.,0.,0,"{Pattern basis sel
ection} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Meth
odSCOLIBPS"},
            {"seed",9,0,10,0,541,0,0.,0.,0.,0,"{Random seed for stochastic pa
ttern search} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodSCOLIBPS"},
            {"show_misc_options",8,0,11,0,543,0,0.,0.,0.,0,"{Show miscellaneo
us options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#
MethodSCOLIBDC"},
            {"solution_accuracy",2,0,9,0,538},
            {"solution_target",10,0,9,0,539,0,0.,0.,0.,0,"{Desired solution t
arget} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dSCOLIBDC"},
            {"stochastic",8,0,5,0,425,0,0.,0.,0.,0,"{Stochastic pattern searc
h} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCO
LIBPS"},
            {"synchronization",8,2,8,0,437,kw_36,0.,0.,0.,0,"{Evaluation sync
hronization} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodSCOLIBPS"},
            {"threshold_delta",10,0,14,2,461,0,0.,0.,0.,0,"{Threshold for off
set values} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#
MethodSCOLIBPS"},
            {"total_pattern_size",9,0,6,0,427,0,0.,0.,0.,0,"{Total number of
points in pattern} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommand
s.html#MethodSCOLIBPS"}
            }
```

42.1.4.42 static KeyWord kw_38 () [static]

Initial value:

```
{
            {"constant_penalty",8,0,4,0,451,0,0.,0.,0.,0,"{Control of dynamic
 penalty} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodSCOLIBSW"},
```

```
          {"constraint_penalty",10,0,12,0,455,0,0.,0.,0.,0,"{Constraint pen
alty} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
SCOLIBPS"},
          {"contract_after_failure",9,0,1,0,445,0,0.,0.,0.,0,"{Number of co
nsecutive failures before contraction} http://dakota.sandia.gov/licensing/votd/ht
ml-ref/MethodCommands.html#MethodSCOLIBSW"},
          {"contraction_factor",10,0,11,0,453,0,0.,0.,0.,0,"{Pattern contra
ction factor} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodSCOLIBPS"},
          {"expand_after_success",9,0,3,0,449,0,0.,0.,0.,0,"{Number of cons
ecutive improvements before expansion} http://dakota.sandia.gov/licensing/votd/ht
ml-ref/MethodCommands.html#MethodSCOLIBSW"},
          {"initial_delta",10,0,9,1,459,0,0.,0.,0.,0,"{Initial offset value
} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOL
IBPS"},
          {"misc_options",15,0,8,0,545,0,0.,0.,0.,0,"{Specify miscellaneous
 options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodSCOLIBDC"},
          {"no_expansion",8,0,2,0,447,0,0.,0.,0.,0,"{No expansion flag} htt
p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSCOLIBSW"
},
          {"seed",9,0,6,0,541,0,0.,0.,0.,0,"{Random seed for stochastic pat
tern search} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodSCOLIBPS"},
          {"show_misc_options",8,0,7,0,543,0,0.,0.,0.,0,"{Show miscellaneou
s options} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodSCOLIBDC"},
          {"solution_accuracy",2,0,5,0,538},
          {"solution_target",10,0,5,0,539,0,0.,0.,0.,0,"{Desired solution t
arget} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dSCOLIBDC"},
          {"threshold_delta",10,0,10,2,461,0,0.,0.,0.,0,"{Threshold for off
set values} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#
MethodSCOLIBPS"}
          }
```

42.1.4.43 static KeyWord kw_39 () [static]

Initial value:

```
{
          {"linear_equality_constraint_matrix",14,0,14,0,403,0,0.,0.,0.,0,"
{Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html
-ref/MethodCommands.html#MethodIndControl"},
          {"linear_equality_scale_types",15,0,16,0,407,0,0.,0.,0.,0,"{Linea
r equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodIndControl"},
          {"linear_equality_scales",14,0,17,0,409,0,0.,0.,0.,0,"{Linear equ
ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodIndControl"},
          {"linear_equality_targets",14,0,15,0,405,0,0.,0.,0.,0,"{Linear eq
uality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodIndControl"},
          {"linear_inequality_constraint_matrix",14,0,9,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
          {"linear_inequality_lower_bounds",14,0,10,0,395,0,0.,0.,0.,0,"{Li
near inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"linear_inequality_scale_types",15,0,12,0,399,0,0.,0.,0.,0,"{Lin
```

```
ear inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"linear_inequality_scales",14,0,13,0,401,0,0.,0.,0.,0,"{Linear i
nequality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands
.html#MethodIndControl"},
          {"linear_inequality_upper_bounds",14,0,11,0,397,0,0.,0.,0.,0,"{Li
near inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"}
          }
```

42.1.4.44 static KeyWord kw_40 () [static]

Initial value:

```
{
          {"drop_tolerance",10,0,1,0,907}
          }
```

42.1.4.45 static KeyWord kw_41 () [static]

Initial value:

```
{
          {"box_behnken",8,0,1,1,897,0,0.,0.,0.,0,"[CHOOSE DACE type]"},
          {"central_composite",8,0,1,1,899},
          {"fixed_seed",8,0,5,0,909,0,0.,0.,0.,0,"{Fixed seed flag} http://
dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodDDACE"},
          {"grid",8,0,1,1,887},
          {"lhs",8,0,1,1,893},
          {"main_effects",8,0,2,0,901,0,0.,0.,0.,0,"{Main effects} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodDDACE"},
          {"oa_lhs",8,0,1,1,895},
          {"oas",8,0,1,1,891},
          {"quality_metrics",8,0,3,0,903,0,0.,0.,0.,0,"{Quality metrics} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodDDACE"},

          {"random",8,0,1,1,889},
          {"samples",9,0,8,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
          {"seed",9,0,7,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
          {"symbols",9,0,6,0,911,0,0.,0.,0.,0,"{Number of symbols} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodDDACE"},
          {"variance_based_decomp",8,1,4,0,905,kw_40,0.,0.,0.,0,"{Variance
based decomposition} http://dakota.sandia.gov/licensing/votd/html-ref/MethodComma
nds.html#MethodDDACE"}
          }
```

42.1.4.46 static KeyWord kw_42 () [static]

Initial value:

```
{
          {"maximize",8,0,1,1,177},
          {"minimize",8,0,1,1,175}
          }
```

42.1.4.47 static KeyWord kw_43 () [static]

Initial value:

```
{
                {"bfgs",8,0,1,1,167},
                {"frcg",8,0,1,1,163},
                {"linear_equality_constraint_matrix",14,0,7,0,403,0,0.,0.,0.,0,"{
     Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
     ref/MethodCommands.html#MethodIndControl"},
                {"linear_equality_scale_types",15,0,9,0,407,0,0.,0.,0.,0,"{Linear
      equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
     ommands.html#MethodIndControl"},
                {"linear_equality_scales",14,0,10,0,409,0,0.,0.,0.,0,"{Linear equ
     ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
     l#MethodIndControl"},
                {"linear_equality_targets",14,0,8,0,405,0,0.,0.,0.,0,"{Linear equ
     ality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.ht
     ml#MethodIndControl"},
                {"linear_inequality_constraint_matrix",14,0,2,0,393,0,0.,0.,0.,0,
     "{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
     tml-ref/MethodCommands.html#MethodIndControl"},
                {"linear_inequality_lower_bounds",14,0,3,0,395,0,0.,0.,0.,0,"{Lin
     ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linear_inequality_scale_types",15,0,5,0,399,0,0.,0.,0.,0,"{Line
     ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linear_inequality_scales",14,0,6,0,401,0,0.,0.,0.,0,"{Linear in
     equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
     html#MethodIndControl"},
                {"linear_inequality_upper_bounds",14,0,4,0,397,0,0.,0.,0.,0,"{Lin
     ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"mmfd",8,0,1,1,165},
                {"optimization_type",8,2,11,0,173,kw_42,0.,0.,0.,0,"{Optimization
      type} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
     dDOTDC"},
                {"slp",8,0,1,1,169},
                {"sqp",8,0,1,1,171}
                }
```

42.1.4.48 static KeyWord kw_44 () [static]

Initial value:

```
{
                {"linear_equality_constraint_matrix",14,0,6,0,403,0,0.,0.,0.,0,"{
     Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
     ref/MethodCommands.html#MethodIndControl"},
                {"linear_equality_scale_types",15,0,8,0,407,0,0.,0.,0.,0,"{Linear
      equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
     ommands.html#MethodIndControl"},
                {"linear_equality_scales",14,0,9,0,409,0,0.,0.,0.,0,"{Linear equa
     lity scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
     #MethodIndControl"},
                {"linear_equality_targets",14,0,7,0,405,0,0.,0.,0.,0,"{Linear equ
     ality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.ht
     ml#MethodIndControl"},
```

```
          {"linear_inequality_constraint_matrix",14,0,1,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
          {"linear_inequality_lower_bounds",14,0,2,0,395,0,0.,0.,0.,0,"{Lin
ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
          {"linear_inequality_scale_types",15,0,4,0,399,0,0.,0.,0.,0,"{Line
ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
          {"linear_inequality_scales",14,0,5,0,401,0,0.,0.,0.,0,"{Linear in
equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodIndControl"},
          {"linear_inequality_upper_bounds",14,0,3,0,397,0,0.,0.,0.,0,"{Lin
ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
          {"optimization_type",8,2,10,0,173,kw_42,0.,0.,0.,0,"{Optimization
 type} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dDOTDC"}
          }
```

42.1.4.49 static KeyWord kw_45 () [static]

Initial value:

```
{
          {"dakota",8,0,1,1,585},
          {"surfpack",8,0,1,1,583}
          }
```

42.1.4.50 static KeyWord kw_46 () [static]

Initial value:

```
{
          {"gaussian_process",8,2,1,0,581,kw_45,0.,0.,0.,0,"{GP selection}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
          {"kriging",0,2,1,0,580,kw_45},
          {"seed",9,0,3,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
          {"use_derivatives",8,0,2,0,587,0,0.,0.,0.,0,"{Derivative usage} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"}
          }
```

42.1.4.51 static KeyWord kw_47 () [static]

Initial value:

```
{
          {"grid",8,0,1,1,927,0,0.,0.,0.,0,"[CHOOSE trial type]"},
          {"halton",8,0,1,1,929},
          {"random",8,0,1,1,931,0,0.,0.,0.,0,"@"}
          }
```

42.1.4.52 static KeyWord kw_48 () [static]

Initial value:

```
{
                {"drop_tolerance",10,0,1,0,921}
                }
```

42.1.4.53 static KeyWord kw_49 () [static]

Initial value:

```
{
                {"fixed_seed",8,0,4,0,923,0,0.,0.,0.,0,"{Fixed seed flag} http://
     dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE"},
                {"latinize",8,0,1,0,915,0,0.,0.,0.,0,"{Latinization of samples} h
     ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE
     "},
                {"num_trials",9,0,6,0,933,0,0.,0.,0.,0,"{Number of trials  } http
     ://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE"},

                {"quality_metrics",8,0,2,0,917,0,0.,0.,0.,0,"{Quality metrics} ht
     tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE"
     },
                {"samples",9,0,8,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
     akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"seed",9,0,7,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
     dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
                {"trial_type",8,3,5,0,925,kw_47,0.,0.,0.,0,"{Trial type} http://d
     akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE"},
                {"variance_based_decomp",8,1,3,0,919,kw_48,0.,0.,0.,0,"{Variance
     based decomposition} http://dakota.sandia.gov/licensing/votd/html-ref/MethodComma
     nds.html#MethodFSUDACE"}
                }
```

42.1.4.54 static KeyWord kw_50 () [static]

Initial value:

```
{
                {"drop_tolerance",10,0,1,0,1101}
                }
```

42.1.4.55 static KeyWord kw_51 () [static]

Initial value:

```
{
                {"fixed_sequence",8,0,6,0,1105,0,0.,0.,0.,0,"{Fixed sequence flag
     } http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUD
     ACE"},
                {"halton",8,0,1,1,1091,0,0.,0.,0.,0,"[CHOOSE sequence type]"},
                {"hammersley",8,0,1,1,1093},
```

```
          {"latinize",8,0,2,0,1095,0,0.,0.,0.,0,"{Latinization of samples}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDAC
E"},
          {"prime_base",13,0,9,0,1111,0,0.,0.,0.,0,"{Prime bases for sequen
ces} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodF
SUDACE"},
          {"quality_metrics",8,0,3,0,1097,0,0.,0.,0.,0,"{Quality metrics} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE
"},
          {"samples",9,0,5,0,1103,0,0.,0.,0.,0,"{Number of samples} http://
dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodFSUDACE"},
          {"sequence_leap",13,0,8,0,1109,0,0.,0.,0.,0,"{Sequence leaping in
dices} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Metho
dFSUDACE"},
          {"sequence_start",13,0,7,0,1107,0,0.,0.,0.,0,"{Sequence starting
indices} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Met
hodFSUDACE"},
          {"variance_based_decomp",8,1,4,0,1099,kw_50,0.,0.,0.,0,"{Variance
 based decomposition} http://dakota.sandia.gov/licensing/votd/html-ref/MethodComm
ands.html#MethodFSUDACE"}
          }
```

42.1.4.56 static KeyWord kw_52 () [static]

Initial value:

```
{
          {"complementary",8,0,1,1,803},
          {"cumulative",8,0,1,1,801}
          }
```

42.1.4.57 static KeyWord kw_53 () [static]

Initial value:

```
{
          {"num_gen_reliability_levels",13,0,1,0,811,0,0.,0.,0.,0,"{Number
of generalized reliability levels} http://dakota.sandia.gov/licensing/votd/html-r
ef/MethodCommands.html#MethodNonD"}
          }
```

42.1.4.58 static KeyWord kw_54 () [static]

Initial value:

```
{
          {"num_probability_levels",13,0,1,0,807,0,0.,0.,0.,0,"{Number of p
robability levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommand
s.html#MethodNonD"}
          }
```

42.1.4.59 static KeyWord kw_55 () [static]

Initial value:

```
{
                {"mt19937",8,0,1,1,815},
                {"rnum2",8,0,1,1,817}
                }
```

42.1.4.60 static KeyWord kw_56 () [static]

Initial value:

```
{
                {"dakota",8,0,1,1,783},
                {"surfpack",8,0,1,1,781}
                }
```

42.1.4.61 static KeyWord kw_57 () [static]

Initial value:

```
{
                {"gaussian_process",8,2,1,0,779,kw_56},
                {"kriging",0,2,1,0,778,kw_56},
                {"use_derivatives",8,0,2,0,785}
                }
```

42.1.4.62 static KeyWord kw_58 () [static]

Initial value:

```
{
                {"gen_reliabilities",8,0,1,1,797},
                {"probabilities",8,0,1,1,795}
                }
```

42.1.4.63 static KeyWord kw_59 () [static]

Initial value:

```
{
                {"compute",8,2,2,0,793,kw_58},
                {"num_response_levels",13,0,1,0,791}
                }
```

42.1.4.64 static KeyWord kw_60 () [static]

Initial value:

```
{
                {"distribution",8,2,5,0,799,kw_52,0.,0.,0.,0,"{Distribution type}
 http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonD"
},
                {"ego",8,3,1,0,777,kw_57},
                {"gen_reliability_levels",14,1,7,0,809,kw_53,0.,0.,0.,0,"{General
ized reliability levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
ommands.html#MethodNonD"},
                {"lhs",8,0,1,0,787},
                {"probability_levels",14,1,6,0,805,kw_54,0.,0.,0.,0,"{Probability
 levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Met
hodNonD"},
                {"response_levels",14,2,2,0,789,kw_59},
                {"rng",8,2,8,0,813,kw_55,0.,0.,0.,0,"{Random number generator} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"}
,
                {"samples",9,0,4,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"seed",9,0,3,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"}
                }
```

42.1.4.65 static KeyWord kw_61 () [static]

Initial value:

```
{
                {"dakota",8,0,1,1,827},
                {"surfpack",8,0,1,1,825}
                }
```

42.1.4.66 static KeyWord kw_62 () [static]

Initial value:

```
{
                {"gaussian_process",8,2,1,0,823,kw_61,0.,0.,0.,0,"{EGO GP selecti
on} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNo
nDGlobalIntervalEst"},
                {"kriging",0,2,1,0,822,kw_61},
                {"use_derivatives",8,0,2,0,829,0,0.,0.,0.,0,"{Derivative usage} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDGlo
balIntervalEst"}
                }
```

42.1.4.67 static KeyWord kw_63 () [static]

Initial value:

```
{
                {"mt19937",8,0,1,1,835},
                {"rnum2",8,0,1,1,837}
                }
```

42.1.4.68 static KeyWord kw_64 () [static]

Initial value:

```
{
                {"ego",8,3,1,0,821,kw_62},
                {"lhs",8,0,1,0,831},
                {"rng",8,2,2,0,833,kw_63,0.,0.,0.,0,"{Random seed generator} http
    ://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDGlobal
    IntervalEst"},
                {"samples",9,0,4,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
    akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"seed",9,0,3,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
    dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"}
                }
```

42.1.4.69 static KeyWord kw_65 () [static]

Initial value:

```
{
                {"complementary",8,0,1,1,1079},
                {"cumulative",8,0,1,1,1077}
                }
```

42.1.4.70 static KeyWord kw_66 () [static]

Initial value:

```
{
                {"num_gen_reliability_levels",13,0,1,0,1087}
                }
```

42.1.4.71 static KeyWord kw_67 () [static]

Initial value:

```
{
                {"num_probability_levels",13,0,1,0,1083}
                }
```

42.1.4.72 static KeyWord kw_68 () [static]

Initial value:

```
{
                {"gen_reliabilities",8,0,1,1,1073},
                {"probabilities",8,0,1,1,1071}
                }
```

42.1.4.73 static KeyWord kw_69 () [static]

Initial value:

```
{
                {"compute",8,2,2,0,1069,kw_68},
                {"num_response_levels",13,0,1,0,1067}
                }
```

42.1.4.74 static KeyWord kw_70 () [static]

Initial value:

```
{
                {"mt19937",8,0,1,1,1061},
                {"rnum2",8,0,1,1,1063}
                }
```

42.1.4.75 static KeyWord kw_71 () [static]

Initial value:

```
{
                {"dakota",8,0,1,0,1051},
                {"surfpack",8,0,1,0,1049}
                }
```

42.1.4.76 static KeyWord kw_72 () [static]

Initial value:

```
{
                {"all_variables",8,0,2,0,1053,0,0.,0.,0.,0,"{All variables flag}
     http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDGl
     obalRel"},
                {"distribution",8,2,7,0,1075,kw_65},
                {"gen_reliability_levels",14,1,9,0,1085,kw_66},
                {"probability_levels",14,1,8,0,1081,kw_67},
                {"response_levels",14,2,6,0,1065,kw_69},
                {"rng",8,2,5,0,1059,kw_70},
                {"seed",9,0,4,0,1057,0,0.,0.,0.,0,"{Refinement seed} http://dakot
```

```
a.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDLocalRel"},
          {"u_gaussian_process",8,2,1,1,1047,kw_71},
          {"u_kriging",0,0,1,1,1046},
          {"use_derivatives",8,0,3,0,1055,0,0.,0.,0.,0,"{Derivative usage}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDGl
obalRel"},
          {"x_gaussian_process",8,2,1,1,1045,kw_71},
          {"x_kriging",0,2,1,1,1044,kw_71}
          }
```

42.1.4.77 static KeyWord kw_73 () [static]

Initial value:

```
{
          {"gen_reliabilities",8,0,1,1,773},
          {"probabilities",8,0,1,1,771}
          }
```

42.1.4.78 static KeyWord kw_74 () [static]

Initial value:

```
{
          {"compute",8,2,2,0,769,kw_73},
          {"num_response_levels",13,0,1,0,767}
          }
```

42.1.4.79 static KeyWord kw_75 () [static]

Initial value:

```
{
          {"distribution",8,2,4,0,799,kw_52,0.,0.,0.,0,"{Distribution type}
 http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonD"
},
          {"gen_reliability_levels",14,1,6,0,809,kw_53,0.,0.,0.,0,"{General
ized reliability levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
ommands.html#MethodNonD"},
          {"probability_levels",14,1,5,0,805,kw_54,0.,0.,0.,0,"{Probability
 levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Met
hodNonD"},
          {"response_levels",14,2,1,0,765,kw_74},
          {"rng",8,2,7,0,813,kw_55,0.,0.,0.,0,"{Random number generator} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"}
,
          {"samples",9,0,3,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
          {"seed",9,0,2,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"}
          }
```

42.1.4.80 static KeyWord kw_76 () [static]

Initial value:

```
{
                {"list_of_points",14,0,1,1,1123,0,0.,0.,0.,0,"{List of points to
     evaluate} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
     thodPSLPS"}
                }
```

42.1.4.81 static KeyWord kw_77 () [static]

Initial value:

```
{
                {"complementary",8,0,1,1,979},
                {"cumulative",8,0,1,1,977}
                }
```

42.1.4.82 static KeyWord kw_78 () [static]

Initial value:

```
{
                {"num_gen_reliability_levels",13,0,1,0,973}
                }
```

42.1.4.83 static KeyWord kw_79 () [static]

Initial value:

```
{
                {"num_probability_levels",13,0,1,0,969}
                }
```

42.1.4.84 static KeyWord kw_80 () [static]

Initial value:

```
{
                {"gen_reliabilities",8,0,1,1,965},
                {"probabilities",8,0,1,1,963}
                }
```

42.1.4.85 static KeyWord kw_81 () [static]

Initial value:

```
{
                {"compute",8,2,2,0,961,kw_80},
                {"num_response_levels",13,0,1,0,959}
                }
```

42.1.4.86 static KeyWord kw_82 () [static]

Initial value:

```
{
                {"distribution",8,2,5,0,975,kw_77},
                {"gen_reliability_levels",14,1,4,0,971,kw_78},
                {"nip",8,0,1,0,955},
                {"probability_levels",14,1,3,0,967,kw_79},
                {"response_levels",14,2,2,0,957,kw_81},
                {"sqp",8,0,1,0,953}
                }
```

42.1.4.87 static KeyWord kw_83 () [static]

Initial value:

```
{
                {"nip",8,0,1,0,985},
                {"sqp",8,0,1,0,983}
                }
```

42.1.4.88 static KeyWord kw_84 () [static]

Initial value:

```
{
                {"adapt_import",8,0,1,1,1019},
                {"import",8,0,1,1,1017},
                {"mm_adapt_import",8,0,1,1,1021},
                {"samples",9,0,2,0,1023,0,0.,0.,0.,0,"{Refinement samples} http:/
     /dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDLocalRel
     "},
                {"seed",9,0,3,0,1025,0,0.,0.,0.,0,"{Random seed} http://dakota.sa
     ndia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"}
                }
```

42.1.4.89 static KeyWord kw_85 () [static]

Initial value:

```
{
                {"first_order",8,0,1,1,1011},
                {"sample_refinement",8,5,2,0,1015,kw_84},
                {"second_order",8,0,1,1,1013}
                }
```

42.1.4.90 static KeyWord kw_86 () [static]

Initial value:

```
{
                {"nip",8,0,2,0,1007},
                {"no_approx",8,0,1,1,1003},
                {"sqp",8,0,2,0,1005},
                {"u_taylor_mean",8,0,1,1,993},
                {"u_taylor_mpp",8,0,1,1,997},
                {"u_two_point",8,0,1,1,1001},
                {"x_taylor_mean",8,0,1,1,991},
                {"x_taylor_mpp",8,0,1,1,995},
                {"x_two_point",8,0,1,1,999}
                }
```

42.1.4.91 static KeyWord kw_87 () [static]

Initial value:

```
{
                {"num_reliability_levels",13,0,1,0,1041}
                }
```

42.1.4.92 static KeyWord kw_88 () [static]

Initial value:

```
{
                {"gen_reliabilities",8,0,1,1,1037},
                {"probabilities",8,0,1,1,1033},
                {"reliabilities",8,0,1,1,1035}
                }
```

42.1.4.93 static KeyWord kw_89 () [static]

Initial value:

```
{
                {"compute",8,3,2,0,1031,kw_88},
                {"num_response_levels",13,0,1,0,1029}
                }
```

42.1.4.94 static KeyWord kw_90 () [static]

Initial value:

```
{
                {"distribution",8,2,5,0,1075,kw_65},
                {"gen_reliability_levels",14,1,7,0,1085,kw_66},
                {"integration",8,3,2,0,1009,kw_85,0.,0.,0.,0,"{Integration method
     } http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonD
     LocalRel"},
                {"mpp_search",8,9,1,0,989,kw_86,0.,0.,0.,0,"{MPP search type} htt
     p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDLocal
     Rel"},
                {"probability_levels",14,1,6,0,1081,kw_67},
                {"reliability_levels",14,1,4,0,1039,kw_87},
                {"response_levels",14,2,3,0,1027,kw_89}
                }
```

42.1.4.95 static KeyWord kw_91 () [static]

Initial value:

```
{
                {"num_offspring",0x19,0,2,0,371,0,0.,0.,0.,0,"{Number of offsprin
     g in random shuffle crossover} http://dakota.sandia.gov/licensing/votd/html-ref/M
     ethodCommands.html#MethodJEGADC"},
                {"num_parents",0x19,0,1,0,369,0,0.,0.,0.,0,"{Number of parents in
      random shuffle crossover} http://dakota.sandia.gov/licensing/votd/html-ref/Metho
     dCommands.html#MethodJEGADC"}
                }
```

42.1.4.96 static KeyWord kw_92 () [static]

Initial value:

```
{
                {"crossover_rate",10,0,2,0,373,0,0.,0.,0.,0,"{Crossover rate} htt
     p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"},

                {"multi_point_binary",9,0,1,1,361,0,0.,0.,0.,0,"{Multi point bina
     ry crossover} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
     l#MethodJEGADC"},
                {"multi_point_parameterized_binary",9,0,1,1,363,0,0.,0.,0.,0,"{Mu
     lti point parameterized binary crossover} http://dakota.sandia.gov/licensing/votd
     /html-ref/MethodCommands.html#MethodJEGADC"},
                {"multi_point_real",9,0,1,1,365,0,0.,0.,0.,0,"{Multi point real c
     rossover} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
     thodJEGADC"},
                {"shuffle_random",8,2,1,1,367,kw_91,0.,0.,0.,0,"{Random shuffle c
     rossover} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
     thodJEGADC"}
                }
```

42.1.4.97 static KeyWord kw_93 () [static]

Initial value:

```
{
                {"flat_file",11,0,1,1,357},
                {"simple_random",8,0,1,1,353},
                {"unique_random",8,0,1,1,355}
                }
```

42.1.4.98 static KeyWord kw_94 () [static]

Initial value:

```
{
                {"mutation_scale",10,0,1,0,387,0,0.,0.,0.,0,"{Mutation scale} htt
     p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"}
                }
```

42.1.4.99 static KeyWord kw_95 () [static]

Initial value:

```
{
                {"bit_random",8,0,1,1,377},
                {"mutation_rate",10,0,2,0,389,0,0.,0.,0.,0,"{Mutation rate} http:
     //dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"},
                {"offset_cauchy",8,1,1,1,383,kw_94},
                {"offset_normal",8,1,1,1,381,kw_94},
                {"offset_uniform",8,1,1,1,385,kw_94},
                {"replace_uniform",8,0,1,1,379}
                }
```

42.1.4.100 static KeyWord kw_96 () [static]

Initial value:

```
{
                {"metric_tracker",8,0,1,1,303,0,0.,0.,0.,0,"{Convergence type} ht
     tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGAMOGA
     "},
                {"num_generations",0x29,0,3,0,307,0,0.,0.,0.,0,"{Number generatio
     ns for metric_tracker converger} http://dakota.sandia.gov/licensing/votd/html-ref
     /MethodCommands.html#MethodJEGAMOGA"},
                {"percent_change",10,0,2,0,305,0,0.,0.,0.,0,"{Percent change limi
     t for metric_tracker converger} http://dakota.sandia.gov/licensing/votd/html-ref/
     MethodCommands.html#MethodJEGAMOGA"}
                }
```

42.1.4.101 static KeyWord kw_97 () [static]

Initial value:

```
{
                {"domination_count",8,0,1,1,281},
                {"layer_rank",8,0,1,1,279}
                }
```

42.1.4.102 static KeyWord kw_98 () [static]

Initial value:

```
{
                {"distance",14,0,1,1,299},
                {"radial",14,0,1,1,297}
                }
```

42.1.4.103 static KeyWord kw_99 () [static]

Initial value:

```
{
                {"orthogonal_distance",14,0,1,1,311,0,0.,0.,0.,0,"{Post_processor
 distance} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodJEGAMOGA"}
                }
```

42.1.4.104 static KeyWord kw_100 () [static]

Initial value:

```
{
                {"shrinkage_fraction",10,0,1,0,293},
                {"shrinkage_percentage",2,0,1,0,292}
                }
```

42.1.4.105 static KeyWord kw_101 () [static]

Initial value:

```
{
                {"below_limit",10,2,1,1,291,kw_100,0.,0.,0.,0,"{Below limit selec
tion} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
JEGADC"},
                {"elitist",8,0,1,1,285},
                {"roulette_wheel",8,0,1,1,287},
                {"unique_roulette_wheel",8,0,1,1,289}
                }
```

42.1.4.106 static KeyWord kw_102 () [static]

Initial value:

```
{
                {"convergence_type",8,3,4,0,301,kw_96},
                {"crossover_type",8,5,19,0,359,kw_92,0.,0.,0.,0,"{Crossover type}
 http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGAD
C"},
                {"fitness_type",8,2,1,0,277,kw_97,0.,0.,0.,0,"{Fitness type} http
://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGAMOGA"}
,
                {"initialization_type",8,3,18,0,351,kw_93,0.,0.,0.,0,"{Initializa
tion type} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodJEGADC"},
                {"linear_equality_constraint_matrix",14,0,11,0,403,0,0.,0.,0.,0,"
{Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html
-ref/MethodCommands.html#MethodIndControl"},
                {"linear_equality_scale_types",15,0,13,0,407,0,0.,0.,0.,0,"{Linea
r equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodIndControl"},
                {"linear_equality_scales",14,0,14,0,409,0,0.,0.,0.,0,"{Linear equ
ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodIndControl"},
                {"linear_equality_targets",14,0,12,0,405,0,0.,0.,0.,0,"{Linear eq
uality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodIndControl"},
                {"linear_inequality_constraint_matrix",14,0,6,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
                {"linear_inequality_lower_bounds",14,0,7,0,395,0,0.,0.,0.,0,"{Lin
ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
                {"linear_inequality_scale_types",15,0,9,0,399,0,0.,0.,0.,0,"{Line
ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
                {"linear_inequality_scales",14,0,10,0,401,0,0.,0.,0.,0,"{Linear i
nequality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands
.html#MethodIndControl"},
                {"linear_inequality_upper_bounds",14,0,8,0,397,0,0.,0.,0.,0,"{Lin
ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
                {"log_file",11,0,16,0,347,0,0.,0.,0.,0,"{Log file} http://dakota.
sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"},
                {"mutation_type",8,6,20,0,375,kw_95,0.,0.,0.,0,"{Mutation type} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"
},
                {"niching_type",8,2,3,0,295,kw_98,0.,0.,0.,0,"{Niche pressure typ
e} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEG
AMOGA"},
                {"population_size",9,0,15,0,345,0,0.,0.,0.,0,"{Number of populati
on members} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#
MethodJEGADC"},
                {"postprocessor_type",8,1,5,0,309,kw_99,0.,0.,0.,0,"{Post_process
or type} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Met
hodJEGAMOGA"},
                {"print_each_pop",8,0,17,0,349,0,0.,0.,0.,0,"{Population output}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC
"},
                {"replacement_type",8,4,2,0,283,kw_101,0.,0.,0.,0,"{Replacement t
ype} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJ
EGADC"},
```

```
            {"seed",9,0,21,0,391,0,0.,0.,0.,0,"{Random seed} http://dakota.sa
ndia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"}
            }
```

42.1.4.107 static KeyWord kw_103 () [static]

Initial value:

```
{
            {"partitions",13,0,1,1,1133,0,0.,0.,0.,0,"{Partitions per variabl
e} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodPSM
PS"}
            }
```

42.1.4.108 static KeyWord kw_104 () [static]

Initial value:

```
{
            {"min_boxsize_limit",10,0,2,0,947,0,0.,0.,0.,0,"{Min boxsize limi
t} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNCS
UDC"},
            {"solution_accuracy",2,0,1,0,944},
            {"solution_target",10,0,1,0,945,0,0.,0.,0.,0,"{Solution Target }
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNCSUDC
"},
            {"volume_boxsize_limit",10,0,3,0,949,0,0.,0.,0.,0,"{Volume boxsiz
e limit} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Met
hodNCSUDC"}
            }
```

42.1.4.109 static KeyWord kw_105 () [static]

Initial value:

```
{
            {"absolute_conv_tol",10,0,2,0,551,0,0.,0.,0.,0,"{Absolute functio
n convergence tolerance} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
ommands.html#MethodLSNL2SOL"},
            {"covariance",9,0,8,0,563,0,0.,0.,0.,0,"{Covariance post-processi
ng} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodLS
NL2SOL"},
            {"false_conv_tol",10,0,6,0,559,0,0.,0.,0.,0,"{False convergence t
olerance} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodLSNL2SOL"},
            {"function_precision",10,0,1,0,549,0,0.,0.,0.,0,"{Relative precis
ion in least squares terms} http://dakota.sandia.gov/licensing/votd/html-ref/Meth
odCommands.html#MethodLSNL2SOL"},
            {"initial_trust_radius",10,0,7,0,561,0,0.,0.,0.,0,"{Initial trust
 region radius} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodLSNL2SOL"},
            {"regression_diagnostics",8,0,9,0,565,0,0.,0.,0.,0,"{Regression d
iagnostics post-processing} http://dakota.sandia.gov/licensing/votd/html-ref/Meth
odCommands.html#MethodLSNL2SOL"},
```

```
            {"singular_conv_tol",10,0,4,0,555,0,0.,0.,0.,0,"{Singular converg
ence tolerance} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodLSNL2SOL"},
            {"singular_radius",10,0,5,0,557,0,0.,0.,0.,0,"{Step limit for sct
ol} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodLS
NL2SOL"},
            {"x_conv_tol",10,0,3,0,553,0,0.,0.,0.,0,"{Convergence tolerance f
or change in parameter vector} http://dakota.sandia.gov/licensing/votd/html-ref/M
ethodCommands.html#MethodLSNL2SOL"}
            }
```

42.1.4.110 static KeyWord kw_106 () [static]

Initial value:

```
{
            {"num_reliability_levels",13,0,1,0,749,0,0.,0.,0.,0,"{Number of r
eliability levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommand
s.html#MethodNonD"}
            }
```

42.1.4.111 static KeyWord kw_107 () [static]

Initial value:

```
{
            {"gen_reliabilities",8,0,1,1,761},
            {"probabilities",8,0,1,1,757},
            {"reliabilities",8,0,1,1,759}
            }
```

42.1.4.112 static KeyWord kw_108 () [static]

Initial value:

```
{
            {"compute",8,3,2,0,755,kw_107,0.,0.,0.,0,"{Target statistics for
response levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodNonD"},
            {"num_response_levels",13,0,1,0,753,0,0.,0.,0.,0,"{Number of resp
onse levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodNonD"}
            }
```

42.1.4.113 static KeyWord kw_109 () [static]

Initial value:

```
{
            {"expansion_order",13,0,5,1,641,0,0.,0.,0.,0,"{Expansion order} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDPCE
```

```
"},
          {"expansion_terms",9,0,5,1,643,0,0.,0.,0.,0,"{Expansion terms} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDPCE"
},
          {"ratio_order",10,0,1,0,627,0,0.,0.,0.,0,"{Order of collocation o
versampling relationship} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodNonDPCE"},
          {"reuse_points",8,0,2,0,629},
          {"reuse_samples",0,0,2,0,628},
          {"tensor_grid",8,0,4,0,633},
          {"use_derivatives",8,0,3,0,631}
          }
```

42.1.4.114 static KeyWord kw_110 () [static]

Initial value:

```
{
          {"expansion_order",13,0,5,1,641,0,0.,0.,0.,0,"{Expansion order} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDPCE
"},
          {"expansion_terms",9,0,5,1,643,0,0.,0.,0.,0,"{Expansion terms} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDPCE"
}
          }
```

42.1.4.115 static KeyWord kw_111 () [static]

Initial value:

```
{
          {"expansion_order",13,0,2,1,641,0,0.,0.,0.,0,"{Expansion order} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDPCE
"},
          {"expansion_terms",9,0,2,1,643,0,0.,0.,0.,0,"{Expansion terms} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDPCE"
},
          {"incremental_lhs",8,0,1,0,637,0,0.,0.,0.,0,"{Use incremental LHS
 for expansion_samples} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCo
mmands.html#MethodNonDPCE"}
          }
```

42.1.4.116 static KeyWord kw_112 () [static]

Initial value:

```
{
          {"decay",8,0,1,1,599},
          {"generalized",8,0,1,1,601},
          {"sobol",8,0,1,1,597}
          }
```

42.1.4.117 static KeyWord kw_113 () [static]

Initial value:

```
{
                {"dimension_adaptive",8,3,1,1,595,kw_112},
                {"uniform",8,0,1,1,593}
                }
```

42.1.4.118 static KeyWord kw_114 () [static]

Initial value:

```
{
                {"dimension_preference",14,0,1,0,615,0,0.,0.,0.,0,"{Dimension pre
     ference for anisotropic tensor and sparse grids} http://dakota.sandia.gov/licensi
     ng/votd/html-ref/MethodCommands.html#MethodNonDPCE"},
                {"nested",8,0,2,0,617},
                {"non_nested",8,0,2,0,619}
                }
```

42.1.4.119 static KeyWord kw_115 () [static]

Initial value:

```
{
                {"adapt_import",8,0,1,1,655},
                {"import",8,0,1,1,653},
                {"mm_adapt_import",8,0,1,1,657}
                }
```

42.1.4.120 static KeyWord kw_116 () [static]

Initial value:

```
{
                {"lhs",8,0,1,1,661},
                {"random",8,0,1,1,663}
                }
```

42.1.4.121 static KeyWord kw_117 () [static]

Initial value:

```
{
                {"dimension_preference",14,0,2,0,615,0,0.,0.,0.,0,"{Dimension pre
     ference for anisotropic tensor and sparse grids} http://dakota.sandia.gov/licensi
     ng/votd/html-ref/MethodCommands.html#MethodNonDPCE"},
                {"nested",8,0,3,0,617},
                {"non_nested",8,0,3,0,619},
                {"restricted",8,0,1,0,611},
                {"unrestricted",8,0,1,0,613}
                }
```

42.1.4.122 static KeyWord kw_118 () [static]

Initial value:

```
{
            {"drop_tolerance",10,0,2,0,649,0,0.,0.,0.,0,"{VBD tolerance for o
mitting small indices} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCom
mands.html#MethodNonDMC"},
            {"univariate_effects",8,0,1,0,647,0,0.,0.,0.,0,"{Restriction of V
BD indices to main/total} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodNonDPCE"}
            }
```

42.1.4.123 static KeyWord kw_119 () [static]

Initial value:

```
{
            {"all_variables",8,0,15,0,743,0,0.,0.,0.,0,"{All variables flag}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC
"},
            {"askey",8,0,2,0,603},
            {"collocation_points",9,7,3,1,623,kw_109,0.,0.,0.,0,"{Number coll
ocation points to estimate coeffs} http://dakota.sandia.gov/licensing/votd/html-r
ef/MethodCommands.html#MethodNonDPCE"},
            {"collocation_ratio",10,7,3,1,625,kw_109,0.,0.,0.,0,"{Collocation
 point oversampling ratio to estimate coeffs} http://dakota.sandia.gov/licensing/
votd/html-ref/MethodCommands.html#MethodNonDPCE"},
            {"cubature_integrand",9,0,3,1,621,0,0.,0.,0.,0,"{Cubature integra
nd order for PCE coefficient estimation} http://dakota.sandia.gov/licensing/votd/
html-ref/MethodCommands.html#MethodNonDPCE"},
            {"distribution",8,2,9,0,799,kw_52,0.,0.,0.,0,"{Distribution type}
 http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonD"
},
            {"expansion_import_file",11,2,3,1,639,kw_110,0.,0.,0.,0,"{PCE coe
ffs import file} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodNonDPCE"},
            {"expansion_samples",9,3,3,1,635,kw_111,0.,0.,0.,0,"{Number simul
ation samples to estimate coeffs} http://dakota.sandia.gov/licensing/votd/html-re
f/MethodCommands.html#MethodNonDPCE"},
            {"fixed_seed",8,0,16,0,745,0,0.,0.,0.,0,"{Fixed seed flag} http:/
/dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
            {"gen_reliability_levels",14,1,11,0,809,kw_53,0.,0.,0.,0,"{Genera
lized reliability levels} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodNonD"},
            {"p_refinement",8,2,1,0,591,kw_113,0.,0.,0.,0,"{Automated polynom
ial order refinement} http://dakota.sandia.gov/licensing/votd/html-ref/MethodComm
ands.html#MethodNonDPCE"},
            {"probability_levels",14,1,10,0,805,kw_54,0.,0.,0.,0,"{Probabilit
y levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodNonD"},
            {"quadrature_order",13,3,3,1,607,kw_114,0.,0.,0.,0,"{Quadrature o
rder for PCE coefficient estimation} http://dakota.sandia.gov/licensing/votd/html
-ref/MethodCommands.html#MethodNonDPCE"},
            {"reliability_levels",14,1,13,0,747,kw_106,0.,0.,0.,0,"{Reliabili
ty levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodNonD"},
            {"response_levels",14,2,14,0,751,kw_108,0.,0.,0.,0,"{Response lev
els} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodN
```

```
onD"},
          {"rng",8,2,12,0,813,kw_55,0.,0.,0.,0,"{Random number generator} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"
},
          {"sample_refinement",8,3,5,0,651,kw_115,0.,0.,0.,0,"{Importance s
ampling refinement} http://dakota.sandia.gov/licensing/votd/html-ref/MethodComman
ds.html#MethodNonDPCE"},
          {"sample_type",8,2,6,0,659,kw_116,0.,0.,0.,0,"{Sampling type} htt
p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},

          {"samples",9,0,8,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
          {"seed",9,0,7,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
          {"sparse_grid_level",13,5,3,1,609,kw_117,0.,0.,0.,0,"{Sparse grid
 level for PCE coefficient estimation} http://dakota.sandia.gov/licensing/votd/ht
ml-ref/MethodCommands.html#MethodNonDPCE"},
          {"variance_based_decomp",8,2,4,0,645,kw_118,0.,0.,0.,0,"{Variance
 based decomposition (VBD)} http://dakota.sandia.gov/licensing/votd/html-ref/Meth
odCommands.html#MethodNonDMC"},
          {"wiener",8,0,2,0,605}
          }
```

42.1.4.124 static KeyWord kw_120 () [static]

Initial value:

```
{
          {"previous_samples",9,0,1,1,737,0,0.,0.,0.,0,"{Previous samples f
or incremental approaches} http://dakota.sandia.gov/licensing/votd/html-ref/Metho
dCommands.html#MethodNonDMC"}
          }
```

42.1.4.125 static KeyWord kw_121 () [static]

Initial value:

```
{
          {"incremental_lhs",8,1,1,1,733,kw_120},
          {"incremental_random",8,1,1,1,735,kw_120},
          {"lhs",8,0,1,1,731},
          {"random",8,0,1,1,729}
          }
```

42.1.4.126 static KeyWord kw_122 () [static]

Initial value:

```
{
          {"drop_tolerance",10,0,1,0,741}
          }
```

42.1.4.127 static KeyWord kw_123 () [static]

Initial value:

```
{
                {"all_variables",8,0,11,0,743,0,0.,0.,0.,0,"{All variables flag}
     http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC
     "},
                {"distribution",8,2,5,0,799,kw_52,0.,0.,0.,0,"{Distribution type}
      http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonD"
     },
                {"fixed_seed",8,0,12,0,745,0,0.,0.,0.,0,"{Fixed seed flag} http:/
     /dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"gen_reliability_levels",14,1,7,0,809,kw_53,0.,0.,0.,0,"{General
     ized reliability levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
     ommands.html#MethodNonD"},
                {"probability_levels",14,1,6,0,805,kw_54,0.,0.,0.,0,"{Probability
      levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Met
     hodNonD"},
                {"reliability_levels",14,1,9,0,747,kw_106,0.,0.,0.,0,"{Reliabilit
     y levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
     thodNonD"},
                {"response_levels",14,2,10,0,751,kw_108,0.,0.,0.,0,"{Response lev
     els} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodN
     onD"},
                {"rng",8,2,8,0,813,kw_55,0.,0.,0.,0,"{Random number generator} ht
     tp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"}
     ,
                {"sample_type",8,4,1,0,727,kw_121},
                {"samples",9,0,4,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
     akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"seed",9,0,3,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
     dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
                {"variance_based_decomp",8,1,2,0,739,kw_122}
                }
```

42.1.4.128 static KeyWord kw_124 () [static]

Initial value:

```
{
                {"generalized",8,0,1,1,677},
                {"sobol",8,0,1,1,675}
                }
```

42.1.4.129 static KeyWord kw_125 () [static]

Initial value:

```
{
                {"dimension_adaptive",8,2,1,1,673,kw_124},
                {"uniform",8,0,1,1,671}
                }
```

42.1.4.130 static KeyWord kw_126 () [static]

Initial value:

```
{
                {"hierarchical",8,0,1,0,683},
                {"nodal",8,0,1,0,681}
                }
```

42.1.4.131 static KeyWord kw_127 () [static]

Initial value:

```
{
                {"adapt_import",8,0,1,1,715},
                {"import",8,0,1,1,713},
                {"mm_adapt_import",8,0,1,1,717}
                }
```

42.1.4.132 static KeyWord kw_128 () [static]

Initial value:

```
{
                {"lhs",8,0,1,1,721},
                {"random",8,0,1,1,723}
                }
```

42.1.4.133 static KeyWord kw_129 () [static]

Initial value:

```
{
                {"restricted",8,0,1,0,693},
                {"unrestricted",8,0,1,0,695}
                }
```

42.1.4.134 static KeyWord kw_130 () [static]

Initial value:

```
{
                {"drop_tolerance",10,0,2,0,709,0,0.,0.,0.,0,"{VBD tolerance for o
     mitting small indices} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCom
     mands.html#MethodNonDSC"},
                {"univariate_effects",8,0,1,0,707,0,0.,0.,0.,0,"{Restriction of V
     BD indices to main/total} http://dakota.sandia.gov/licensing/votd/html-ref/Method
     Commands.html#MethodNonDSC"}
                }
```

42.1.4.135 static KeyWord kw_131 () [static]

Initial value:

```
{
            {"all_variables",8,0,18,0,743,0,0.,0.,0.,0,"{All variables flag}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC
"},
            {"askey",8,0,2,0,685},
            {"dimension_preference",14,0,4,0,697,0,0.,0.,0.,0,"{Dimension pre
ference for anisotropic tensor and sparse grids} http://dakota.sandia.gov/licensi
ng/votd/html-ref/MethodCommands.html#MethodNonDSC"},
            {"distribution",8,2,12,0,799,kw_52,0.,0.,0.,0,"{Distribution type
} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonD
"},
            {"fixed_seed",8,0,19,0,745,0,0.,0.,0.,0,"{Fixed seed flag} http:/
/dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
            {"gen_reliability_levels",14,1,14,0,809,kw_53,0.,0.,0.,0,"{Genera
lized reliability levels} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodNonD"},
            {"h_refinement",8,2,1,0,669,kw_125},
            {"nested",8,0,6,0,701},
            {"non_nested",8,0,6,0,703},
            {"p_refinement",8,2,1,0,667,kw_125},
            {"piecewise",8,2,2,0,679,kw_126},
            {"probability_levels",14,1,13,0,805,kw_54,0.,0.,0.,0,"{Probabilit
y levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Me
thodNonD"},
            {"quadrature_order",13,0,3,1,689,0,0.,0.,0.,0,"{Quadrature order
for collocation points} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCo
mmands.html#MethodNonDSC"},
            {"reliability_levels",14,1,16,0,747,kw_106,0.,0.,0.,0,"{Reliabili
ty levels} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodNonD"},
            {"response_levels",14,2,17,0,751,kw_108,0.,0.,0.,0,"{Response lev
els} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodN
onD"},
            {"rng",8,2,15,0,813,kw_55,0.,0.,0.,0,"{Random number generator} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"
},
            {"sample_refinement",8,3,8,0,711,kw_127},
            {"sample_type",8,2,9,0,719,kw_128},
            {"samples",9,0,11,0,939,0,0.,0.,0.,0,"{Number of samples} http://
dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
            {"seed",9,0,10,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.sa
ndia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"},
            {"sparse_grid_level",13,2,3,1,691,kw_129,0.,0.,0.,0,"{Sparse grid
 level for collocation points} http://dakota.sandia.gov/licensing/votd/html-ref/M
ethodCommands.html#MethodNonDSC"},
            {"use_derivatives",8,0,5,0,699,0,0.,0.,0.,0,"{Derivative usage fl
ag} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNo
nDSC"},
            {"variance_based_decomp",8,2,7,0,705,kw_130,0.,0.,0.,0,"{Variance
-based decomposition (VBD)} http://dakota.sandia.gov/licensing/votd/html-ref/Meth
odCommands.html#MethodNonDSC"},
            {"wiener",8,0,2,0,687}
            }
```

42.1.4.136 static KeyWord kw_132 () [static]

Initial value:

```
{
                {"misc_options",15,0,1,0,569}
                }
```

42.1.4.137 static KeyWord kw_133 () [static]

Initial value:

```
{
                {"function_precision",10,0,11,0,197,0,0.,0.,0.,0,"{Function preci
     sion} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
     NPSOLDC"},
                {"linear_equality_constraint_matrix",14,0,6,0,403,0,0.,0.,0.,0,"{
     Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
     ref/MethodCommands.html#MethodIndControl"},
                {"linear_equality_scale_types",15,0,8,0,407,0,0.,0.,0.,0,"{Linear
      equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
     ommands.html#MethodIndControl"},
                {"linear_equality_scales",14,0,9,0,409,0,0.,0.,0.,0,"{Linear equa
     lity scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
     #MethodIndControl"},
                {"linear_equality_targets",14,0,7,0,405,0,0.,0.,0.,0,"{Linear equ
     ality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.ht
     ml#MethodIndControl"},
                {"linear_inequality_constraint_matrix",14,0,1,0,393,0,0.,0.,0.,0,
     "{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
     tml-ref/MethodCommands.html#MethodIndControl"},
                {"linear_inequality_lower_bounds",14,0,2,0,395,0,0.,0.,0.,0,"{Lin
     ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linear_inequality_scale_types",15,0,4,0,399,0,0.,0.,0.,0,"{Line
     ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linear_inequality_scales",14,0,5,0,401,0,0.,0.,0.,0,"{Linear in
     equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
     html#MethodIndControl"},
                {"linear_inequality_upper_bounds",14,0,3,0,397,0,0.,0.,0.,0,"{Lin
     ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linesearch_tolerance",10,0,12,0,199,0,0.,0.,0.,0,"{Line search
     tolerance} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
     ethodNPSOLDC"},
                {"verify_level",9,0,10,0,195,0,0.,0.,0.,0,"{Gradient verification
      level} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Meth
     odNPSOLDC"}
                }
```

42.1.4.138 static KeyWord kw_134 () [static]

Initial value:

```
{
```

```
            {"gradient_tolerance",10,0,11,0,233},
            {"linear_equality_constraint_matrix",14,0,6,0,403,0,0.,0.,0.,0,"{
Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
ref/MethodCommands.html#MethodIndControl"},
            {"linear_equality_scale_types",15,0,8,0,407,0,0.,0.,0.,0,"{Linear
 equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
ommands.html#MethodIndControl"},
            {"linear_equality_scales",14,0,9,0,409,0,0.,0.,0.,0,"{Linear equa
lity scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html
#MethodIndControl"},
            {"linear_equality_targets",14,0,7,0,405,0,0.,0.,0.,0,"{Linear equ
ality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.ht
ml#MethodIndControl"},
            {"linear_inequality_constraint_matrix",14,0,1,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
            {"linear_inequality_lower_bounds",14,0,2,0,395,0,0.,0.,0.,0,"{Lin
ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"linear_inequality_scale_types",15,0,4,0,399,0,0.,0.,0.,0,"{Line
ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"linear_inequality_scales",14,0,5,0,401,0,0.,0.,0.,0,"{Linear in
equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodIndControl"},
            {"linear_inequality_upper_bounds",14,0,3,0,397,0,0.,0.,0.,0,"{Lin
ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"max_step",10,0,10,0,231}
            }
```

42.1.4.139 static KeyWord kw_135 () [static]

Initial value:

```
{
            {"linear_equality_constraint_matrix",14,0,7,0,403,0,0.,0.,0.,0,"{
Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
ref/MethodCommands.html#MethodIndControl"},
            {"linear_equality_scale_types",15,0,9,0,407,0,0.,0.,0.,0,"{Linear
 equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
ommands.html#MethodIndControl"},
            {"linear_equality_scales",14,0,10,0,409,0,0.,0.,0.,0,"{Linear equ
ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodIndControl"},
            {"linear_equality_targets",14,0,8,0,405,0,0.,0.,0.,0,"{Linear equ
ality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.ht
ml#MethodIndControl"},
            {"linear_inequality_constraint_matrix",14,0,2,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
            {"linear_inequality_lower_bounds",14,0,3,0,395,0,0.,0.,0.,0,"{Lin
ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"linear_inequality_scale_types",15,0,5,0,399,0,0.,0.,0.,0,"{Line
ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"linear_inequality_scales",14,0,6,0,401,0,0.,0.,0.,0,"{Linear in
equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodIndControl"},
```

```
                {"linear_inequality_upper_bounds",14,0,4,0,397,0,0.,0.,0.,0,"{Lin
     ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"search_scheme_size",9,0,1,0,237}
                }
```

42.1.4.140 static KeyWord kw_136 () [static]

Initial value:

```
{
                {"gradient_based_line_search",8,0,1,1,217,0,0.,0.,0.,0,"[CHOOSE l
     ine search type]"},
                {"tr_pds",8,0,1,1,221},
                {"trust_region",8,0,1,1,219},
                {"value_based_line_search",8,0,1,1,215}
                }
```

42.1.4.141 static KeyWord kw_137 () [static]

Initial value:

```
{
                {"centering_parameter",10,0,5,0,229},
                {"central_path",11,0,3,0,225},
                {"gradient_tolerance",10,0,16,0,233},
                {"linear_equality_constraint_matrix",14,0,11,0,403,0,0.,0.,0.,0,"
     {Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html
     -ref/MethodCommands.html#MethodIndControl"},
                {"linear_equality_scale_types",15,0,13,0,407,0,0.,0.,0.,0,"{Linea
     r equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Method
     Commands.html#MethodIndControl"},
                {"linear_equality_scales",14,0,14,0,409,0,0.,0.,0.,0,"{Linear equ
     ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
     l#MethodIndControl"},
                {"linear_equality_targets",14,0,12,0,405,0,0.,0.,0.,0,"{Linear eq
     uality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
     tml#MethodIndControl"},
                {"linear_inequality_constraint_matrix",14,0,6,0,393,0,0.,0.,0.,0,
     "{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
     tml-ref/MethodCommands.html#MethodIndControl"},
                {"linear_inequality_lower_bounds",14,0,7,0,395,0,0.,0.,0.,0,"{Lin
     ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linear_inequality_scale_types",15,0,9,0,399,0,0.,0.,0.,0,"{Line
     ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"linear_inequality_scales",14,0,10,0,401,0,0.,0.,0.,0,"{Linear i
     nequality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands
     .html#MethodIndControl"},
                {"linear_inequality_upper_bounds",14,0,8,0,397,0,0.,0.,0.,0,"{Lin
     ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
     hodCommands.html#MethodIndControl"},
                {"max_step",10,0,15,0,231},
                {"merit_function",11,0,2,0,223},
                {"search_method",8,4,1,0,213,kw_136},
                {"steplength_to_boundary",10,0,4,0,227}
                }
```

42.1.4.142 static KeyWord kw_138 () [static]

Initial value:

```
{
                {"debug",8,0,1,1,67,0,0.,0.,0.,0,"[CHOOSE output level]"},
                {"normal",8,0,1,1,71},
                {"quiet",8,0,1,1,73},
                {"silent",8,0,1,1,75},
                {"verbose",8,0,1,1,69}
                }
```

42.1.4.143 static KeyWord kw_139 () [static]

Initial value:

```
{
                {"partitions",13,0,1,0,937,0,0.,0.,0.,0,"{Number of partitions} h
     ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodPSUADE"
     },
                {"samples",9,0,3,0,939,0,0.,0.,0.,0,"{Number of samples} http://d
     akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodNonDMC"},
                {"seed",9,0,2,0,941,0,0.,0.,0.,0,"{Random seed} http://dakota.san
     dia.gov/licensing/votd/html-ref/MethodCommands.html#MethodEG"}
                }
```

42.1.4.144 static KeyWord kw_140 () [static]

Initial value:

```
{
                {"converge_order",8,0,1,1,1139},
                {"converge_qoi",8,0,1,1,1141},
                {"estimate_order",8,0,1,1,1137},
                {"refinement_rate",10,0,2,0,1143,0,0.,0.,0.,0,"{Refinement rate}
     http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSolnRi
     chardson"}
                }
```

42.1.4.145 static KeyWord kw_141 () [static]

Initial value:

```
{
                {"num_generations",0x29,0,2,0,343},
                {"percent_change",10,0,1,0,341}
                }
```

42.1.4.146 static KeyWord kw_142 () [static]

Initial value:

```
{
                {"num_generations",0x29,0,2,0,337,0,0.,0.,0.,0,"{Number of genera
     tions (for convergence test) } http://dakota.sandia.gov/licensing/votd/html-ref/M
     ethodCommands.html#MethodJEGASOGA"},
                {"percent_change",10,0,1,0,335,0,0.,0.,0.,0,"{Percent change in f
     itness} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Meth
     odJEGASOGA"}
                }
```

42.1.4.147 static KeyWord kw_143 () [static]

Initial value:

```
{
                {"average_fitness_tracker",8,2,1,1,339,kw_141},
                {"best_fitness_tracker",8,2,1,1,333,kw_142}
                }
```

42.1.4.148 static KeyWord kw_144 () [static]

Initial value:

```
{
                {"constraint_penalty",10,0,2,0,319,0,0.,0.,0.,0,"{Constraint pena
     lty in merit function} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCom
     mands.html#MethodJEGASOGA"},
                {"merit_function",8,0,1,1,317}
                }
```

42.1.4.149 static KeyWord kw_145 () [static]

Initial value:

```
{
                {"elitist",8,0,1,1,323},
                {"favor_feasible",8,0,1,1,325},
                {"roulette_wheel",8,0,1,1,327},
                {"unique_roulette_wheel",8,0,1,1,329}
                }
```

42.1.4.150 static KeyWord kw_146 () [static]

Initial value:

```
{
                {"convergence_type",8,2,3,0,331,kw_143,0.,0.,0.,0,"{Convergence t
     ype} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJ
     EGASOGA"},
                {"crossover_type",8,5,17,0,359,kw_92,0.,0.,0.,0,"{Crossover type}
      http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGAD
     C"},
```

```
            {"fitness_type",8,2,1,0,315,kw_144,0.,0.,0.,0,"{Fitness type} htt
p://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGASOGA"
},
            {"initialization_type",8,3,16,0,351,kw_93,0.,0.,0.,0,"{Initializa
tion type} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodJEGADC"},
            {"linear_equality_constraint_matrix",14,0,9,0,403,0,0.,0.,0.,0,"{
Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
ref/MethodCommands.html#MethodIndControl"},
            {"linear_equality_scale_types",15,0,11,0,407,0,0.,0.,0.,0,"{Linea
r equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodIndControl"},
            {"linear_equality_scales",14,0,12,0,409,0,0.,0.,0.,0,"{Linear equ
ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodIndControl"},
            {"linear_equality_targets",14,0,10,0,405,0,0.,0.,0.,0,"{Linear eq
uality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodIndControl"},
            {"linear_inequality_constraint_matrix",14,0,4,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
            {"linear_inequality_lower_bounds",14,0,5,0,395,0,0.,0.,0.,0,"{Lin
ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"linear_inequality_scale_types",15,0,7,0,399,0,0.,0.,0.,0,"{Line
ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"linear_inequality_scales",14,0,8,0,401,0,0.,0.,0.,0,"{Linear in
equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodIndControl"},
            {"linear_inequality_upper_bounds",14,0,6,0,397,0,0.,0.,0.,0,"{Lin
ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
            {"log_file",11,0,14,0,347,0,0.,0.,0.,0,"{Log file} http://dakota.
sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"},
            {"mutation_type",8,6,18,0,375,kw_95,0.,0.,0.,0,"{Mutation type} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"
},
            {"population_size",9,0,13,0,345,0,0.,0.,0.,0,"{Number of populati
on members} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#
MethodJEGADC"},
            {"print_each_pop",8,0,15,0,349,0,0.,0.,0.,0,"{Population output}
http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC
"},
            {"replacement_type",8,4,2,0,321,kw_145,0.,0.,0.,0,"{Replacement t
ype} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJ
EGASOGA"},
            {"seed",9,0,19,0,391,0,0.,0.,0.,0,"{Random seed} http://dakota.sa
ndia.gov/licensing/votd/html-ref/MethodCommands.html#MethodJEGADC"}
            }
```

42.1.4.151 static KeyWord kw_147 () [static]

Initial value:

```
{
            {"function_precision",10,0,12,0,197,0,0.,0.,0.,0,"{Function preci
sion} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
NPSOLDC"},
            {"linear_equality_constraint_matrix",14,0,7,0,403,0,0.,0.,0.,0,"{
```

```
Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html-
ref/MethodCommands.html#MethodIndControl"},
          {"linear_equality_scale_types",15,0,9,0,407,0,0.,0.,0.,0,"{Linear
 equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/MethodC
ommands.html#MethodIndControl"},
          {"linear_equality_scales",14,0,10,0,409,0,0.,0.,0.,0,"{Linear equ
ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodIndControl"},
          {"linear_equality_targets",14,0,8,0,405,0,0.,0.,0.,0,"{Linear equ
ality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.ht
ml#MethodIndControl"},
          {"linear_inequality_constraint_matrix",14,0,2,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
          {"linear_inequality_lower_bounds",14,0,3,0,395,0,0.,0.,0.,0,"{Lin
ear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
          {"linear_inequality_scale_types",15,0,5,0,399,0,0.,0.,0.,0,"{Line
ar inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
          {"linear_inequality_scales",14,0,6,0,401,0,0.,0.,0.,0,"{Linear in
equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodIndControl"},
          {"linear_inequality_upper_bounds",14,0,4,0,397,0,0.,0.,0.,0,"{Lin
ear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Met
hodCommands.html#MethodIndControl"},
          {"linesearch_tolerance",10,0,13,0,199,0,0.,0.,0.,0,"{Line search
tolerance} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#M
ethodNPSOLDC"},
          {"nlssol",8,0,1,1,193},
          {"npsol",8,0,1,1,191},
          {"verify_level",9,0,11,0,195,0,0.,0.,0.,0,"{Gradient verification
 level} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Meth
odNPSOLDC"}
          }
```

42.1.4.152 static KeyWord kw_148 () [static]

Initial value:

```
{
          {"approx_method_name",11,0,1,1,573,0,0.,0.,0.,0,"[CHOOSE sub-meth
od ref.]{Approximate sub-problem minimization method name} http://dakota.sandia.g
ov/licensing/votd/html-ref/MethodCommands.html#MethodSBG"},
          {"approx_method_pointer",11,0,1,1,575,0,0.,0.,0.,0,"{Approximate
sub-problem minimization method pointer} http://dakota.sandia.gov/licensing/votd/
html-ref/MethodCommands.html#MethodSBG"},
          {"replace_points",8,0,2,0,577,0,0.,0.,0.,0,"{Replace points used
in surrogate construction with best points from previous iteration} http://dakota
.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodSBG"}
          }
```

42.1.4.153 static KeyWord kw_149 () [static]

Initial value:

```
{
```

```
                {"filter",8,0,1,1,145,0,0.,0.,0.,0,"@[CHOOSE acceptance logic]"},

                {"tr_ratio",8,0,1,1,143}
                }
```

42.1.4.154 static KeyWord kw_150 () [static]

Initial value:

```
{
                {"augmented_lagrangian_objective",8,0,1,1,121,0,0.,0.,0.,0,"[CHOO
     SE objective formulation]"},
                {"lagrangian_objective",8,0,1,1,123},
                {"linearized_constraints",8,0,2,2,127,0,0.,0.,0.,0,"[CHOOSE const
     raint formulation]"},
                {"no_constraints",8,0,2,2,129},
                {"original_constraints",8,0,2,2,125,0,0.,0.,0.,0,"@"},
                {"original_primary",8,0,1,1,117,0,0.,0.,0.,0,"@"},
                {"single_objective",8,0,1,1,119}
                }
```

42.1.4.155 static KeyWord kw_151 () [static]

Initial value:

```
{
                {"homotopy",8,0,1,1,149}
                }
```

42.1.4.156 static KeyWord kw_152 () [static]

Initial value:

```
{
                {"adaptive_penalty_merit",8,0,1,1,135,0,0.,0.,0.,0,"[CHOOSE merit
      function]"},
                {"augmented_lagrangian_merit",8,0,1,1,139,0,0.,0.,0.,0,"@"},
                {"lagrangian_merit",8,0,1,1,137},
                {"penalty_merit",8,0,1,1,133}
                }
```

42.1.4.157 static KeyWord kw_153 () [static]

Initial value:

```
{
                {"contract_threshold",10,0,3,0,107,0,0.,0.,0.,0,"{Shrink trust re
     gion if trust region ratio is below this value} http://dakota.sandia.gov/licensin
     g/votd/html-ref/MethodCommands.html#MethodSBL"},
                {"contraction_factor",10,0,5,0,111,0,0.,0.,0.,0,"{Trust region co
     ntraction factor} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands
```

```
.html#MethodSBL"},
          {"expand_threshold",10,0,4,0,109,0,0.,0.,0.,0,"{Expand trust regi
on if trust region ratio is above this value} http://dakota.sandia.gov/licensing/
votd/html-ref/MethodCommands.html#MethodSBL"},
          {"expansion_factor",10,0,6,0,113,0,0.,0.,0.,0,"{Trust region expa
nsion factor} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodSBL"},
          {"initial_size",10,0,1,0,103,0,0.,0.,0.,0,"{Trust region initial
size (relative to bounds)} http://dakota.sandia.gov/licensing/votd/html-ref/Metho
dCommands.html#MethodSBL"},
          {"minimum_size",10,0,2,0,105,0,0.,0.,0.,0,"{Trust region minimum
size} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
SBL"}
          }
```

42.1.4.158 static KeyWord kw_154 () [static]

Initial value:

```
{
          {"acceptance_logic",8,2,7,0,141,kw_149,0.,0.,0.,0,"{SBL iterate a
cceptance logic} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodSBL"},
          {"approx_method_name",11,0,1,1,93,0,0.,0.,0.,0,"[CHOOSE sub-metho
d ref.]{Approximate sub-problem minimization method name} http://dakota.sandia.go
v/licensing/votd/html-ref/MethodCommands.html#MethodSBL"},
          {"approx_method_pointer",11,0,1,1,95,0,0.,0.,0.,0,"{Approximate s
ub-problem minimization method pointer} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodSBL"},
          {"approx_subproblem",8,7,5,0,115,kw_150,0.,0.,0.,0,"{Approximate
subproblem formulation} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCo
mmands.html#MethodSBL"},
          {"constraint_relax",8,1,8,0,147,kw_151,0.,0.,0.,0,"{SBL constrain
t relaxation method for infeasible iterates} http://dakota.sandia.gov/licensing/v
otd/html-ref/MethodCommands.html#MethodSBL"},
          {"linear_equality_constraint_matrix",14,0,14,0,403,0,0.,0.,0.,0,"
{Linear equality coefficient matrix} http://dakota.sandia.gov/licensing/votd/html
-ref/MethodCommands.html#MethodIndControl"},
          {"linear_equality_scale_types",15,0,16,0,407,0,0.,0.,0.,0,"{Linea
r equality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Method
Commands.html#MethodIndControl"},
          {"linear_equality_scales",14,0,17,0,409,0,0.,0.,0.,0,"{Linear equ
ality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.htm
l#MethodIndControl"},
          {"linear_equality_targets",14,0,15,0,405,0,0.,0.,0.,0,"{Linear eq
uality targets} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.h
tml#MethodIndControl"},
          {"linear_inequality_constraint_matrix",14,0,9,0,393,0,0.,0.,0.,0,
"{Linear inequality coefficient matrix} http://dakota.sandia.gov/licensing/votd/h
tml-ref/MethodCommands.html#MethodIndControl"},
          {"linear_inequality_lower_bounds",14,0,10,0,395,0,0.,0.,0.,0,"{Li
near inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"linear_inequality_scale_types",15,0,12,0,399,0,0.,0.,0.,0,"{Lin
ear inequality scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"linear_inequality_scales",14,0,13,0,401,0,0.,0.,0.,0,"{Linear i
nequality scales} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands
.html#MethodIndControl"},
          {"linear_inequality_upper_bounds",14,0,11,0,397,0,0.,0.,0.,0,"{Li
```

```
near inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodIndControl"},
          {"merit_function",8,4,6,0,131,kw_152,0.,0.,0.,0,"{SBL merit funct
ion} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodS
BL"},
          {"soft_convergence_limit",9,0,2,0,97,0,0.,0.,0.,0,"{Soft converge
nce limit for SBL iterations} http://dakota.sandia.gov/licensing/votd/html-ref/Me
thodCommands.html#MethodSBL"},
          {"trust_region",8,6,4,0,101,kw_153,0.,0.,0.,0,"{Trust region grou
p specification} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.
html#MethodSBL"},
          {"truth_surrogate_bypass",8,0,3,0,99,0,0.,0.,0.,0,"{Flag for bypa
ssing lower level surrogates in truth verifications} http://dakota.sandia.gov/lic
ensing/votd/html-ref/MethodCommands.html#MethodSBL"}
          }
```

42.1.4.159 static KeyWord kw_155 () [static]

Initial value:

```
{
          {"final_point",14,0,1,1,1115,0,0.,0.,0.,0,"[CHOOSE final pt or in
crement]{Termination point of vector} http://dakota.sandia.gov/licensing/votd/htm
l-ref/MethodCommands.html#MethodPSVPS"},
          {"num_steps",9,0,2,2,1119,0,0.,0.,0.,0,"{Number of steps along ve
ctor} http://dakota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#Method
PSVPS"},
          {"step_vector",14,0,1,1,1117,0,0.,0.,0.,0,"{Step vector} http://d
akota.sandia.gov/licensing/votd/html-ref/MethodCommands.html#MethodPSVPS"}
          }
```

42.1.4.160 static KeyWord kw_157 () [static]

Initial value:

```
{
          {"optional_interface_responses_pointer",11,0,1,0,1311,0,0.,0.,0.,
0,"{Responses pointer for nested model optional interfaces} http://dakota.sandia.
gov/licensing/votd/html-ref/ModelCommands.html#ModelNested"}
          }
```

42.1.4.161 static KeyWord kw_158 () [static]

Initial value:

```
{
          {"primary_response_mapping",14,0,3,0,1319,0,0.,0.,0.,0,"{Primary
response mappings for nested models} http://dakota.sandia.gov/licensing/votd/html
-ref/ModelCommands.html#ModelNested"},
          {"primary_variable_mapping",15,0,1,0,1315,0,0.,0.,0.,0,"{Primary
variable mappings for nested models} http://dakota.sandia.gov/licensing/votd/html
-ref/ModelCommands.html#ModelNested"},
          {"secondary_response_mapping",14,0,4,0,1321,0,0.,0.,0.,0,"{Second
ary response mappings for nested models} http://dakota.sandia.gov/licensing/votd/
```

```
html-ref/ModelCommands.html#ModelNested"},
          {"secondary_variable_mapping",15,0,2,0,1317,0,0.,0.,0.,0,"{Second
ary variable mappings for nested models} http://dakota.sandia.gov/licensing/votd/
html-ref/ModelCommands.html#ModelNested"}
          }
```

42.1.4.162 static KeyWord kw_159 () [static]

Initial value:

```
{
          {"optional_interface_pointer",11,1,1,0,1309,kw_157,0.,0.,0.,0,"{O
ptional interface set pointer} http://dakota.sandia.gov/licensing/votd/html-ref/M
odelCommands.html#ModelNested"},
          {"sub_method_pointer",11,4,2,1,1313,kw_158,0.,0.,0.,0,"{Sub-metho
d pointer for nested models} http://dakota.sandia.gov/licensing/votd/html-ref/Mod
elCommands.html#ModelNested"}
          }
```

42.1.4.163 static KeyWord kw_160 () [static]

Initial value:

```
{
          {"interface_pointer",11,0,1,0,1155,0,0.,0.,0.,0,"{Interface set p
ointer} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#Model
Single"}
          }
```

42.1.4.164 static KeyWord kw_161 () [static]

Initial value:

```
{
          {"additive",8,0,2,2,1269,0,0.,0.,0.,0,"[CHOOSE correction type]"}
,
          {"combined",8,0,2,2,1273},
          {"first_order",8,0,1,1,1265,0,0.,0.,0.,0,"[CHOOSE correction orde
r]"},
          {"multiplicative",8,0,2,2,1271},
          {"second_order",8,0,1,1,1267},
          {"zeroth_order",8,0,1,1,1263}
          }
```

42.1.4.165 static KeyWord kw_162 () [static]

Initial value:

```
{
          {"constant",8,0,1,1,1171},
          {"linear",8,0,1,1,1173},
          {"reduced_quadratic",8,0,1,1,1175}
          }
```

42.1.4.166 static KeyWord kw_163 () [static]

Initial value:

```
{
                {"point_selection",8,0,1,0,1167,0,0.,0.,0.,0,"{GP point selection
     } http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"
     },
                {"trend",8,3,2,0,1169,kw_162,0.,0.,0.,0,"{GP trend function} http
     ://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"}
                }
```

42.1.4.167 static KeyWord kw_164 () [static]

Initial value:

```
{
                {"constant",8,0,1,1,1181},
                {"linear",8,0,1,1,1183},
                {"quadratic",8,0,1,1,1187},
                {"reduced_quadratic",8,0,1,1,1185}
                }
```

42.1.4.168 static KeyWord kw_165 () [static]

Initial value:

```
{
                {"correlation_lengths",14,0,4,0,1193,0,0.,0.,0.,0,"{Surfpack GP c
     orrelation lengths} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommand
     s.html#ModelSurrG"},
                {"max_trials",0x19,0,3,0,1191,0,0.,0.,0.,0,"{Surfpack GP maximum
     trials} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#Model
     SurrG"},
                {"optimization_method",11,0,2,0,1189,0,0.,0.,0.,0,"{Surfpack GP o
     ptimization method} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommand
     s.html#ModelSurrG"},
                {"trend",8,4,1,0,1179,kw_164,0.,0.,0.,0,"{Surfpack GP trend funct
     ion} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSur
     rG"}
                }
```

42.1.4.169 static KeyWord kw_166 () [static]

Initial value:

```
{
                {"dakota",8,2,1,1,1165,kw_163},
                {"surfpack",8,4,1,1,1177,kw_165}
                }
```

42.1.4.170 static KeyWord kw_167 () [static]

Initial value:

```
{
                {"cubic",8,0,1,1,1203},
                {"linear",8,0,1,1,1201}
                }
```

42.1.4.171 static KeyWord kw_168 () [static]

Initial value:

```
{
                {"interpolation",8,2,2,0,1199,kw_167,0.,0.,0.,0,"{MARS interpolat
     ion} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSur
     rG"},
                {"max_bases",9,0,1,0,1197,0,0.,0.,0.,0,"{MARS maximum bases} http
     ://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"}
                }
```

42.1.4.172 static KeyWord kw_169 () [static]

Initial value:

```
{
                {"poly_order",9,0,1,0,1207,0,0.,0.,0.,0,"{MLS polynomial order} h
     ttp://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"},
                {"weight_function",9,0,2,0,1209,0,0.,0.,0.,0,"{MLS weight functio
     n} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG
     "}
                }
```

42.1.4.173 static KeyWord kw_170 () [static]

Initial value:

```
{
                {"nodes",9,0,1,0,1213,0,0.,0.,0.,0,"{ANN number nodes} http://dak
     ota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"},
                {"random_weight",9,0,3,0,1217,0,0.,0.,0.,0,"{ANN random weight} h
     ttp://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"},
                {"range",10,0,2,0,1215,0,0.,0.,0.,0,"{ANN range} http://dakota.sa
     ndia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"}
                }
```

42.1.4.174 static KeyWord kw_171 () [static]

Initial value:

```
{
                {"annotated",8,0,1,0,1255,0,0.,0.,0.,0,"{Data file in annotated f
     ormat} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelS
     urrG"},
                {"freeform",8,0,1,0,1257,0,0.,0.,0.,0,"{Data file in freeform for
     mat} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSur
     rG"}
                }
```

42.1.4.175 static KeyWord kw_172 () [static]

Initial value:

```
{
                {"cubic",8,0,1,1,1235,0,0.,0.,0.,0,"[CHOOSE polynomial order]"},
                {"linear",8,0,1,1,1231},
                {"quadratic",8,0,1,1,1233}
                }
```

42.1.4.176 static KeyWord kw_173 () [static]

Initial value:

```
{
                {"bases",9,0,1,0,1221,0,0.,0.,0.,0,"{RBF number of bases} http://
     dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"},
                {"max_pts",9,0,2,0,1223,0,0.,0.,0.,0,"{RBF maximum points} http:/
     /dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"},
                {"max_subsets",9,0,4,0,1227},
                {"min_partition",9,0,3,0,1225,0,0.,0.,0.,0,"{RBF minimum partitio
     ns} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurr
     G"}
                }
```

42.1.4.177 static KeyWord kw_174 () [static]

Initial value:

```
{
                {"all",8,0,1,1,1247},
                {"none",8,0,1,1,1251},
                {"region",8,0,1,1,1249}
                }
```

42.1.4.178 static KeyWord kw_175 () [static]

Initial value:

```
{
                {"correction",8,6,7,0,1261,kw_161,0.,0.,0.,0,"{Surrogate correcti
     on approach} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#
```

```
ModelSurrG"},
          {"dace_method_pointer",11,0,3,0,1243,0,0.,0.,0.,0,"{Design of exp
eriments method pointer} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCo
mmands.html#ModelSurrG"},
          {"diagnostics",15,0,8,0,1275,0,0.,0.,0.,0,"{Print diagnostic metr
ics about the surrogate goodness of fit} http://dakota.sandia.gov/licensing/votd/
html-ref/ModelCommands.html#ModelSurrG"},
          {"gaussian_process",8,2,1,1,1163,kw_166,0.,0.,0.,0,"[CHOOSE surro
gate type]{DAKOTA Gaussian process} http://dakota.sandia.gov/licensing/votd/html-
ref/ModelCommands.html#ModelSurrG"},
          {"kriging",0,2,1,1,1162,kw_166},
          {"mars",8,2,1,1,1195,kw_168,0.,0.,0.,0,"{Multivariate adaptive re
gression splines} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.
html#ModelSurrG"},
          {"minimum_points",8,0,2,0,1239},
          {"moving_least_squares",8,2,1,1,1205,kw_169,0.,0.,0.,0,"{Moving l
east squares} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html
#ModelSurrG"},
          {"neural_network",8,3,1,1,1211,kw_170,0.,0.,0.,0,"{Artificial neu
ral network} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#
ModelSurrG"},
          {"points_file",11,2,5,0,1253,kw_171,0.,0.,0.,0,"{File import of s
amples for global approximation builds} http://dakota.sandia.gov/licensing/votd/h
tml-ref/ModelCommands.html#ModelSurrG"},
          {"polynomial",8,3,1,1,1229,kw_172,0.,0.,0.,0,"{Polynomial} http:/
/dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrG"},
          {"radial_basis",8,4,1,1,1219,kw_173},
          {"recommended_points",8,0,2,0,1241},
          {"reuse_points",8,3,4,0,1245,kw_174},
          {"reuse_samples",0,3,4,0,1244,kw_174},
          {"samples_file",3,2,5,0,1252,kw_171},
          {"total_points",9,0,2,0,1237},
          {"use_derivatives",8,0,6,0,1259,0,0.,0.,0.,0,"{Surfpack GP gradie
nt enhancement} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.ht
ml#ModelSurrG"}
          }
```

42.1.4.179 static KeyWord kw_176 () [static]

Initial value:

```
{
          {"additive",8,0,2,2,1301,0,0.,0.,0.,0,"[CHOOSE correction type]"}
,
          {"combined",8,0,2,2,1305},
          {"first_order",8,0,1,1,1297,0,0.,0.,0.,0,"[CHOOSE correction orde
r]"},
          {"multiplicative",8,0,2,2,1303},
          {"second_order",8,0,1,1,1299},
          {"zeroth_order",8,0,1,1,1295}
          }
```

42.1.4.180 static KeyWord kw_177 () [static]

Initial value:

```
{
```

```
            {"correction",8,6,3,3,1293,kw_176,0.,0.,0.,0,"{Surrogate correcti
on approach} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#
ModelSurrH"},
            {"high_fidelity_model_pointer",11,0,2,2,1291,0,0.,0.,0.,0,"{Point
er to the high fidelity model specification} http://dakota.sandia.gov/licensing/v
otd/html-ref/ModelCommands.html#ModelSurrH"},
            {"low_fidelity_model_pointer",11,0,1,1,1289,0,0.,0.,0.,0,"{Pointe
r to the low fidelity model specification} http://dakota.sandia.gov/licensing/vot
d/html-ref/ModelCommands.html#ModelSurrH"}
            }
```

42.1.4.181 static KeyWord kw_178 () [static]

Initial value:

```
{
            {"actual_model_pointer",11,0,2,2,1285,0,0.,0.,0.,0,"{Pointer to t
he truth model specification} http://dakota.sandia.gov/licensing/votd/html-ref/Mo
delCommands.html#ModelSurrMP"},
            {"taylor_series",8,0,1,1,1283,0,0.,0.,0.,0,"{Taylor series local
approximation } http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.ht
ml#ModelSurrL"}
            }
```

42.1.4.182 static KeyWord kw_179 () [static]

Initial value:

```
{
            {"actual_model_pointer",11,0,2,2,1285,0,0.,0.,0.,0,"{Pointer to t
he truth model specification} http://dakota.sandia.gov/licensing/votd/html-ref/Mo
delCommands.html#ModelSurrMP"},
            {"tana",8,0,1,1,1279,0,0.,0.,0.,0,"{Two-point adaptive nonlinear
approximation } http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.ht
ml#ModelSurrMP"}
            }
```

42.1.4.183 static KeyWord kw_180 () [static]

Initial value:

```
{
            {"global",8,18,2,1,1161,kw_175,0.,0.,0.,0,"[CHOOSE surrogate cate
gory]{Global approximations } http://dakota.sandia.gov/licensing/votd/html-ref/Mo
delCommands.html#ModelSurrG"},
            {"hierarchical",8,3,2,1,1287,kw_177,0.,0.,0.,0,"{Hierarchical app
roximation } http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#
ModelSurrH"},
            {"id_surrogates",13,0,1,0,1159,0,0.,0.,0.,0,"{Surrogate response
ids} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSur
rogate"},
            {"local",8,2,2,1,1281,kw_178,0.,0.,0.,0,"{Local approximation} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelSurrL"},
            {"multipoint",8,2,2,1,1277,kw_179,0.,0.,0.,0,"{Multipoint approxi
```

```
mation} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#Model
SurrMP"}
          }
```

42.1.4.184 static KeyWord kw_181 () [static]

Initial value:

```
{
          {"id_model",11,0,1,0,1147,0,0.,0.,0.,0,"{Model set identifier} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#ModelIndControl
"},
          {"nested",8,2,4,1,1307,kw_159,0.,0.,0.,0,"[CHOOSE model type]"},
          {"responses_pointer",11,0,3,0,1151,0,0.,0.,0.,0,"{Responses set p
ointer} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#Model
IndControl"},
          {"single",8,1,4,1,1153,kw_160,0.,0.,0.,0,"@"},
          {"surrogate",8,5,4,1,1157,kw_180},
          {"variables_pointer",11,0,2,0,1149,0,0.,0.,0.,0,"{Variables set p
ointer} http://dakota.sandia.gov/licensing/votd/html-ref/ModelCommands.html#Model
IndControl"}
          }
```

42.1.4.185 static KeyWord kw_182 () [static]

Initial value:

```
{
          {"annotated",8,0,2,0,1727,0,0.,0.,0.,0,"{Data file in annotated f
ormat} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespFnL
S"},
          {"freeform",8,0,2,0,1729,0,0.,0.,0.,0,"{Data file in freeform for
mat} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespFnLS"
},
          {"num_config_variables",0x29,0,3,0,1731,0,0.,0.,0.,0,"{Configurat
ion variable columns in file} http://dakota.sandia.gov/licensing/votd/html-ref/Re
spCommands.html#RespFnLS"},
          {"num_experiments",0x29,0,1,0,1725,0,0.,0.,0.,0,"{Experiments (ro
ws) in file} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#R
espFnLS"},
          {"num_std_deviations",0x29,0,4,0,1733,0,0.,0.,0.,0,"{Standard dev
iation columns in file} http://dakota.sandia.gov/licensing/votd/html-ref/RespComm
ands.html#RespFnLS"}
          }
```

42.1.4.186 static KeyWord kw_183 () [static]

Initial value:

```
{
          {"nonlinear_equality_scale_types",0x80f,0,2,0,1749,0,0.,0.,0.,0,"
{Nonlinear equality scaling types} http://dakota.sandia.gov/licensing/votd/html-r
ef/RespCommands.html#RespFnLS",0,"nonlinear_equality_constraints"},
          {"nonlinear_equality_scales",0x80e,0,3,0,1751,0,0.,0.,0.,0,"{Nonl
```

```
inear equality scales} http://dakota.sandia.gov/licensing/votd/html-ref/RespComma
nds.html#RespFnLS",0,"nonlinear_equality_constraints"},
          {"nonlinear_equality_targets",14,0,1,0,1747,0,0.,0.,0.,0,"{Nonlin
ear equality targets} http://dakota.sandia.gov/licensing/votd/html-ref/RespComman
ds.html#RespFnLS",0,"nonlinear_equality_constraints"}
          }
```

42.1.4.187 static KeyWord kw_184 () [static]

Initial value:

```
{
          {"nonlinear_inequality_lower_bounds",14,0,1,0,1737,0,0.,0.,0.,0,"
{Nonlinear inequality lower bounds} http://dakota.sandia.gov/licensing/votd/html-
ref/RespCommands.html#RespFnLS",0,"nonlinear_inequality_constraints"},
          {"nonlinear_inequality_scale_types",0x80f,0,3,0,1741,0,0.,0.,0.,0
,"{Nonlinear inequality scaling types} http://dakota.sandia.gov/licensing/votd/ht
ml-ref/RespCommands.html#RespFnLS",0,"nonlinear_inequality_constraints"},
          {"nonlinear_inequality_scales",0x80e,0,4,0,1743,0,0.,0.,0.,0,"{No
nlinear inequality scales} http://dakota.sandia.gov/licensing/votd/html-ref/RespC
ommands.html#RespFnLS",0,"nonlinear_inequality_constraints"},
          {"nonlinear_inequality_upper_bounds",14,0,2,0,1739,0,0.,0.,0.,0,"
{Nonlinear inequality upper bounds} http://dakota.sandia.gov/licensing/votd/html-
ref/RespCommands.html#RespFnLS",0,"nonlinear_inequality_constraints"}
          }
```

42.1.4.188 static KeyWord kw_185 () [static]

Initial value:

```
{
          {"calibration_data_file",11,5,4,0,1723,kw_182,0.,0.,0.,0,"{Calibr
ation data file name} http://dakota.sandia.gov/licensing/votd/html-ref/RespComman
ds.html#RespFnLS"},
          {"calibration_term_scale_types",0x80f,0,1,0,1717,0,0.,0.,0.,0,"{C
alibration scaling types} http://dakota.sandia.gov/licensing/votd/html-ref/RespCo
mmands.html#RespFnLS",0,"calibration_terms"},
          {"calibration_term_scales",0x80e,0,2,0,1719,0,0.,0.,0.,0,"{Calibr
ation scales} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#
RespFnLS",0,"calibration_terms"},
          {"calibration_weights",14,0,3,0,1721,0,0.,0.,0.,0,"{Calibration t
erm weights} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#R
espFnLS",0,"calibration_terms"},
          {"least_squares_data_file",3,5,4,0,1722,kw_182},
          {"least_squares_term_scale_types",0x807,0,1,0,1716,0,0.,0.,0.,0,0
,0,"calibration_terms"},
          {"least_squares_term_scales",0x806,0,2,0,1718,0,0.,0.,0.,0,0,0,"c
alibration_terms"},
          {"least_squares_weights",6,0,3,0,1720,0,0.,0.,0.,0,0,0,"calibrati
on_terms"},
          {"nonlinear_equality_constraints",0x29,3,6,0,1745,kw_183,0.,0.,0.
,0,"{Number of nonlinear equality constraints} http://dakota.sandia.gov/licensing
/votd/html-ref/RespCommands.html#RespFnLS"},
          {"nonlinear_inequality_constraints",0x29,4,5,0,1735,kw_184,0.,0.,
0.,0,"{Number of nonlinear inequality constraints} http://dakota.sandia.gov/licen
sing/votd/html-ref/RespCommands.html#RespFnLS"},
          {"num_nonlinear_equality_constraints",0x21,3,6,0,1744,kw_183},
```

```
                {"num_nonlinear_inequality_constraints",0x21,4,5,0,1734,kw_184}
                }
```

42.1.4.189 static KeyWord kw_186 () [static]

Initial value:

```
{
                {"ignore_bounds",8,0,1,0,1771,0,0.,0.,0.,0,"{Ignore variable boun
     ds} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespGradMi
     xed"}
                }
```

42.1.4.190 static KeyWord kw_187 () [static]

Initial value:

```
{
                {"central",8,0,6,0,1779,0,0.,0.,0.,0,"[CHOOSE difference interval
     ]"},
                {"dakota",8,1,4,0,1769,kw_186,0.,0.,0.,0,"@[CHOOSE gradient sourc
     e]"},
                {"fd_gradient_step_size",0x406,0,7,0,1780,0,0.,0.,0.001},
                {"fd_step_size",0x40e,0,7,0,1781,0,0.,0.,0.001,0,"{Finite differe
     nce step size} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html
     #RespGradMixed"},
                {"forward",8,0,6,0,1777,0,0.,0.,0.,0,"@"},
                {"id_analytic_gradients",13,0,2,2,1763,0,0.,0.,0.,0,"{Analytic de
     rivatives function list} http://dakota.sandia.gov/licensing/votd/html-ref/RespCom
     mands.html#RespGradMixed"},
                {"id_numerical_gradients",13,0,1,1,1761,0,0.,0.,0.,0,"{Numerical
     derivatives function list} http://dakota.sandia.gov/licensing/votd/html-ref/RespC
     ommands.html#RespGradMixed"},
                {"interval_type",8,0,5,0,1775,0,0.,0.,0.,0,"{Interval type} http:
     //dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespGradNum"},
                {"method_source",8,0,3,0,1767,0,0.,0.,0.,0,"{Method source} http:
     //dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespGradNum"},
                {"vendor",8,0,4,0,1773}
                }
```

42.1.4.191 static KeyWord kw_188 () [static]

Initial value:

```
{
                {"fd_hessian_step_size",6,0,1,0,1806},
                {"fd_step_size",14,0,1,0,1807,0,0.,0.,0.,0,"{Finite difference st
     ep size} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespH
     essMixed"}
                }
```

42.1.4.192 static KeyWord kw_189 () [static]

Initial value:

```
{
                {"damped",8,0,1,0,1817,0,0.,0.,0.,0,"{Numerical safeguarding of B
     FGS update} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#Re
     spHessMixed"}
                }
```

42.1.4.193 static KeyWord kw_190 () [static]

Initial value:

```
{
                {"bfgs",8,1,1,1,1815,kw_189,0.,0.,0.,0,"[CHOOSE Hessian approx.]"
     },
                {"sr1",8,0,1,1,1819}
                }
```

42.1.4.194 static KeyWord kw_191 () [static]

Initial value:

```
{
                {"central",8,0,2,0,1811,0,0.,0.,0.,0,"[CHOOSE difference interval
     ]"},
                {"forward",8,0,2,0,1809,0,0.,0.,0.,0,"@"},
                {"id_analytic_hessians",13,0,4,0,1821,0,0.,0.,0.,0,"{Analytic Hes
     sians function list} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommand
     s.html#RespHessMixed"},
                {"id_numerical_hessians",13,2,1,0,1805,kw_188,0.,0.,0.,0,"{Numeri
     cal Hessians function list} http://dakota.sandia.gov/licensing/votd/html-ref/Resp
     Commands.html#RespHessMixed"},
                {"id_quasi_hessians",13,2,3,0,1813,kw_190,0.,0.,0.,0,"{Quasi Hess
     ians function list} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands
     .html#RespHessMixed"}
                }
```

42.1.4.195 static KeyWord kw_192 () [static]

Initial value:

```
{
                {"nonlinear_equality_scale_types",0x80f,0,2,0,1711,0,0.,0.,0.,0,"
     {Nonlinear equality constraint scaling types} http://dakota.sandia.gov/licensing/
     votd/html-ref/RespCommands.html#RespFnOpt",0,"nonlinear_equality_constraints"},
                {"nonlinear_equality_scales",0x80e,0,3,0,1713,0,0.,0.,0.,0,"{Nonl
     inear equality constraint scales} http://dakota.sandia.gov/licensing/votd/html-re
     f/RespCommands.html#RespFnOpt",0,"nonlinear_equality_constraints"},
                {"nonlinear_equality_targets",14,0,1,0,1709,0,0.,0.,0.,0,"{Nonlin
     ear equality constraint targets} http://dakota.sandia.gov/licensing/votd/html-ref
     /RespCommands.html#RespFnOpt",0,"nonlinear_equality_constraints"}
                }
```

42.1.4.196 static KeyWord kw_193 () [static]

Initial value:

```
{
                {"nonlinear_inequality_lower_bounds",14,0,1,0,1699,0,0.,0.,0.,0,"
     {Nonlinear inequality constraint lower bounds} http://dakota.sandia.gov/licensing
     /votd/html-ref/RespCommands.html#RespFnOpt",0,"nonlinear_inequality_constraints"}
     ,
                {"nonlinear_inequality_scale_types",0x80f,0,3,0,1703,0,0.,0.,0.,0
     ,"{Nonlinear inequality constraint scaling types} http://dakota.sandia.gov/licens
     ing/votd/html-ref/RespCommands.html#RespFnOpt",0,"nonlinear_inequality_constraint
     s"},
                {"nonlinear_inequality_scales",0x80e,0,4,0,1705,0,0.,0.,0.,0,"{No
     nlinear inequality constraint scales} http://dakota.sandia.gov/licensing/votd/htm
     l-ref/RespCommands.html#RespFnOpt",0,"nonlinear_inequality_constraints"},
                {"nonlinear_inequality_upper_bounds",14,0,2,0,1701,0,0.,0.,0.,0,"
     {Nonlinear inequality constraint upper bounds} http://dakota.sandia.gov/licensing
     /votd/html-ref/RespCommands.html#RespFnOpt",0,"nonlinear_inequality_constraints"}

                }
```

42.1.4.197 static KeyWord kw_194 () [static]

Initial value:

```
{
                {"multi_objective_weights",14,0,3,0,1695,0,0.,0.,0.,0,"{Multiobje
     ctive weightings} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.h
     tml#RespFnOpt",0,"objective_functions"},
                {"nonlinear_equality_constraints",0x29,3,5,0,1707,kw_192,0.,0.,0.
     ,0,"{Number of nonlinear equality constraints} http://dakota.sandia.gov/licensing
     /votd/html-ref/RespCommands.html#RespFnOpt"},
                {"nonlinear_inequality_constraints",0x29,4,4,0,1697,kw_193,0.,0.,
     0.,0,"{Number of nonlinear inequality constraints} http://dakota.sandia.gov/licen
     sing/votd/html-ref/RespCommands.html#RespFnOpt"},
                {"num_nonlinear_equality_constraints",0x21,3,5,0,1706,kw_192},
                {"num_nonlinear_inequality_constraints",0x21,4,4,0,1696,kw_193},
                {"objective_function_scale_types",0x80f,0,1,0,1691,0,0.,0.,0.,0,"
     {Objective function scaling types} http://dakota.sandia.gov/licensing/votd/html-r
     ef/RespCommands.html#RespFnOpt",0,"objective_functions"},
                {"objective_function_scales",0x80e,0,2,0,1693,0,0.,0.,0.,0,"{Obje
     ctive function scales} http://dakota.sandia.gov/licensing/votd/html-ref/RespComma
     nds.html#RespFnOpt",0,"objective_functions"}
                }
```

42.1.4.198 static KeyWord kw_195 () [static]

Initial value:

```
{
                {"central",8,0,6,0,1779,0,0.,0.,0.,0,"[CHOOSE difference interval
     ]"},
                {"dakota",8,1,4,0,1769,kw_186,0.,0.,0.,0,"@[CHOOSE gradient sourc
     e]"},
                {"fd_gradient_step_size",0x406,0,7,0,1780,0,0.,0.,0.001},
```

```
                {"fd_step_size",0x40e,0,7,0,1781,0,0.,0.,0.001,0,"{Finite differe
nce step size} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html
#RespGradMixed"},
                {"forward",8,0,6,0,1777,0,0.,0.,0.,0,"@"},
                {"interval_type",8,0,5,0,1775,0,0.,0.,0.,0,"{Interval type} http:
//dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespGradNum"},
                {"method_source",8,0,3,0,1767,0,0.,0.,0.,0,"{Method source} http:
//dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespGradNum"},
                {"vendor",8,0,4,0,1773}
                }
```

42.1.4.199 static KeyWord kw_196 () [static]

Initial value:

```
{
                {"central",8,0,2,0,1791,0,0.,0.,0.,0,"[CHOOSE difference interval
]"},
                {"fd_hessian_step_size",6,0,1,0,1786},
                {"fd_step_size",14,0,1,0,1787,0,0.,0.,0.,0,"{Finite difference st
ep size} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespH
essNum"},
                {"forward",8,0,2,0,1789,0,0.,0.,0.,0,"@"}
                }
```

42.1.4.200 static KeyWord kw_197 () [static]

Initial value:

```
{
                {"damped",8,0,1,0,1797,0,0.,0.,0.,0,"{Numerical safeguarding of B
FGS update} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#Re
spHessQuasi"}
                }
```

42.1.4.201 static KeyWord kw_198 () [static]

Initial value:

```
{
                {"bfgs",8,1,1,1,1795,kw_197,0.,0.,0.,0,"[CHOOSE Hessian approx.]"
},
                {"sr1",8,0,1,1,1799}
                }
```

42.1.4.202 static KeyWord kw_199 () [static]

Initial value:

```
{
                {"analytic_gradients",8,0,4,2,1757,0,0.,0.,0.,0,"[CHOOSE gradient
```

```
 type]"},
          {"analytic_hessians",8,0,5,3,1801,0,0.,0.,0.,0,"[CHOOSE Hessian t
ype]"},
          {"calibration_terms",0x29,12,3,1,1715,kw_185,0.,0.,0.,0,"{Number
of calibration terms} http://dakota.sandia.gov/licensing/votd/html-ref/RespComman
ds.html#RespFnLS"},
          {"descriptors",15,0,2,0,1687,0,0.,0.,0.,0,"{Response labels} http
://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespLabels"},
          {"id_responses",11,0,1,0,1685,0,0.,0.,0.,0,"{Responses set identi
fier} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespSetI
d"},
          {"mixed_gradients",8,10,4,2,1759,kw_187,0.,0.,0.,0,"{Mixed gradie
nts} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespGradM
ixed"},
          {"mixed_hessians",8,5,5,3,1803,kw_191,0.,0.,0.,0,"{Mixed Hessians
} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespHessMixe
d"},
          {"no_gradients",8,0,4,2,1755,0,0.,0.,0.,0,"@"},
          {"no_hessians",8,0,5,3,1783,0,0.,0.,0.,0,"@"},
          {"num_least_squares_terms",0x21,12,3,1,1714,kw_185},
          {"num_objective_functions",0x21,7,3,1,1688,kw_194},
          {"num_response_functions",0x21,0,3,1,1752},
          {"numerical_gradients",8,8,4,2,1765,kw_195,0.,0.,0.,0,"{Numerical
 gradients} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#Re
spGradNum"},
          {"numerical_hessians",8,4,5,3,1785,kw_196,0.,0.,0.,0,"{Numerical
Hessians} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#Resp
HessNum"},
          {"objective_functions",0x29,7,3,1,1689,kw_194,0.,0.,0.,0,"{Number
 of objective functions} http://dakota.sandia.gov/licensing/votd/html-ref/RespCom
mands.html#RespFnOpt"},
          {"quasi_hessians",8,2,5,3,1793,kw_198,0.,0.,0.,0,"{Quasi Hessians
} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.html#RespHessQuas
i"},
          {"response_descriptors",7,0,2,0,1686},
          {"response_functions",0x29,0,3,1,1753,0,0.,0.,0.,0,"{Number of re
sponse functions} http://dakota.sandia.gov/licensing/votd/html-ref/RespCommands.h
tml#RespFnGen"}
          }
```

42.1.4.203 static KeyWord kw_200 () [static]

Initial value:

```
{
          {"method_list",15,0,1,1,33,0,0.,0.,0.,0,"{List of methods} http:/
/dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratHybrid"}
          }
```

42.1.4.204 static KeyWord kw_201 () [static]

Initial value:

```
{
          {"global_method_pointer",11,0,1,1,25,0,0.,0.,0.,0,"{Pointer to th
e global method specification} http://dakota.sandia.gov/licensing/votd/html-ref/S
tratCommands.html#StratHybrid"},
```

```
            {"local_method_pointer",11,0,2,2,27,0,0.,0.,0.,0,"{Pointer to the
 local method specification} http://dakota.sandia.gov/licensing/votd/html-ref/Str
atCommands.html#StratHybrid"},
            {"local_search_probability",10,0,3,0,29,0,0.,0.,0.,0,"{Probabilit
y of executing local searches} http://dakota.sandia.gov/licensing/votd/html-ref/S
tratCommands.html#StratHybrid"}
            }
```

42.1.4.205 static KeyWord kw_202 () [static]

Initial value:

```
{
            {"method_list",15,0,1,1,21,0,0.,0.,0.,0,"{List of methods} http:/
/dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratHybrid"}
            }
```

42.1.4.206 static KeyWord kw_203 () [static]

Initial value:

```
{
            {"collaborative",8,1,1,1,31,kw_200,0.,0.,0.,0,"[CHOOSE hybrid typ
e]{Collaborative hybrid} http://dakota.sandia.gov/licensing/votd/html-ref/StratCo
mmands.html#StratHybrid"},
            {"coupled",0,3,1,1,22,kw_201},
            {"embedded",8,3,1,1,23,kw_201,0.,0.,0.,0,"{Embedded hybrid} http:
//dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratHybrid"},
            {"sequential",8,1,1,1,19,kw_202,0.,0.,0.,0,"{Sequential hybrid} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratHybrid"},

            {"uncoupled",0,1,1,1,18,kw_202}
            }
```

42.1.4.207 static KeyWord kw_204 () [static]

Initial value:

```
{
            {"seed",9,0,1,0,41,0,0.,0.,0.,0,"{Seed for random starting points
} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratMultiS
tart"}
            }
```

42.1.4.208 static KeyWord kw_205 () [static]

Initial value:

```
{
            {"method_pointer",11,0,1,1,37,0,0.,0.,0.,0,"{Method pointer} http
://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratMultiStart"}
```

```
,
          {"random_starts",9,1,2,0,39,kw_204,0.,0.,0.,0,"{Number of random
starting points} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.h
tml#StratMultiStart"},
          {"starting_points",14,0,3,0,43,0,0.,0.,0.,0,"{List of user-specif
ied starting points} http://dakota.sandia.gov/licensing/votd/html-ref/StratComman
ds.html#StratMultiStart"}
          }
```

42.1.4.209 static KeyWord kw_206 () [static]

Initial value:

```
{
          {"seed",9,0,1,0,51,0,0.,0.,0.,0,"{Seed for random weighting sets}
 http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratParetoS
et"}
          }
```

42.1.4.210 static KeyWord kw_207 () [static]

Initial value:

```
{
          {"method_pointer",11,0,1,1,47,0,0.,0.,0.,0,"{Optimization method
pointer} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#Stra
tParetoSet"},
          {"multi_objective_weight_sets",6,0,3,0,52},
          {"opt_method_pointer",3,0,1,1,46},
          {"random_weight_sets",9,1,2,0,49,kw_206,0.,0.,0.,0,"{Number of ra
ndom weighting sets} http://dakota.sandia.gov/licensing/votd/html-ref/StratComman
ds.html#StratParetoSet"},
          {"weight_sets",14,0,3,0,53,0,0.,0.,0.,0,"{List of user-specified
weighting sets} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.ht
ml#StratParetoSet"}
          }
```

42.1.4.211 static KeyWord kw_208 () [static]

Initial value:

```
{
          {"method_pointer",11,0,1,0,57,0,0.,0.,0.,0,"{Method pointer} http
://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#StratSingle"}
          }
```

42.1.4.212 static KeyWord kw_209 () [static]

Initial value:

```
{
            {"tabular_graphics_file",11,0,1,0,7,0,0.,0.,0.,0,"{File name for
    tabular graphics data} http://dakota.sandia.gov/licensing/votd/html-ref/StratComm
    ands.html#StratIndControl"}
            }
```

42.1.4.213 static KeyWord kw_210 () [static]

Initial value:

```
{
            {"graphics",8,0,1,0,3,0,0.,0.,0.,0,"{Graphics flag} http://dakota
    .sandia.gov/licensing/votd/html-ref/StratCommands.html#StratIndControl"},
            {"hybrid",8,5,7,1,17,kw_203,0.,0.,0.,0,"[CHOOSE strategy type]{Hy
    brid strategy} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.htm
    l#StratHybrid"},
            {"iterator_self_scheduling",8,0,5,0,13,0,0.,0.,0.,0,"{Self-schedu
    ling of iterator jobs} http://dakota.sandia.gov/licensing/votd/html-ref/StratComm
    ands.html#StratIndControl"},
            {"iterator_servers",9,0,4,0,11,0,0.,0.,0.,0,"{Number of iterator
    servers} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#Stra
    tIndControl"},
            {"iterator_static_scheduling",8,0,6,0,15,0,0.,0.,0.,0,"{Static sc
    heduling of iterator jobs} http://dakota.sandia.gov/licensing/votd/html-ref/Strat
    Commands.html#StratIndControl"},
            {"multi_start",8,3,7,1,35,kw_205,0.,0.,0.,0,"{Multi-start iterati
    on strategy} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#
    StratMultiStart"},
            {"output_precision",0x29,0,3,0,9,0,0.,0.,0.,0,"{Numeric output pr
    ecision} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#Stra
    tIndControl"},
            {"pareto_set",8,5,7,1,45,kw_207,0.,0.,0.,0,"{Pareto set optimizat
    ion strategy} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html
    #StratParetoSet"},
            {"single_method",8,1,7,1,55,kw_208,0.,0.,0.,0,"@{Single method st
    rategy} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands.html#Strat
    Single"},
            {"tabular_graphics_data",8,1,2,0,5,kw_209,0.,0.,0.,0,"{Tabulation
     of graphics data} http://dakota.sandia.gov/licensing/votd/html-ref/StratCommands
    .html#StratIndControl"}
            }
```

42.1.4.214 static KeyWord kw_211 () [static]

Initial value:

```
{
            {"alphas",14,0,1,1,1435,0,0.,0.,0.,0,"{beta uncertain alphas} htt
    p://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Beta",0,"b
    eta_uncertain"},
            {"betas",14,0,2,2,1437,0,0.,0.,0.,0,"{beta uncertain betas} http:
    //dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Beta",0,"bet
    a_uncertain"},
            {"buv_alphas",6,0,1,1,1434,0,0.,0.,0.,0,0,0,"beta_uncertain"},
            {"buv_betas",6,0,2,2,1436,0,0.,0.,0.,0,0,0,"beta_uncertain"},
            {"buv_descriptors",7,0,5,0,1442,0,0.,0.,0.,0,0,0,"beta_uncertain"
    },
```

```
            {"buv_lower_bounds",6,0,3,3,1438,0,0.,0.,0.,0,0,0,"beta_uncertain
"},
            {"buv_upper_bounds",6,0,4,4,1440,0,0.,0.,0.,0,0,0,"beta_uncertain
"},
            {"descriptors",15,0,5,0,1443,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Beta",0,"beta_u
ncertain"},
            {"lower_bounds",14,0,3,3,1439,0,0.,0.,0.,0,"{Distribution lower b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Beta",0,"beta_uncertain"},
            {"upper_bounds",14,0,4,4,1441,0,0.,0.,0.,0,"{Distribution upper b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Beta",0,"beta_uncertain"}
            }
```

42.1.4.215 static KeyWord kw_212 () [static]

Initial value:

```
{
            {"descriptors",15,0,3,0,1501,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Binomial",0,"bi
nomial_uncertain"},
            {"num_trials",13,0,2,2,1499,0,0.,0.,0.,0,"{binomial uncertain num
_trials} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAU
V_Binomial",0,"binomial_uncertain"},
            {"prob_per_trial",14,0,1,1,1497,0,0.,0.,0.,0,"{binomial uncertain
 prob_per_trial} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.htm
l#VarDAUV_Binomial",0,"binomial_uncertain"}
            }
```

42.1.4.216 static KeyWord kw_213 () [static]

Initial value:

```
{
            {"cdv_descriptors",7,0,6,0,1338,0,0.,0.,0.,0,0,0,"continuous_desi
gn"},
            {"cdv_initial_point",6,0,1,0,1328,0,0.,0.,0.,0,0,0,"continuous_de
sign"},
            {"cdv_lower_bounds",6,0,2,0,1330,0,0.,0.,0.,0,0,0,"continuous_des
ign"},
            {"cdv_scale_types",0x807,0,4,0,1334,0,0.,0.,0.,0,0,0,"continuous_
design"},
            {"cdv_scales",0x806,0,5,0,1336,0,0.,0.,0.,0,0,0,"continuous_desig
n"},
            {"cdv_upper_bounds",6,0,3,0,1332,0,0.,0.,0.,0,0,0,"continuous_des
ign"},
            {"descriptors",15,0,6,0,1339,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCDV",0,"continuous_d
esign"},
            {"initial_point",14,0,1,0,1329,0,0.,0.,0.,0,"{Initial point} http
://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCDV",0,"continuo
us_design"},
            {"lower_bounds",14,0,2,0,1331,0,0.,0.,0.,0,"{Lower bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCDV",0,"continuous
_design"},
```

```
            {"scale_types",0x80f,0,4,0,1335,0,0.,0.,0.,0,"{Scaling types} htt
p://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCDV",0,"continu
ous_design"},
            {"scales",0x80e,0,5,0,1337,0,0.,0.,0.,0,"{Scales} http://dakota.s
andia.gov/licensing/votd/html-ref/VarCommands.html#VarCDV",0,"continuous_design"}
,
            {"upper_bounds",14,0,3,0,1333,0,0.,0.,0.,0,"{Upper bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCDV",0,"continuous
_design"}
            }
```

42.1.4.217 static KeyWord kw_214 () [static]

Initial value:

```
{
            {"csv_descriptors",7,0,4,0,1556,0,0.,0.,0.,0,0,0,"continuous_stat
e"},
            {"csv_initial_state",6,0,1,0,1550,0,0.,0.,0.,0,0,0,"continuous_st
ate"},
            {"csv_lower_bounds",6,0,2,0,1552,0,0.,0.,0.,0,0,0,"continuous_sta
te"},
            {"csv_upper_bounds",6,0,3,0,1554,0,0.,0.,0.,0,0,0,"continuous_sta
te"},
            {"descriptors",15,0,4,0,1557,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCSV",0,"continuous_s
tate"},
            {"initial_state",14,0,1,0,1551,0,0.,0.,0.,0,"{Initial states} htt
p://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCSV",0,"continu
ous_state"},
            {"lower_bounds",14,0,2,0,1553,0,0.,0.,0.,0,"{Lower bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCSV",0,"continuous
_state"},
            {"upper_bounds",14,0,3,0,1555,0,0.,0.,0.,0,"{Upper bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCSV",0,"continuous
_state"}
            }
```

42.1.4.218 static KeyWord kw_215 () [static]

Initial value:

```
{
            {"ddv_descriptors",7,0,4,0,1348,0,0.,0.,0.,0,0,0,"discrete_design
_range"},
            {"ddv_initial_point",5,0,1,0,1342,0,0.,0.,0.,0,0,0,"discrete_desi
gn_range"},
            {"ddv_lower_bounds",5,0,2,0,1344,0,0.,0.,0.,0,0,0,"discrete_desig
n_range"},
            {"ddv_upper_bounds",5,0,3,0,1346,0,0.,0.,0.,0,0,0,"discrete_desig
n_range"},
            {"descriptors",15,0,4,0,1349,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDRIV",0,"discrete_d
esign_range"},
            {"initial_point",13,0,1,0,1343,0,0.,0.,0.,0,"{Initial point} http
://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDRIV",0,"discre
te_design_range"},
```

```
            {"lower_bounds",13,0,2,0,1345,0,0.,0.,0.,0,"{Lower bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDRIV",0,"discrete
_design_range"},
            {"upper_bounds",13,0,3,0,1347,0,0.,0.,0.,0,"{Upper bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDRIV",0,"discrete
_design_range"}
            }
```

42.1.4.219 static KeyWord kw_216 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1359,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDSIV",0,"discrete_d
esign_set_integer"},
            {"initial_point",13,0,1,0,1353,0,0.,0.,0.,0,"{Initial point} http
://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDSIV",0,"discre
te_design_set_integer"},
            {"num_set_values",13,0,2,0,1355,0,0.,0.,0.,0,"{Number of values f
or each variable} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.ht
ml#VarDDSIV",0,"discrete_design_set_integer"},
            {"set_values",13,0,3,1,1357,0,0.,0.,0.,0,"{Set values} http://dak
ota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDSIV"}
            }
```

42.1.4.220 static KeyWord kw_217 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1369,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDSRV",0,"discrete_d
esign_set_real"},
            {"initial_point",14,0,1,0,1363,0,0.,0.,0.,0,"{Initial point} http
://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDSRV",0,"discre
te_design_set_real"},
            {"num_set_values",13,0,2,0,1365,0,0.,0.,0.,0,"{Number of values f
or each variable} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.ht
ml#VarDDSRV",0,"discrete_design_set_real"},
            {"set_values",14,0,3,1,1367,0,0.,0.,0.,0,"{Set values} http://dak
ota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDDSRV"}
            }
```

42.1.4.221 static KeyWord kw_218 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1567,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSRIV",0,"discrete_s
tate_range"},
            {"dsv_descriptors",7,0,4,0,1566,0,0.,0.,0.,0,0,0,"discrete_state_
range"},
```

```
            {"dsv_initial_state",5,0,1,0,1560,0,0.,0.,0.,0,0,0,"discrete_stat
e_range"},
            {"dsv_lower_bounds",5,0,2,0,1562,0,0.,0.,0.,0,0,0,"discrete_state
_range"},
            {"dsv_upper_bounds",5,0,3,0,1564,0,0.,0.,0.,0,0,0,"discrete_state
_range"},
            {"initial_state",13,0,1,0,1561,0,0.,0.,0.,0,"{Initial states} htt
p://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSRIV",0,"discr
ete_state_range"},
            {"lower_bounds",13,0,2,0,1563,0,0.,0.,0.,0,"{Lower bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSRIV",0,"discrete
_state_range"},
            {"upper_bounds",13,0,3,0,1565,0,0.,0.,0.,0,"{Upper bounds} http:/
/dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSRIV",0,"discrete
_state_range"}
            }
```

42.1.4.222 static KeyWord kw_219 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1577,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSSIV",0,"discrete_s
tate_set_integer"},
            {"initial_state",13,0,1,0,1571,0,0.,0.,0.,0,"{Initial state} http
://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSSIV",0,"discre
te_state_set_integer"},
            {"num_set_values",13,0,2,0,1573,0,0.,0.,0.,0,"{Number of values f
or each variable} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.ht
ml#VarDSSIV",0,"discrete_state_set_integer"},
            {"set_values",13,0,3,1,1575,0,0.,0.,0.,0,"{Set values} http://dak
ota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSSIV"}
            }
```

42.1.4.223 static KeyWord kw_220 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1587,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSSRV",0,"discrete_s
tate_set_real"},
            {"initial_state",14,0,1,0,1581,0,0.,0.,0.,0,"{Initial state} http
://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSSRV",0,"discre
te_state_set_real"},
            {"num_set_values",13,0,2,0,1583,0,0.,0.,0.,0,"{Number of values f
or each variable} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.ht
ml#VarDSSRV",0,"discrete_state_set_real"},
            {"set_values",14,0,3,1,1585,0,0.,0.,0.,0,"{Set values} http://dak
ota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDSSRV"}
            }
```

42.1.4.224 static KeyWord kw_221 () [static]

Initial value:

```
{
                {"betas",14,0,1,1,1429,0,0.,0.,0.,0,"{exponential uncertain betas
     } http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Expon
     ential",0,"exponential_uncertain"},
                {"descriptors",15,0,2,0,1431,0,0.,0.,0.,0,"{Descriptors} http://d
     akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Exponential",0,
     "exponential_uncertain"},
                {"euv_betas",6,0,1,1,1428,0,0.,0.,0.,0,0,0,"exponential_uncertain
     "},
                {"euv_descriptors",7,0,2,0,1430,0,0.,0.,0.,0,0,0,"exponential_unc
     ertain"}
                }
```

42.1.4.225 static KeyWord kw_222 () [static]

Initial value:

```
{
                {"alphas",14,0,1,1,1463,0,0.,0.,0.,0,"{frechet uncertain alphas}
     http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Frechet
     ",0,"frechet_uncertain"},
                {"betas",14,0,2,2,1465,0,0.,0.,0.,0,"{frechet uncertain betas} ht
     tp://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Frechet",
     0,"frechet_uncertain"},
                {"descriptors",15,0,3,0,1467,0,0.,0.,0.,0,"{Descriptors} http://d
     akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Frechet",0,"fre
     chet_uncertain"},
                {"fuv_alphas",6,0,1,1,1462,0,0.,0.,0.,0,0,0,"frechet_uncertain"},

                {"fuv_betas",6,0,2,2,1464,0,0.,0.,0.,0,0,0,"frechet_uncertain"},
                {"fuv_descriptors",7,0,3,0,1466,0,0.,0.,0.,0,0,0,"frechet_uncerta
     in"}
                }
```

42.1.4.226 static KeyWord kw_223 () [static]

Initial value:

```
{
                {"alphas",14,0,1,1,1447,0,0.,0.,0.,0,"{gamma uncertain alphas} ht
     tp://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Gamma",0,
     "gamma_uncertain"},
                {"betas",14,0,2,2,1449,0,0.,0.,0.,0,"{gamma uncertain betas} http
     ://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Gamma",0,"g
     amma_uncertain"},
                {"descriptors",15,0,3,0,1451,0,0.,0.,0.,0,"{Descriptors} http://d
     akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Gamma",0,"gamma
     _uncertain"},
                {"gauv_alphas",6,0,1,1,1446,0,0.,0.,0.,0,0,0,"gamma_uncertain"},
                {"gauv_betas",6,0,2,2,1448,0,0.,0.,0.,0,0,0,"gamma_uncertain"},
                {"gauv_descriptors",7,0,3,0,1450,0,0.,0.,0.,0,0,0,"gamma_uncertai
     n"}
                }
```

42.1.4.227 static KeyWord kw_224 () [static]

Initial value:

```
{
                {"descriptors",15,0,2,0,1515,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Geometric",0,"g
eometric_uncertain"},
                {"prob_per_trial",14,0,1,1,1513,0,0.,0.,0.,0,"{geometric uncertai
n prob_per_trial} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.ht
ml#VarDAUV_Geometric",0,"geometric_uncertain"}
                }
```

42.1.4.228 static KeyWord kw_225 () [static]

Initial value:

```
{
                {"alphas",14,0,1,1,1455,0,0.,0.,0.,0,"{gumbel uncertain alphas} h
ttp://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Gumbel",
0,"gumbel_uncertain"},
                {"betas",14,0,2,2,1457,0,0.,0.,0.,0,"{gumbel uncertain betas} htt
p://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Gumbel",0,
"gumbel_uncertain"},
                {"descriptors",15,0,3,0,1459,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Gumbel",0,"gumb
el_uncertain"},
                {"guuv_alphas",6,0,1,1,1454,0,0.,0.,0.,0,0,0,"gumbel_uncertain"},

                {"guuv_betas",6,0,2,2,1456,0,0.,0.,0.,0,0,0,"gumbel_uncertain"},
                {"guuv_descriptors",7,0,3,0,1458,0,0.,0.,0.,0,0,0,"gumbel_uncerta
in"}
                }
```

42.1.4.229 static KeyWord kw_226 () [static]

Initial value:

```
{
                {"abscissas",14,0,2,1,1481,0,0.,0.,0.,0,"{sets of abscissas for b
in-based histogram variables} http://dakota.sandia.gov/licensing/votd/html-ref/Va
rCommands.html#VarCAUV_Bin_Histogram"},
                {"counts",14,0,3,2,1485,0,0.,0.,0.,0,"{sets of counts for bin-bas
ed histogram variables} http://dakota.sandia.gov/licensing/votd/html-ref/VarComma
nds.html#VarCAUV_Bin_Histogram"},
                {"descriptors",15,0,4,0,1487,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Bin_Histogram",
0,"histogram_bin_uncertain"},
                {"huv_bin_abscissas",6,0,2,1,1480},
                {"huv_bin_counts",6,0,3,2,1484},
                {"huv_bin_descriptors",7,0,4,0,1486,0,0.,0.,0.,0,0,0,"histogram_b
in_uncertain"},
                {"huv_bin_ordinates",6,0,3,2,1482},
                {"huv_num_bin_pairs",5,0,1,0,1478,0,0.,0.,0.,0,0,0,"histogram_bin
_uncertain"},
                {"num_pairs",13,0,1,0,1479,0,0.,0.,0.,0,"{key to apportionment am
```

```
ong bin-based histogram variables} http://dakota.sandia.gov/licensing/votd/html-r
ef/VarCommands.html#VarCAUV_Bin_Histogram",0,"histogram_bin_uncertain"},
          {"ordinates",14,0,3,2,1483,0,0.,0.,0.,0,"{sets of ordinates for b
in-based histogram variables} http://dakota.sandia.gov/licensing/votd/html-ref/Va
rCommands.html#VarCAUV_Bin_Histogram"}
          }
```

42.1.4.230 static KeyWord kw_227 () [static]

Initial value:

```
{
          {"abscissas",14,0,2,1,1531,0,0.,0.,0.,0,"{sets of abscissas for p
oint-based histogram variables} http://dakota.sandia.gov/licensing/votd/html-ref/
VarCommands.html#VarDAUV_Point_Histogram"},
          {"counts",14,0,3,2,1533,0,0.,0.,0.,0,"{sets of counts for point-b
ased histogram variables} http://dakota.sandia.gov/licensing/votd/html-ref/VarCom
mands.html#VarDAUV_Point_Histogram"},
          {"descriptors",15,0,4,0,1535,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Point_Histogram
",0,"histogram_point_uncertain"},
          {"huv_num_point_pairs",5,0,1,0,1528,0,0.,0.,0.,0,0,0,"histogram_p
oint_uncertain"},
          {"huv_point_abscissas",6,0,2,1,1530},
          {"huv_point_counts",6,0,3,2,1532},
          {"huv_point_descriptors",7,0,4,0,1534,0,0.,0.,0.,0,0,0,"histogram
_point_uncertain"},
          {"num_pairs",13,0,1,0,1529,0,0.,0.,0.,0,"{key to apportionment am
ong point-based histogram variables} http://dakota.sandia.gov/licensing/votd/html
-ref/VarCommands.html#VarDAUV_Point_Histogram",0,"histogram_point_uncertain"}
          }
```

42.1.4.231 static KeyWord kw_228 () [static]

Initial value:

```
{
          {"descriptors",15,0,4,0,1525,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Hypergeometric"
,0,"hypergeometric_uncertain"},
          {"num_drawn",13,0,3,3,1523,0,0.,0.,0.,0,"{hypergeometric uncertai
n num_drawn } http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#V
arDAUV_Hypergeometric",0,"hypergeometric_uncertain"},
          {"selected_population",13,0,2,2,1521,0,0.,0.,0.,0,"{hypergeometri
c uncertain selected_population} http://dakota.sandia.gov/licensing/votd/html-ref
/VarCommands.html#VarDAUV_Hypergeometric",0,"hypergeometric_uncertain"},
          {"total_population",13,0,1,1,1519,0,0.,0.,0.,0,"{hypergeometric u
ncertain total_population} http://dakota.sandia.gov/licensing/votd/html-ref/VarCo
mmands.html#VarDAUV_Hypergeometric",0,"hypergeometric_uncertain"}
          }
```

42.1.4.232 static KeyWord kw_229 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1547,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCEUV_Interval",0,"in
terval_uncertain"},
            {"interval_bounds",14,0,3,2,1545,0,0.,0.,0.,0,"{bounds per interv
al} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCEUV_Int
erval"},
            {"interval_probs",14,0,2,1,1543,0,0.,0.,0.,0,"{basic probability
assignments per interval} http://dakota.sandia.gov/licensing/votd/html-ref/VarCom
mands.html#VarCEUV_Interval"},
            {"iuv_descriptors",7,0,4,0,1546,0,0.,0.,0.,0,0,0,"interval_uncert
ain"},
            {"iuv_interval_bounds",6,0,3,2,1544},
            {"iuv_interval_probs",6,0,2,1,1542},
            {"iuv_num_intervals",5,0,1,0,1540,0,0.,0.,0.,0,0,0,"interval_unce
rtain"},
            {"num_intervals",13,0,1,0,1541,0,0.,0.,0.,0,"{number of intervals
 defined for each interval variable} http://dakota.sandia.gov/licensing/votd/html
-ref/VarCommands.html#VarCEUV_Interval",0,"interval_uncertain"}
            }
```

42.1.4.233 static KeyWord kw_230 () [static]

Initial value:

```
{
            {"lnuv_zetas",6,0,1,1,1386,0,0.,0.,0.,0,0,0,"lognormal_uncertain"
},
            {"zetas",14,0,1,1,1387,0,0.,0.,0.,0,"{lognormal uncertain zetas}
http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Lognorm
al",0,"lognormal_uncertain"}
            }
```

42.1.4.234 static KeyWord kw_231 () [static]

Initial value:

```
{
            {"error_factors",14,0,1,1,1393,0,0.,0.,0.,0,"[CHOOSE variance spe
c.]{lognormal uncertain error factors} http://dakota.sandia.gov/licensing/votd/ht
ml-ref/VarCommands.html#VarCAUV_Lognormal",0,"lognormal_uncertain"},
            {"lnuv_error_factors",6,0,1,1,1392,0,0.,0.,0.,0,0,0,"lognormal_un
certain"},
            {"lnuv_std_deviations",6,0,1,1,1390,0,0.,0.,0.,0,0,0,"lognormal_u
ncertain"},
            {"std_deviations",14,0,1,1,1391,0,0.,0.,0.,0,"@{lognormal uncerta
in standard deviations} http://dakota.sandia.gov/licensing/votd/html-ref/VarComma
nds.html#VarCAUV_Lognormal",0,"lognormal_uncertain"}
            }
```

42.1.4.235 static KeyWord kw_232 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1399,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Lognormal",0,"l
ognormal_uncertain"},
            {"lambdas",14,2,1,1,1385,kw_230,0.,0.,0.,0,"[CHOOSE characterizat
ion]{lognormal uncertain lambdas} http://dakota.sandia.gov/licensing/votd/html-re
f/VarCommands.html#VarCAUV_Lognormal",0,"lognormal_uncertain"},
            {"lnuv_descriptors",7,0,4,0,1398,0,0.,0.,0.,0,0,0,"lognormal_unce
rtain"},
            {"lnuv_lambdas",6,2,1,1,1384,kw_230,0.,0.,0.,0,0,0,"lognormal_unc
ertain"},
            {"lnuv_lower_bounds",6,0,2,0,1394,0,0.,0.,0.,0,0,0,"lognormal_unc
ertain"},
            {"lnuv_means",6,4,1,1,1388,kw_231,0.,0.,0.,0,0,0,"lognormal_uncer
tain"},
            {"lnuv_upper_bounds",6,0,3,0,1396,0,0.,0.,0.,0,0,0,"lognormal_unc
ertain"},
            {"lower_bounds",14,0,2,0,1395,0,0.,0.,0.,0,"{Distribution lower b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Lognormal",0,"lognormal_uncertain"},
            {"means",14,4,1,1,1389,kw_231,0.,0.,0.,0,"@{lognormal uncertain m
eans} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_L
ognormal",0,"lognormal_uncertain"},
            {"upper_bounds",14,0,3,0,1397,0,0.,0.,0.,0,"{Distribution upper b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Lognormal",0,"lognormal_uncertain"}
            }
```

42.1.4.236 static KeyWord kw_233 () [static]

Initial value:

```
{
            {"descriptors",15,0,3,0,1415,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Loguniform",0,"
loguniform_uncertain"},
            {"lower_bounds",14,0,1,1,1411,0,0.,0.,0.,0,"{Distribution lower b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Loguniform",0,"loguniform_uncertain"},
            {"luuv_descriptors",7,0,3,0,1414,0,0.,0.,0.,0,0,0,"loguniform_unc
ertain"},
            {"luuv_lower_bounds",6,0,1,1,1410,0,0.,0.,0.,0,0,0,"loguniform_un
certain"},
            {"luuv_upper_bounds",6,0,2,2,1412,0,0.,0.,0.,0,0,0,"loguniform_un
certain"},
            {"upper_bounds",14,0,2,2,1413,0,0.,0.,0.,0,"{Distribution upper b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Loguniform",0,"loguniform_uncertain"}
            }
```

42.1.4.237 static KeyWord kw_234 () [static]

Initial value:

```
{
            {"descriptors",15,0,3,0,1509,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Negative_Binomi
```

```
al",0,"negative_binomial_uncertain"},
          {"num_trials",13,0,2,2,1507,0,0.,0.,0.,0,"{negative binomial unce
rtain success num_trials} http://dakota.sandia.gov/licensing/votd/html-ref/VarCom
mands.html#VarDAUV_Negative_Binomial",0,"negative_binomial_uncertain"},
          {"prob_per_trial",14,0,1,1,1505,0,0.,0.,0.,0,"{negative binomial
uncertain success prob_per_trial} http://dakota.sandia.gov/licensing/votd/html-re
f/VarCommands.html#VarDAUV_Negative_Binomial",0,"negative_binomial_uncertain"}
          }
```

42.1.4.238 static KeyWord kw_235 () [static]

Initial value:

```
{
          {"descriptors",15,0,5,0,1381,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Normal",0,"norm
al_uncertain"},
          {"lower_bounds",14,0,3,0,1377,0,0.,0.,0.,0,"{Distribution lower b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Normal",0,"normal_uncertain"},
          {"means",14,0,1,1,1373,0,0.,0.,0.,0,"{normal uncertain means} htt
p://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Normal",0,
"normal_uncertain"},
          {"nuv_descriptors",7,0,5,0,1380,0,0.,0.,0.,0,0,0,"normal_uncertai
n"},
          {"nuv_lower_bounds",6,0,3,0,1376,0,0.,0.,0.,0,0,0,"normal_uncerta
in"},
          {"nuv_means",6,0,1,1,1372,0,0.,0.,0.,0,0,0,"normal_uncertain"},
          {"nuv_std_deviations",6,0,2,2,1374,0,0.,0.,0.,0,0,0,"normal_uncer
tain"},
          {"nuv_upper_bounds",6,0,4,0,1378,0,0.,0.,0.,0,0,0,"normal_uncerta
in"},
          {"std_deviations",14,0,2,2,1375,0,0.,0.,0.,0,"{normal uncertain s
tandard deviations} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.
html#VarCAUV_Normal",0,"normal_uncertain"},
          {"upper_bounds",14,0,4,0,1379,0,0.,0.,0.,0,"{Distribution upper b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Normal",0,"normal_uncertain"}
          }
```

42.1.4.239 static KeyWord kw_236 () [static]

Initial value:

```
{
          {"descriptors",15,0,2,0,1493,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Poisson",0,"poi
sson_uncertain"},
          {"lambdas",14,0,1,1,1491,0,0.,0.,0.,0,"{poisson uncertain lambdas
} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarDAUV_Poiss
on",0,"poisson_uncertain"}
          }
```

42.1.4.240 static KeyWord kw_237 () [static]

Initial value:

```
{
            {"descriptors",15,0,4,0,1425,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Triangular",0,"
triangular_uncertain"},
            {"lower_bounds",14,0,2,2,1421,0,0.,0.,0.,0,"{Distribution lower b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Triangular",0,"triangular_uncertain"},
            {"modes",14,0,1,1,1419,0,0.,0.,0.,0,"{triangular uncertain modes}
 http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Triang
ular",0,"triangular_uncertain"},
            {"tuv_descriptors",7,0,4,0,1424,0,0.,0.,0.,0,0,0,"triangular_unce
rtain"},
            {"tuv_lower_bounds",6,0,2,2,1420,0,0.,0.,0.,0,0,0,"triangular_unc
ertain"},
            {"tuv_modes",6,0,1,1,1418,0,0.,0.,0.,0,0,0,"triangular_uncertain"
},
            {"tuv_upper_bounds",6,0,3,3,1422,0,0.,0.,0.,0,0,0,"triangular_unc
ertain"},
            {"upper_bounds",14,0,3,3,1423,0,0.,0.,0.,0,"{Distribution upper b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Triangular",0,"triangular_uncertain"}
            }
```

42.1.4.241 static KeyWord kw_238 () [static]

Initial value:

```
{
            {"descriptors",15,0,3,0,1407,0,0.,0.,0.,0,"{Descriptors} http://d
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Uniform",0,"uni
form_uncertain"},
            {"lower_bounds",14,0,1,1,1403,0,0.,0.,0.,0,"{Distribution lower b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Uniform",0,"uniform_uncertain"},
            {"upper_bounds",14,0,2,2,1405,0,0.,0.,0.,0,"{Distribution upper b
ounds} http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_
Uniform",0,"uniform_uncertain"},
            {"uuv_descriptors",7,0,3,0,1406,0,0.,0.,0.,0,0,0,"uniform_uncerta
in"},
            {"uuv_lower_bounds",6,0,1,1,1402,0,0.,0.,0.,0,0,0,"uniform_uncert
ain"},
            {"uuv_upper_bounds",6,0,2,2,1404,0,0.,0.,0.,0,0,0,"uniform_uncert
ain"}
            }
```

42.1.4.242 static KeyWord kw_239 () [static]

Initial value:

```
{
            {"alphas",14,0,1,1,1471,0,0.,0.,0.,0,"{weibull uncertain alphas}
http://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Weibull
",0,"weibull_uncertain"},
            {"betas",14,0,2,2,1473,0,0.,0.,0.,0,"{weibull uncertain betas} ht
tp://dakota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Weibull",
0,"weibull_uncertain"},
            {"descriptors",15,0,3,0,1475,0,0.,0.,0.,0,"{Descriptors} http://d
```

```
akota.sandia.gov/licensing/votd/html-ref/VarCommands.html#VarCAUV_Weibull",0,"wei
bull_uncertain"},
          {"wuv_alphas",6,0,1,1,1470,0,0.,0.,0.,0,0,0,"weibull_uncertain"},

          {"wuv_betas",6,0,2,2,1472,0,0.,0.,0.,0,0,0,"weibull_uncertain"},
          {"wuv_descriptors",7,0,3,0,1474,0,0.,0.,0.,0,0,0,"weibull_uncerta
in"}
          }
```

42.1.4.243 static KeyWord kw_241 () [static]

Initial value:

```
{
          {"interface",0x308,10,5,5,1589,kw_9,0.,0.,0.,0,"{Interface} An in
terface specifies how function evaluations will be performed in order to map a se
t of parameters into a set of responses. http://www.cs.sandia.gov/DAKOTA/licensin
g/votd/html-ref/InterfCommands.html"},
          {"method",0x308,75,2,2,59,kw_156,0.,0.,0.,0,"{Method} A method sp
ecifies the name and controls of an iterative procedure, e.g., a sensitivity anal
ysis, uncertainty quantification, or optimization method. http://www.cs.sandia.go
v/DAKOTA/licensing/votd/html-ref/MethodCommands.html"},
          {"model",8,6,3,3,1145,kw_181,0.,0.,0.,0,"{Model} A model consists
 of a model type and maps specified variables through an interface to generate re
sponses. http://www.cs.sandia.gov/DAKOTA/licensing/votd/html-ref/ModelCommands.ht
ml"},
          {"responses",0x308,18,6,6,1683,kw_199,0.,0.,0.,0,"{Responses} A r
esponses object specifies the data that can be returned to DAKOTA through the int
erface after the completion of a function evaluation. http://www.cs.sandia.gov/DA
KOTA/licensing/votd/html-ref/RespCommands.html"},
          {"strategy",0x108,10,1,1,1,kw_210,0.,0.,0.,0,"{Strategy} The stra
tegy specifies the top level technique which will govern the management of iterat
ors and models in the solution of the problem of interest. http://www.cs.sandia.g
ov/DAKOTA/licensing/votd/html-ref/StratCommands.html"},
          {"variables",0x308,29,4,4,1323,kw_240,0.,0.,0.,0,"{Variables} A v
ariables object specifies the parameter set to be iterated by a particular method
. http://www.cs.sandia.gov/DAKOTA/licensing/votd/html-ref/VarCommands.html"}
          }
```

42.1.4.244 KeyWord kw_242[4] [static]

Initial value:

```
{
          {"descriptors",15,0,4,0,0,0.,0.,0,N_vae(dailbl,DAUIVar_hypergeome
tric)},
          {"num_drawn",13,0,3,3,0,0.,0.,0,N_vam(intDL,hyperGeomUncNumDrawn)
},
          {"selected_population",13,0,2,2,0,0.,0.,0,N_vam(intDL,hyperGeomUn
cSelectedPop)},
          {"total_population",13,0,1,1,0,0.,0.,0,N_vam(intDL,hyperGeomUncTo
talPop)}
          }
```

42.1.4.245 KeyWord kw_243[8] [static]

Initial value:

```
{
                {"descriptors",15,0,4,0,0,0.,0.,0,N_vae(ceulbl,CEUVar_interval)},

                {"interval_bounds",14,0,3,2,0,0.,0.,0,N_vam(vrl,Var_Info_Ivb)},
                {"interval_probs",14,0,2,1,0,0.,0.,0,N_vam(vrl,Var_Info_Ivp)},
                {"iuv_descriptors",7,0,4,0,0,0.,0.,-3,N_vae(ceulbl,CEUVar_interva
     l)},
                {"iuv_interval_bounds",6,0,3,2,0,0.,0.,-3,N_vam(vrl,Var_Info_Ivb)
     },
                {"iuv_interval_probs",6,0,2,1,0,0.,0.,-3,N_vam(vrl,Var_Info_Ivp)}
     ,
                {"iuv_num_intervals",5,0,1,0,0,0.,0.,1,N_vam(vil,Var_Info_nIv)},
                {"num_intervals",13,0,1,0,0,0.,0.,0,N_vam(vil,Var_Info_nIv)}
                }
```

42.1.4.246 KeyWord kw_244[2] [static]

Initial value:

```
{
                {"lnuv_zetas",6,0,1,1,0,0.,0.,1,N_vam(RealLb,lognormalUncZetas)},

                {"zetas",14,0,1,1,0,0.,0.,0,N_vam(RealLb,lognormalUncZetas)}
                }
```

42.1.4.247 KeyWord kw_245[4] [static]

Initial value:

```
{
                {"error_factors",14,0,1,1,0,0.,0.,0,N_vam(RealLb,lognormalUncErrF
     acts)},
                {"lnuv_error_factors",6,0,1,1,0,0.,0.,-1,N_vam(RealLb,lognormalUn
     cErrFacts)},
                {"lnuv_std_deviations",6,0,1,1,0,0.,0.,1,N_vam(RealLb,lognormalUn
     cStdDevs)},
                {"std_deviations",14,0,1,1,0,0.,0.,0,N_vam(RealLb,lognormalUncStd
     Devs)}
                }
```

42.1.4.248 KeyWord kw_246[10] [static]

Initial value:

```
{
                {"descriptors",15,0,4,0,0,0.,0.,0,N_vae(caulbl,CAUVar_lognormal)}
     ,
                {"lambdas",14,2,1,1,kw_244,0.,0.,0,N_vam(RealLd,lognormalUncLambd
     as)},
```

```
                {"lnuv_descriptors",7,0,4,0,0,0.,0.,-2,N_vae(caulbl,CAUVar_lognor
mal)},
                {"lnuv_lambdas",6,2,1,1,kw_244,0.,0.,-2,N_vam(RealLd,lognormalUnc
Lambdas)},
                {"lnuv_lower_bounds",6,0,2,0,0,0.,0.,3,N_vam(RealLb,lognormalUncL
owerBnds)},
                {"lnuv_means",6,4,1,1,kw_245,0.,0.,3,N_vam(RealLb,lognormalUncMea
ns)},
                {"lnuv_upper_bounds",6,0,3,0,0,0.,0.,3,N_vam(RealUb,lognormalUncU
pperBnds)},
                {"lower_bounds",14,0,2,0,0,0.,0.,0,N_vam(RealLb,lognormalUncLower
Bnds)},
                {"means",14,4,1,1,kw_245,0.,0.,0,N_vam(RealLb,lognormalUncMeans)}
,
                {"upper_bounds",14,0,3,0,0,0.,0.,0,N_vam(RealUb,lognormalUncUpper
Bnds)}
                }
```

42.1.4.249 KeyWord kw_247[6] [static]

Initial value:

```
{
                {"descriptors",15,0,3,0,0,0.,0.,0,N_vae(caulbl,CAUVar_loguniform)
},
                {"lower_bounds",14,0,1,1,0,0.,0.,0,N_vam(RealLb,loguniformUncLowe
rBnds)},
                {"luuv_descriptors",7,0,3,0,0,0.,0.,-2,N_vae(caulbl,CAUVar_loguni
form)},
                {"luuv_lower_bounds",6,0,1,1,0,0.,0.,-2,N_vam(RealLb,loguniformUn
cLowerBnds)},
                {"luuv_upper_bounds",6,0,2,2,0,0.,0.,1,N_vam(RealUb,loguniformUnc
UpperBnds)},
                {"upper_bounds",14,0,2,2,0,0.,0.,0,N_vam(RealUb,loguniformUncUppe
rBnds)}
                }
```

42.1.4.250 KeyWord kw_248[3] [static]

Initial value:

```
{
                {"descriptors",15,0,3,0,0,0.,0.,0,N_vae(dailbl,DAUIVar_negative_b
inomial)},
                {"num_trials",13,0,2,2,0,0.,0.,0,N_vam(intDL,negBinomialUncNumTri
als)},
                {"prob_per_trial",14,0,1,1,0,0.,0.,0,N_vam(RealLd,negBinomialUncP
robPerTrial)}
                }
```

42.1.4.251 KeyWord kw_249[10] [static]

Initial value:

```
{
                {"descriptors",15,0,5,0,0,0.,0.,0,N_vae(caulbl,CAUVar_normal)},
                {"lower_bounds",14,0,3,0,0,0.,0.,0,N_vam(RealLd,normalUncLowerBnd
     s)},
                {"means",14,0,1,1,0,0.,0.,0,N_vam(RealLd,normalUncMeans)},
                {"nuv_descriptors",7,0,5,0,0,0.,0.,-3,N_vae(caulbl,CAUVar_normal)
     },
                {"nuv_lower_bounds",6,0,3,0,0,0.,0.,-3,N_vam(RealLd,normalUncLowe
     rBnds)},
                {"nuv_means",6,0,1,1,0,0.,0.,-3,N_vam(RealLd,normalUncMeans)},
                {"nuv_std_deviations",6,0,2,2,0,0.,0.,2,N_vam(RealLb,normalUncStd
     Devs)},
                {"nuv_upper_bounds",6,0,4,0,0,0.,0.,2,N_vam(RealLd,normalUncUpper
     Bnds)},
                {"std_deviations",14,0,2,2,0,0.,0.,0,N_vam(RealLb,normalUncStdDev
     s)},
                {"upper_bounds",14,0,4,0,0,0.,0.,0,N_vam(RealLd,normalUncUpperBnd
     s)}
                }
```

42.1.4.252 KeyWord kw_250[2] [static]

Initial value:

```
{
                {"descriptors",15,0,2,0,0,0.,0.,0,N_vae(dailbl,DAUIVar_poisson)},

                {"lambdas",14,0,1,1,0,0.,0.,0,N_vam(RealLd,poissonUncLambdas)}
                }
```

42.1.4.253 KeyWord kw_251[8] [static]

Initial value:

```
{
                {"descriptors",15,0,4,0,0,0.,0.,0,N_vae(caulbl,CAUVar_triangular)
     },
                {"lower_bounds",14,0,2,2,0,0.,0.,0,N_vam(RealLb,triangularUncLowe
     rBnds)},
                {"modes",14,0,1,1,0,0.,0.,0,N_vam(RealLd,triangularUncModes)},
                {"tuv_descriptors",7,0,4,0,0,0.,0.,-3,N_vae(caulbl,CAUVar_triangu
     lar)},
                {"tuv_lower_bounds",6,0,2,2,0,0.,0.,-3,N_vam(RealLb,triangularUnc
     LowerBnds)},
                {"tuv_modes",6,0,1,1,0,0.,0.,-3,N_vam(RealLd,triangularUncModes)}
     ,
                {"tuv_upper_bounds",6,0,3,3,0,0.,0.,1,N_vam(RealUb,triangularUncU
     pperBnds)},
                {"upper_bounds",14,0,3,3,0,0.,0.,0,N_vam(RealUb,triangularUncUppe
     rBnds)}
                }
```

42.1.4.254 KeyWord kw_252[6] [static]

Initial value:

```
{
                {"descriptors",15,0,3,0,0,0.,0.,0,N_vae(caulbl,CAUVar_uniform)},
                {"lower_bounds",14,0,1,1,0,0.,0.,0,N_vam(RealLb,uniformUncLowerBn
     ds)},
                {"upper_bounds",14,0,2,2,0,0.,0.,0,N_vam(RealUb,uniformUncUpperBn
     ds)},
                {"uuv_descriptors",7,0,3,0,0,0.,0.,-3,N_vae(caulbl,CAUVar_uniform
     )},
                {"uuv_lower_bounds",6,0,1,1,0,0.,0.,-3,N_vam(RealLb,uniformUncLow
     erBnds)},
                {"uuv_upper_bounds",6,0,2,2,0,0.,0.,-3,N_vam(RealUb,uniformUncUpp
     erBnds)}
                }
```

42.1.4.255 KeyWord kw_253[6] [static]

Initial value:

```
{
                {"alphas",14,0,1,1,0,0.,0.,0,N_vam(RealLb,weibullUncAlphas)},
                {"betas",14,0,2,2,0,0.,0.,0,N_vam(RealLb,weibullUncBetas)},
                {"descriptors",15,0,3,0,0,0.,0.,0,N_vae(caulbl,CAUVar_weibull)},
                {"wuv_alphas",6,0,1,1,0,0.,0.,-3,N_vam(RealLb,weibullUncAlphas)},

                {"wuv_betas",6,0,2,2,0,0.,0.,-3,N_vam(RealLb,weibullUncBetas)},
                {"wuv_descriptors",7,0,3,0,0,0.,0.,-3,N_vae(caulbl,CAUVar_weibull
     )}
                }
```

42.1.4.256 KeyWord kw_255[6] [static]

Initial value:

```
{
                {"interface",0x308,10,5,5,kw_9,0.,0.,0,N_ifm3(start,0,stop)},
                {"method",0x308,75,2,2,kw_169,0.,0.,0,N_mdm3(start,0,stop)},
                {"model",8,6,3,3,kw_195,0.,0.,0,N_mom3(start,0,stop)},
                {"responses",0x308,18,6,6,kw_213,0.,0.,0,N_rem3(start,0,stop)},
                {"strategy",0x108,10,1,1,kw_224,0.,0.,0,NIDRProblemDescDB::strate
     gy_start},
                {"variables",0x308,29,4,4,kw_254,0.,0.,0,N_vam3(start,0,stop)}
                }
```

42.1.4.257 Var_uinfo CAUVLbl[CAUVar_Nkinds] [static]

Initial value:

```
{
       UncInfo(nuv_, Normal),
       UncInfo(lnuv_, Lognormal),
       UncInfo(uuv_, Uniform),
       UncInfo(luuv_, Loguniform),
       UncInfo(tuv_, Triangular),
       UncInfo(euv_, Exponential),
```

```
        UncInfo(beuv_, Beta),
        UncInfo(gauv_, Gamma),
        UncInfo(guuv_, Gumbel),
        UncInfo(fuv_, Frechet),
        UncInfo(wuv_, Weibull),
        UncInfo(hbuv_, HistogramBin)
        }
```

42.1.4.258 Var_uinfo DAUIVLbl[DAUIVar_Nkinds] [static]

Initial value:

```
{
        UncInfo(puv_, Poisson),
        UncInfo(biuv_, Binomial),
        UncInfo(nbuv_, NegBinomial),
        UncInfo(geuv_, Geometric),
        UncInfo(hguv_, HyperGeom)
        }
```

42.1.4.259 Var_uinfo DAURVLbl[DAURVar_Nkinds] [static]

Initial value:

```
{
        UncInfo(hpuv_, HistogramPt)
        }
```

42.1.4.260 Var_uinfo CEUVLbl[CEUVar_Nkinds] [static]

Initial value:

```
{
        UncInfo(iuv_, Interval)
        }
```

42.1.4.261 Var_uinfo DiscSetLbl[DiscSetVar_Nkinds] [static]

Initial value:

```
{
        DiscSetInfo(ddsiv_, DesSetInt),
        DiscSetInfo(ddsrv_, DesSetReal),
        DiscSetInfo(dssiv_, StateSetInt),
        DiscSetInfo(dssrv_, StateSetReal)
        }
```

42.1.4.262 VarLabelChk Vlch[ ] [static]

Initial value:

```
{
        { AVI numContinuousDesVars, AVI continuousDesignLabels, "cdv_", "cdv_des
     criptors" },
        { AVI numDiscreteDesRangeVars, AVI discreteDesignRangeLabels, "ddriv_",
     "ddriv_descriptors" },
        { AVI numDiscreteDesSetIntVars, AVI discreteDesignSetIntLabels, "ddsiv_"
     , "ddsiv_descriptors" },
        { AVI numDiscreteDesSetRealVars, AVI discreteDesignSetRealLabels, "ddsrv
     _", "ddsrv_descriptors" },
        { AVI numContinuousStateVars, AVI continuousStateLabels, "csv_", "csv_de
     scriptors" },
        { AVI numDiscreteStateRangeVars, AVI discreteStateRangeLabels, "dsriv_",
      "dsriv_descriptors" },
        { AVI numDiscreteStateSetIntVars, AVI discreteStateSetIntLabels, "dssiv_
     ", "dssiv_descriptors" },
        { AVI numDiscreteStateSetRealVars, AVI discreteStateSetRealLabels, "dssr
     v_", "dssrv_descriptors" },
        { AVI numContinuousDesVars, AVI continuousDesignScaleTypes, 0, "cdv_scal
     e_types" }
        }
```

42.1.4.263 VLstuff VLS[N_VLS] [static]

Initial value:

```
{
{CAUVar_Nkinds,  1, AVI CAUv,  CAUVLbl,
       DVR continuousAleatoryUncLabels,
       DVR continuousAleatoryUncLowerBnds,
       DVR continuousAleatoryUncUpperBnds,
       DVR continuousAleatoryUncVars},
{CEUVar_Nkinds,  1, AVI CEUv,  CEUVLbl,
       DVR continuousEpistemicUncLabels,
       DVR continuousEpistemicUncLowerBnds,
       DVR continuousEpistemicUncUpperBnds,
       DVR continuousEpistemicUncVars},
{DAUIVar_Nkinds, 0, AVI DAUIv, DAUIVLbl,
       DVR  discreteIntAleatoryUncLabels,
       RDVR discreteIntAleatoryUncLowerBnds,
       RDVR discreteIntAleatoryUncUpperBnds,
       RDVR discreteIntAleatoryUncVars},
{DAURVar_Nkinds, 1, AVI DAURv, DAURVLbl,
       DVR discreteRealAleatoryUncLabels,
       DVR discreteRealAleatoryUncLowerBnds,
       DVR discreteRealAleatoryUncUpperBnds,
       DVR discreteRealAleatoryUncVars}}
```

42.1.4.264 Var_bgen var_mp_bgen[ ] [static]

Initial value:

```
{
```

```
                Vchu0(gamma_uncertain,numGammaUncVars,Gamma),
                Vchu0(gumbel_uncertain,numGumbelUncVars,Gumbel),
                Vchu0(frechet_uncertain,numFrechetUncVars,Frechet),
                Vchu0(weibull_uncertain,numWeibullUncVars,Weibull),
                Vchu0(histogram_bin_uncertain,numHistogramBinUncVars,HistogramBin
    )
                }
```

42.1.4.265 Var_bgen var_mp_bgen_audr[ ] [static]

Initial value:

```
{
                Vchu0(histogram_point_uncertain,numHistogramPtUncVars,HistogramPt
    )
                }
```

42.1.4.266 Var_bgen var_mp_bgen_audi[ ] [static]

Initial value:

```
{
                Vchu0(poisson_uncertain,numPoissonUncVars,Poisson),
                Vchu0(binomial_uncertain,numBinomialUncVars,Binomial),
                Vchu0(negative_binomial_uncertain,numNegBinomialUncVars,NegBinomi
    al),
                Vchu0(geometric_uncertain,numGeometricUncVars,Geometric),
                Vchu0(hypergeometric_uncertain,numHyperGeomUncVars,HyperGeom)
                }
```

42.1.4.267 Var_bgen var_mp_bgen_eu[ ] [static]

Initial value:

```
{
                Vchu0(interval_uncertain,numIntervalUncVars,Interval)
                }
```

42.1.4.268 Var_bgen var_mp_bgen_dis[ ] [static]

Initial value:

```
{
                Vchu0(discrete_design_set_integer,numDiscreteDesSetIntVars,DDSI),

                Vchu0(discrete_design_set_real,numDiscreteDesSetRealVars,DDSR),
                Vchu0(discrete_state_set_integer,numDiscreteStateSetIntVars,DSSI)
    ,
                Vchu0(discrete_state_set_real,numDiscreteStateSetRealVars,DSSR)
                }
```

42.1.4.269 VarBgen Bgen[ ] [static]

Initial value:

```
{
        BgenInit(var_mp_bgen_audr),
        BgenInit(var_mp_bgen_audi),
        BgenInit(var_mp_bgen_eu)}
```

42.1.4.270 Var_bchk var_mp_bndchk[ ] [static]

Initial value:

```
{
        Vchv(continuous_design,numContinuousDesVars,continuousDesign),
        Vchu1(normal_uncertain,numNormalUncVars,normalUnc),
        Vchu1(lognormal_uncertain,numLognormalUncVars,lognormalUnc),
        Vchu(uniform_uncertain,numUniformUncVars,uniformUnc),
        Vchu(loguniform_uncertain,numLoguniformUncVars,loguniformUnc),
        Vchu(triangular_uncertain,numTriangularUncVars,triangularUnc),
        Vchu0(exponential_uncertain,numExponentialUncVars,Exponential),
        Vchu(beta_uncertain,numBetaUncVars,betaUnc),
        Vchv(continuous_state,numContinuousStateVars,continuousState)
        }
```

42.1.4.271 Var_ibchk var_mp_ibndchk[ ] [static]

Initial value:

```
{
        Vchi(discrete_design_range,numDiscreteDesRangeVars,discreteDesignRange),
        Vchi(discrete_state_range,numDiscreteStateRangeVars,discreteStateRange)
        }
```

42.2 SIM Namespace Reference

A sample namespace for derived classes that use assign_rep() to plug facilities into DAKOTA.

Classes

- class ParallelDirectApplicInterface

  Sample derived interface class for testing parallel simulator plug-ins using assign_rep().

- class SerialDirectApplicInterface

  Sample derived interface class for testing serial simulator plug-ins using assign_rep().

42.2.1 Detailed Description

A sample namespace for derived classes that use assign_rep() to plug facilities into DAKOTA. A typical use of plug-ins with assign_rep() is to publish a simulation interface for use in library mode See Interfacing with DAKOTA as a Library for more information.

Chapter 43

Class Documentation

43.1 ActiveSet Class Reference

Container class for active set tracking information. Contains the active set request vector and the derivative variables vector.

Public Member Functions

- ActiveSet ()

  default constructor

- ActiveSet (size_t num_fns, size_t num_deriv_vars)

  standard constructor

- ActiveSet (size_t num_fns)

  partial constructor

- ActiveSet (const ActiveSet &set)

  copy constructor

- ~ActiveSet ()

  destructor

- ActiveSet & operator= (const ActiveSet &set)

  assignment operator

- void reshape (size_t num_fns, size_t num_deriv_vars)

  reshape requestVector and derivVarsVector

- void reshape (size_t num_fns)

  reshape requestVector

- const ShortArray & request_vector () const

  return the request vector

- void request_vector (const ShortArray &rv)

  set the request vector

- void request_values (const short rv_val)

  set all request vector values

- short request_value (const size_t index) const

  get the value of an entry in the request vector

- void request_value (const short rv_val, const size_t index)

  set the value of an entry in the request vector

- const SizetArray & derivative_vector () const

  return the derivative variables vector

- void derivative_vector (const SizetArray &dvv)

  set the derivative variables vector from a SizetArray

- void derivative_vector (SizetMultiArrayConstView dvv)

  set the derivative variables vector from a SizetMultiArrayConstView

- void derivative_start_value (size_t dvv_start_val)

  set the derivative variables vector values

- void read (std::istream &s)

  read an active set object from an std::istream

- void write (std::ostream &s) const

  write an active set object to an std::ostream

- void write_annotated (std::ostream &s) const

  write an active set object to an std::ostream in annotated format

- void read (BiStream &s)

  read an active set object from the binary restart stream

- void write (BoStream &s) const

  write an active set object to the binary restart stream

- void read (MPIUnpackBuffer &s)

  read an active set object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write an active set object to a packed MPI buffer

Private Attributes

- ShortArray requestVector

  the vector of response requests

- SizetArray derivVarsVector

  the vector of variable ids used for computing derivatives

Friends

- bool operator== (const ActiveSet &set1, const ActiveSet &set2)

  equality operator

- bool operator!= (const ActiveSet &set1, const ActiveSet &set2)

  inequality operator

43.1.1 Detailed Description

Container class for active set tracking information. Contains the active set request vector and the derivative variables vector. The ActiveSet class is a small class whose initial design function is to avoid having to pass the ASV and DVV separately. It is not part of a class hierarchy and does not employ reference-counting/ representation-sharing idioms (e.g., handle-body).

43.1.2 Member Data Documentation

43.1.2.1 ShortArray requestVector `[private]`

the vector of response requests It uses a 0 value for inactive functions and sums 1 (value), 2 (gradient), and 4 (Hessian) for active functions.

Referenced by ActiveSet::ActiveSet(), ActiveSet::operator=(), Dakota::operator==(), ActiveSet::read(), ActiveSet::request_value(), ActiveSet::request_values(), ActiveSet::request_vector(), ActiveSet::reshape(), ActiveSet::write(), and ActiveSet::write_annotated().

43.1.2.2 SizetArray derivVarsVector `[private]`

the vector of variable ids used for computing derivatives These ids will generally identify either the active continuous variables or the inactive continuous variables.

Referenced by ActiveSet::ActiveSet(), ActiveSet::derivative_start_value(), ActiveSet::derivative_vector(), ActiveSet::operator=(), Dakota::operator==(), ActiveSet::read(), ActiveSet::reshape(), ActiveSet::write(), and ActiveSet::write_annotated().

The documentation for this class was generated from the following files:

- DakotaActiveSet.H
- DakotaActiveSet.C

43.2 AnalysisCode Class Reference

Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations. Inheritance diagram for AnalysisCode::

Public Member Functions

- void define_filenames (const int id)

  define modified filenames from user input by handling Unix temp file and tagging options

- void write_parameters_files (const Variables &vars, const ActiveSet &set, const Response &response, const int id)

  write the parameters data and response request data to one or more parameters files (using one or more invocations of write_parameters_file()) in either standard or aprepro format

- void read_results_files (Response &response, const int id)

  read the response object from one or more results files

- const std::vector< String > & program_names () const

  return programNames

- const std::string & input_filter_name () const

  return iFilterName

- const std::string & output_filter_name () const

  return oFilterName

- const std::string & parameters_filename () const

  return paramsFileName

- const std::string & results_filename () const

  return resultsFileName

- const std::string & results_filename (const int id)

  return the results filename entry in fileNameMap corresponding to id

- void suppress_output_flag (const bool flag)

  set suppressOutputFlag

- bool suppress_output_flag () const

  return suppressOutputFlag

- bool command_line_arguments () const

  return commandLineArgs

- bool multiple_parameters_filenames () const

  return multipleParamsFiles

- void file_cleanup () const

  remove temporary files if not fileSaveFlag

Protected Member Functions

- AnalysisCode (const ProblemDescDB &problem_db)

  constructor

- ~AnalysisCode ()

  destructor

- const char * work_dir () const

  return Workdir if useWorkdir is true (only called by derived classes)

Protected Attributes

- bool suppressOutputFlag

  flag set by master processor to suppress output from slave processors

- short outputLevel

  output verbosity level: {SILENT,QUIET,NORMAL,VERBOSE,DEBUG}_OUTPUT

- bool fileTagFlag

  flags tagging of parameter/results files

- bool fileSaveFlag

  flags retention of parameter/results files

- bool commandLineArgs

  flag indicating use of passing of filenames as command line arguments to the analysis drivers and input/output filters

- bool apreproFlag

  flag indicating use of the APREPRO (the Sandia "A PRE PROcessor" utility) format for parameter files

- bool multipleParamsFiles

  flag indicating the need for separate parameters files for multiple analysis drivers

- std::string iFilterName

  the name of the input filter (input_filter user specification)

- std::string oFilterName

  the name of the output filter (output_filter user specification)

- std::vector< String > programNames

  the names of the analysis code programs (analysis_drivers user specification)

- size_t numPrograms

  the number of analysis code programs (length of programNames)

- std::string specifiedParamsFileName

  the name of the parameters file from user specification

- std::string paramsFileName

  the parameters file name actually used (modified with tagging or temp files)

- std::string specifiedResultsFileName

  the name of the results file from user specification

- std::string resultsFileName

  the results file name actually used (modified with tagging or temp files)

- bool allowExistingResults

  by default analysis code interfaces delete results files if they exist; user may override with this flag and we'll try to gather and only fork if needed

- std::string curWorkdir

  working directory when useWorkdir is true

- std::map< int, std::pair< std::string, std::string > > fileNameMap

  stores parameters and results file names used in spawning function evaluations. Map key is the function evaluation identifier.

- bool useWorkdir

  whether to use a new or specified work_directory

- std::string workDir

  its name, if specified...

- bool dirTag

whether to tag the working directory

- bool dirSave

  whether dir_save was specified

- bool dirDel

  whether to delete the directory when Dakota terminates

- std::string templateDir

  template directory (if specified)

- StringArray templateFiles

  template files (if specified)

- bool templateCopy

  whether to force a copy (versus link) every time

- bool templateReplace

  whether to replace existing files

- bool haveTemplateDir

  state variable for template directory

- bool haveWorkdir

  for dirTag, whether we have workDir

- std::string dakDir

  Dakota directory (if needed).

Private Member Functions

- void write_parameters_file (const Variables &vars, const ActiveSet &set, const Response &response, const std::string &prog, const std::vector< String > &an_comps, const std::string &params_fname)

  write the variables, active set vector, derivative variables vector, and analysis components to the specified parameters file in either standard or aprepro format

Private Attributes

- ParallelLibrary & parallelLib

  reference to the ParallelLibrary object. Used in define_filenames().

- String2DArray analysisComponents

  the set of optional analysis components used by the analysis drivers (from the analysis_components interface specification)

43.2.1 Detailed Description

Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations. The AnalysisCode class hierarchy provides simulation spawning services for ApplicationInterface derived classes and alleviates these classes of some of the specifics of simulation code management. The hierarchy does not employ the letter-envelope technique since the ApplicationInterface derived classes instantiate the appropriate derived AnalysisCode class directly.

The documentation for this class was generated from the following files:

- AnalysisCode.H
- AnalysisCode.C

43.3 Analyzer Class Reference

Base class for NonD, DACE, and ParamStudy branches of the iterator hierarchy. Inheritance diagram for Analyzer::

Public Member Functions

- const VariablesArray & all_variables ()

  return the complete set of evaluated variables

- const RealMatrix & all_samples ()

  return the complete set of evaluated samples

- const IntResponseMap & all_responses () const

  return the complete set of computed responses

- virtual void vary_pattern (bool pattern_flag)

  sets varyPattern in derived classes that support it

Protected Member Functions

- Analyzer ()

  default constructor

- Analyzer (Model &model)

  standard constructor

- Analyzer (NoDBBaseConstructor, Model &model)

alternate constructor for instantiations "on the fly" with a Model

- Analyzer (NoDBBaseConstructor)

  alternate constructor for instantiations "on the fly" without a Model

- ~Analyzer ()

  destructor

- virtual void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

- virtual void update_model_from_sample (Model &model, const Real ∗sample_vars)

  update model's current variables with data from sample

- virtual void update_model_from_variables (Model &model, const Variables &vars)

  update model's current variables with data from vars

- void pre_output ()
- void print_results (std::ostream &s)

  print the final iterator results

- const Variables & variables_results () const

  return a single final iterator solution (variables)

- const Response & response_results () const

  return a single final iterator solution (response)

- const VariablesArray & variables_array_results ()

  return multiple final iterator solutions (variables). This should only be used if returns_multiple_points() returns true.

- const ResponseArray & response_array_results ()

  return multiple final iterator solutions (response). This should only be used if returns_multiple_points() returns true.

- void response_results_active_set (const ActiveSet &set)

  set the requested data for the final iterator response results

- bool compact_mode () const

  returns Analyzer::compactMode

- bool returns_multiple_points () const

  indicates if this iterator returns multiple final points. Default return is false. Override to return true if appropriate.

- void evaluate_parameter_sets (Model &model, bool log_resp_flag, bool log_best_flag)

  perform function evaluations to map parameter sets (allVariables) into response sets (allResponses)

- void variance_based_decomp (int ncont, int ndiscint, int ndiscreal, int num_samples)
- void read_variables_responses (int num_evals, size_t num_vars)

  convenience function for reading variables/responses (used in derived classes post_input)

- void print_sobol_indices (std::ostream &s) const

  Printing of VBD results.

- void sample_to_variables (const Real *sample_c_vars, Variables &vars)

  convert samples array to variables array; e.g., allSamples to allVariables

- void samples_to_variables_array (const RealMatrix &sample_matrix, VariablesArray &vars_array)

  convert samples array to variables array; e.g., allSamples to allVariables

- void variables_array_to_samples (const VariablesArray &vars_array, RealMatrix &sample_matrix)

  convert variables array to samples array; e.g., allVariables to allSamples

Protected Attributes

- bool compactMode

  switch for allSamples (compact mode) instead of allVariables (normal mode)

- VariablesArray allVariables

  array of all variables to be evaluated in evaluate_parameter_sets()

- RealMatrix allSamples

  compact alternative to allVariables

- IntResponseMap allResponses

  array of all responses to be computed in evaluate_parameter_sets()

- StringArray allHeaders

  array of headers to insert into output while evaluating allVariables

- size_t numObjFns

  number of objective functions

- size_t numLSqTerms

  number of least squares terms

- RealPairPRPMultiMap bestVarsRespMap

  map which stores best set of solutions

Private Member Functions

- void compute_best_metrics (const Response &response, std::pair< Real, Real > &metrics)

  compares current evaluation to best evaluation and updates best

- void update_best (const Variables &vars, int eval_id, const Response &response)

  compares current evaluation to best evaluation and updates best

- void update_best (const Real *sample_c_vars, int eval_id, const Response &response)

  compares current evaluation to best evaluation and updates best

Private Attributes

- Real vbdDropTol

  tolerance for omitting output of small VBD indices

- RealVectorArray S4

  VBD main effect indices.

- RealVectorArray T4

  VBD total effect indices.

43.3.1 Detailed Description

Base class for NonD, DACE, and ParamStudy branches of the iterator hierarchy. The Analyzer class provides common data and functionality for various types of systems analysis, including nondeterministic analysis, design of experiments, and parameter studies.

43.3.2 Member Function Documentation

43.3.2.1 void pre_output () [protected, virtual]

Generate tabular output with active variables (compactMode) or all variables with their labels and response labels, with no data. Variables are sequenced {cv, div, drv}

Reimplemented from Iterator.

References Analyzer::allSamples, Analyzer::allVariables, ParallelLibrary::command_line_pre_run_output(), ParallelLibrary::command_line_user_modes(), Analyzer::compactMode, Model::current_response(), Model::current_variables(), Dakota::filename(), Iterator::iteratedModel, Iterator::outputLevel, Model::parallel_library(), Dakota::write_data_tabular(), Dakota::write_precision, and Iterator::writePrecision.

43.3.2.2 void print_results (std::ostream & *s*) [protected, virtual]

print the final iterator results This virtual function provides additional iterator-specific final results outputs beyond the function evaluation summary printed in finalize_run().

Reimplemented from Iterator.

Reimplemented in PStudyDACE, Verification, NonDExpansion, NonDGlobalReliability, NonDGPMSABayesCalibration, NonDIncremLHSSampling, NonDInterval, NonDLHSSampling, NonDLocalReliability, and RichExtrapVerification.

References Analyzer::bestVarsRespMap, ParamResponsePair::eval_id(), Response::function_values(), Analyzer::numLSqTerms, Analyzer::numObjFns, ParamResponsePair::prp_parameters(), ParamResponsePair::prp_response(), and Dakota::write_data_partial().

43.3.2.3 void evaluate_parameter_sets (Model & *model*, bool *log_resp_flag*, bool *log_best_flag*) [protected]

perform function evaluations to map parameter sets (allVariables) into response sets (allResponses) Convenience function for derived classes with sets of function evaluations to perform (e.g., NonDSampling, DDACEDesignCompExp, FSUDesignCompExp, ParamStudy).

References Iterator::activeSet, Analyzer::allHeaders, Analyzer::allResponses, Analyzer::allSamples, Analyzer::allVariables, Model::asynch_compute_response(), Iterator::asynchFlag, Analyzer::compactMode, Model::compute_response(), Response::copy(), Model::current_response(), Model::current_variables(), Model::evaluation_id(), Model::synchronize(), Analyzer::update_best(), Analyzer::update_model_from_sample(), and Analyzer::update_model_from_variables().

Referenced by NonDSparseGrid::evaluate_set(), PSUADEDesignCompExp::extract_trends(), ParamStudy::extract_trends(), FSUDesignCompExp::extract_trends(), DDACEDesignCompExp::extract_trends(), NonDLHSSampling::quantify_uncertainty(), NonDIntegration::quantify_uncertainty(), NonDIncremLHSSampling::quantify_uncertainty(), NonDAdaptImpSampling::quantify_uncertainty(), and Analyzer::variance_based_decomp().

43.3.2.4 void variance_based_decomp (int *ncont*, int *ndiscint*, int *ndiscreal*, int *num_samples*) [protected]

Calculation of sensitivity indices obtained by variance based decomposition. These indices are obtained by the Saltelli version of the Sobol VBD which uses (K+2)∗N function evaluations, where K is the number of dimensions (uncertain vars) and N is the number of samples.

References Dakota::abort_handler(), Analyzer::allResponses, Analyzer::allSamples, Analyzer::allVariables, Analyzer::compactMode, Variables::continuous_variables(), Dakota::copy_data(), Variables::discrete_int_variables(), Variables::discrete_real_variables(), Analyzer::evaluate_parameter_sets(), Analyzer::get_parameter_sets(), Iterator::iteratedModel, Iterator::numFunctions, Analyzer::S4, Analyzer::T4, and Analyzer::vary_pattern().

Referenced by FSUDesignCompExp::extract_trends(), DDACEDesignCompExp::extract_trends(), and NonDLHSSampling::quantify_uncertainty().

43.3.2.5 void read_variables_responses (int *num_evals*, size_t *num_vars*) [protected]

convenience function for reading variables/responses (used in derived classes post_input) read num_evals variables/responses from file

References Dakota::abort_handler(), Analyzer::allResponses, Analyzer::allSamples, Analyzer::allVariables, ParallelLibrary::command_line_post_run_input(), ParallelLibrary::command_line_user_modes(), Analyzer::compactMode, Response::copy(), Variables::copy(), Model::current_response(), Model::current_variables(), Dakota::filename(), Iterator::iteratedModel, Iterator::outputLevel, Model::parallel_library(), and Analyzer::update_best().

Referenced by PSUADEDesignCompExp::post_input(), ParamStudy::post_input(), NonDLHSSampling::post_input(), FSUDesignCompExp::post_input(), and DDACEDesignCompExp::post_input().

43.3.2.6 void print_sobol_indices (std::ostream & *s*) const [protected]

Printing of VBD results. printing of variance based decomposition indices.

References Model::continuous_variable_labels(), Model::discrete_int_variable_labels(), Model::discrete_real_variable_labels(), Iterator::iteratedModel, Iterator::numContinuousVars, Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, Iterator::numFunctions, Model::response_labels(), Analyzer::S4, Analyzer::T4, Analyzer::vbdDropTol, and Dakota::write_precision.

Referenced by NonDLHSSampling::print_results(), and PStudyDACE::print_results().

The documentation for this class was generated from the following files:

- DakotaAnalyzer.H
- DakotaAnalyzer.C

43.4 ApplicationInterface Class Reference

Derived class within the interface class hierarchy for supporting interfaces to simulation codes. Inheritance diagram for ApplicationInterface::

Public Member Functions

- ApplicationInterface (const ProblemDescDB &problem_db)

  constructor

- ~ApplicationInterface ()

  destructor

Protected Member Functions

- void init_communicators (const IntArray &message_lengths, const int &max_iterator_concurrency)

  allocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

- void set_communicators (const IntArray &message_lengths)

  set the local parallel partition data for an interface (the partitions are already allocated in ParallelLibrary).

- void free_communicators ()

  deallocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

- void init_serial ()
- int asynch_local_evaluation_concurrency () const

  return asynchLocalEvalConcurrency

- String interface_synchronization () const

  return interfaceSynchronization

- bool evaluation_cache () const

  return evalCacheFlag

- void map (const Variables &vars, const ActiveSet &set, Response &response, const bool asynch_flag=false)

  Provides a "mapping" of variables to responses using a simulation. Protected due to Interface letter-envelope idiom.

- void manage_failure (const Variables &vars, const ActiveSet &set, Response &response, int failed_eval_id)

  manages a simulation failure using abort/retry/recover/continuation

- const IntResponseMap & synch ()

  executes a blocking schedule for asynchronous evaluations in the beforeSynchCorePRPQueue and returns all jobs

- const IntResponseMap & synch_nowait ()

  executes a nonblocking schedule for asynchronous evaluations in the beforeSynchCorePRPQueue and returns a partial set of completed jobs

- void serve_evaluations ()

  run on evaluation servers to serve the iterator master

- void stop_evaluation_servers ()

  used by the iterator master to terminate evaluation servers

- virtual void derived_map (const Variables &vars, const ActiveSet &set, Response &response, int fn_eval_id)

  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- virtual void derived_map_asynch (const ParamResponsePair &pair)

  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- virtual void derived_synch (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- virtual void derived_synch_nowait (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- void self_schedule_analyses ()

  blocking self-schedule of all analyses within a function evaluation using message passing

- void serve_analyses_synch ()

  serve the master analysis scheduler and manage one synchronous analysis job at a time

- virtual int derived_synchronous_local_analysis (const int &analysis_id)

Execute a particular analysis (identified by analysis_id) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve_analyses_synch().

Protected Attributes

- ParallelLibrary & parallelLib

  reference to the ParallelLibrary object used to manage MPI partitions for the concurrent evaluations and concurrent analyses parallelism levels

- bool suppressOutput

  flag for suppressing output on slave processors

- int evalCommSize

  size of evalComm

- int evalCommRank

  processor rank within evalComm

- int evalServerId

  evaluation server identifier

- bool eaDedMasterFlag

  flag for dedicated master partitioning at ea level

- int analysisCommSize

  size of analysisComm

- int analysisCommRank

  processor rank within analysisComm

- int analysisServerId

  analysis server identifier

- int numAnalysisServers

  number of analysis servers

- bool multiProcAnalysisFlag

  flag for multiprocessor analysis partitions

- bool asynchLocalAnalysisFlag

  flag for asynchronous local parallelism of analyses

- int asynchLocalAnalysisConcurrency

  limits the number of concurrent analyses in asynchronous local scheduling and specifies hybrid concurrency when message passing

- int numAnalysisDrivers

  the number of analysis drivers used for each function evaluation (from the analysis_drivers interface specification)

- IntSet completionSet

  the set of completed fn_eval_id's populated by derived_synch() and derived_synch_nowait()

Private Member Functions

- bool duplication_detect (const Variables &vars, Response &response, const bool asynch_flag)

  checks data_pairs and beforeSynchCorePRPQueue to see if the current evaluation request has already been performed or queued

- void self_schedule_evaluations ()

  blocking self-schedule of all evaluations in beforeSynchCorePRPQueue using message passing; executes on iteratorComm master

- void static_schedule_evaluations ()

  blocking static schedule of all evaluations in beforeSynchCorePRPQueue using message passing; executes on iteratorComm master

- void asynchronous_local_evaluations (PRPQueue &prp_queue)

  perform all jobs in prp_queue using asynchronous approaches on the local processor

- void asynchronous_local_evaluations_static (PRPQueue &prp_queue)

  perform all the jobs in prp_queue using asynchronous approaches on the local processor, but schedule statically such that eval_id is always replaced with an equivalent one, modulo asynchLocalEvalConcurrency

- void synchronous_local_evaluations (PRPQueue &prp_queue)

  perform all jobs in prp_queue using synchronous approaches on the local processor

- void asynchronous_local_evaluations_nowait (PRPQueue &prp_queue)

  launch new jobs in prp_queue asynchronously (if capacity is available), perform nonblocking query of all running jobs, and process any completed jobs (handles both local self- and local static-scheduling cases)

- void serve_evaluations_synch ()

  serve the evaluation message passing schedulers and perform one synchronous evaluation at a time

- void serve_evaluations_asynch ()

  serve the evaluation message passing schedulers and manage multiple asynchronous evaluations

- void serve_evaluations_peer ()

  serve the evaluation message passing schedulers and perform one synchronous evaluation at a time as part of the 1st peer

- void set_evaluation_communicators (const IntArray &message_lengths)

  convenience function for updating the local evaluation partition data following ParallelLibrary::init_evaluation_communicators().

- void set_analysis_communicators ()

  convenience function for updating the local analysis partition data following ParallelLibrary::init_analysis_communicators().

- void check_configuration (const int &max_iterator_concurrency)

  perform some error checks on the parallel configuration

- const ParamResponsePair & get_source_pair (const Variables &target_vars)

  convenience function for the continuation approach in manage_failure() for finding the nearest successful "source" evaluation to the failed "target"

- void continuation (const Variables &target_vars, const ActiveSet &set, Response &response, const ParamResponsePair &source_pair, int failed_eval_id)

  performs a 0th order continuation method to step from a successful "source" evaluation to the failed "target". Invoked by manage_failure() for failAction == "continuation".

- void common_input_filtering (const Variables &vars)

  common input filtering operations, e.g. mesh movement

- void common_output_filtering (Response &response)

  common output filtering operations, e.g. data filtering

Private Attributes

- int worldSize

  size of MPI_COMM_WORLD

- int worldRank

  processor rank within MPI_COMM_WORLD

- int iteratorCommSize

  size of iteratorComm

- int iteratorCommRank

  processor rank within iteratorComm

- bool ieMessagePass

  flag for message passing at ie scheduling level

- int numEvalServers

  number of evaluation servers

- bool eaMessagePass

  flag for message passing at ea scheduling level

- int procsPerAnalysis

  processors per analysis servers

- int lenVarsMessage

  length of a MPIPackBuffer containing a Variables object; computed in Model::init_communicators()

- int lenVarsActSetMessage

  length of a MPIPackBuffer containing a Variables object and an ActiveSet object; computed in Model::init_communicators()

- int lenResponseMessage

  length of a MPIPackBuffer containing a Response object; computed in Model::init_communicators()

- int lenPRPairMessage

  length of a MPIPackBuffer containing a ParamResponsePair object; computed in Model::init_communicators()

- String evalScheduling

  user specification of evaluation scheduling algorithm (self, static, or no spec). Used for manual overrides of the auto-configure logic in ParallelLibrary::resolve_inputs().

- String analysisScheduling

  user specification of analysis scheduling algorithm (self, static, or no spec). Used for manual overrides of the auto-configure logic in ParallelLibrary::resolve_inputs().

- int asynchLocalEvalConcurrency

  limits the number of concurrent evaluations in asynchronous local scheduling and specifies hybrid concurrency when message passing

- bool asynchLocalEvalStatic

  whether the asynchronous local evaluations are to be performed with a static schedule (default false)

- IntArray localServerJobMap

  array with one entry per local "server" indicating the job (fn_eval_id) currently running on the server (used for asynchronour local static schedules)

- String interfaceSynchronization

  interface synchronization specification: synchronous (default) or asynchronous

- bool headerFlag

  used by synch_nowait to manage output frequency (since this function may be called many times prior to any completions)

- bool asvControlFlag

used to manage a user request to deactivate the active set vector control. true = modify the ASV each evaluation as appropriate (default); false = ASV values are static so that the user need not check them on each evaluation.

- bool evalCacheFlag

  used to manage a user request to deactivate the function evaluation cache (i.e., queries and insertions using the data_pairs cache).

- bool restartFileFlag

  used to manage a user request to deactivate the restart file (i.e., insertions into write_restart).

- ShortArray defaultASV

  the static ASV values used when the user has selected asvControl = off

- String failAction

  mitigation action for captured simulation failures: abort, retry, recover, or continuation

- int failRetryLimit

  limit on the number of retries for the retry failAction

- RealVector failRecoveryFnVals

  the dummy function values used for the recover failAction

- IntResponseMap historyDuplicateMap

  used to bookkeep asynchronous evaluations which duplicate data_pairs evaluations. Map key is evalIdCntr, map value is corresponding response.

- std::map< int, std::pair< PRPQueueHIter, Response > > beforeSynchDuplicateMap

  used to bookkeep evalIdCntr, beforeSynchCorePRPQueue iterator, and response of asynchronous evaluations which duplicate queued beforeSynchCorePRPQueue evaluations

- PRPQueue beforeSynchCorePRPQueue

  used to bookkeep vars/set/response of nonduplicate asynchronous core evaluations. This is the queue of jobs populated by asynchronous map() that is later scheduled in synch() or synch_nowait().

- PRPQueue beforeSynchAlgPRPQueue

  used to bookkeep vars/set/response of asynchronous algebraic evaluations. This is the queue of algebraic jobs populated by asynchronous map() that is later evaluated in synch() or synch_nowait().

- IntSet runningSet

  used by asynchronous_local_nowait to bookkeep which jobs are running

43.4.1 Detailed Description

Derived class within the interface class hierarchy for supporting interfaces to simulation codes. ApplicationInterface provides an interface class for performing parameter to response mappings using simulation code(s). It provides common functionality for a number of derived classes and contains the majority of all of the scheduling

algorithms in DAKOTA. The derived classes provide the specifics for managing code invocations using system calls, forks, direct procedure calls, or distributed resource facilities.

43.4.2 Member Function Documentation

43.4.2.1 void init_serial () [inline, protected, virtual]

DataInterface.C defaults of 0 servers are needed to distinguish an explicit user request for 1 server (serialization of a parallelism level) from no user request (use parallel auto-config). This default causes problems when init_communicators() is not called for an interface object (e.g., static scheduling fails in DirectApplicInterface::derived_map() for NestedModel::optionalInterface). This is the reason for this function: to reset certain defaults for interface objects that are used serially.

Reimplemented from Interface.

References ApplicationInterface::numAnalysisServers, and ApplicationInterface::numEvalServers.

43.4.2.2 void map (const Variables & *vars*, const ActiveSet & *set*, Response & *response*, const bool *asynch_flag* = false) [protected, virtual]

Provides a "mapping" of variables to responses using a simulation. Protected due to Interface letter-envelope idiom. The function evaluator for application interfaces. Called from derived_compute_response() and derived_-asynch_compute_response() in derived Model classes. If asynch_flag is not set, perform a blocking evaluation (using derived_map()). If asynch_flag is set, add the job to the beforeSynchCorePRPQueue queue for execution by one of the scheduler routines in synch() or synch_nowait(). Duplicate function evaluations are detected with duplication_detect().

Reimplemented from Interface.

References Response::active_set(), Interface::algebraic_mappings(), Interface::algebraicMappings, Interface::asv_mapping(), ApplicationInterface::asvControlFlag, ParallelLibrary::bcast_e(), ApplicationInterface::beforeSynchAlgPRPQueue, ApplicationInterface::beforeSynchCorePRPQueue, Response::copy(), Interface::coreMappings, Interface::currEvalId, Dakota::data_pairs, ApplicationInterface::defaultASV, ApplicationInterface::derived_map(), ApplicationInterface::duplication_detect(), ApplicationInterface::evalCacheFlag, Interface::evalIdCntr, Interface::fineGrainEvalCounters, Interface::fnGradCounter, Interface::fnHessCounter, Interface::fnLabels, Interface::fnValCounter, Response::function_labels(), Interface::init_algebraic_mappings(), Interface::interfaceId, ApplicationInterface::lenVarsActSetMessage, ApplicationInterface::manage_failure(), Interface::multiProcEvalFlag, Interface::newEvalIdCntr, Interface::newFnGradCounter, Interface::newFnHessCounter, Interface::newFnValCounter, Interface::outputLevel, ApplicationInterface::parallelLib, ActiveSet::request_vector(), Interface::response_mapping(), ApplicationInterface::restartFileFlag, and Dakota::write_restart.

43.4.2.3 const IntResponseMap & synch () [protected, virtual]

executes a blocking schedule for asynchronous evaluations in the beforeSynchCorePRPQueue and returns all jobs This function provides blocking synchronization for all cases of asynchronous evaluations, including the local asynchronous case (background system call, nonblocking fork, & multithreads), the message passing case, and the hybrid case. Called from derived_synchronize() in derived Model classes.

Reimplemented from Interface.

References Interface::algebraic_mappings(), Interface::algebraicMappings, ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::asynchLocalEvalStatic, ApplicationInterface::asynchronous_local_evaluations(), ApplicationInterface::asynchronous_local_evaluations_static(), ApplicationInterface::beforeSynchAlgPRPQueue, ApplicationInterface::beforeSynchCorePRPQueue, ApplicationInterface::beforeSynchDuplicateMap, Interface::coreMappings, ApplicationInterface::historyDuplicateMap, Interface::ieDedMasterFlag, ApplicationInterface::ieMessagePass, Interface::outputLevel, Interface::rawResponseMap, Interface::response_mapping(), ApplicationInterface::self_schedule_evaluations(), and ApplicationInterface::static_schedule_evaluations().

43.4.2.4 const IntResponseMap & synch_nowait () [protected, virtual]

executes a nonblocking schedule for asynchronous evaluations in the beforeSynchCorePRPQueue and returns a partial set of completed jobs This function will eventually provide nonblocking synchronization for all cases of asynchronous evaluations, however it currently supports only the local asynchronous case since nonblocking message passing schedulers have not yet been implemented. Called from derived_synchronize_nowait() in derived Model classes.

Reimplemented from Interface.

References Dakota::abort_handler(), Interface::algebraic_mappings(), Interface::algebraicMappings, ApplicationInterface::asynchronous_local_evaluations_nowait(), ApplicationInterface::beforeSynchAlgPRPQueue, ApplicationInterface::beforeSynchCorePRPQueue, ApplicationInterface::beforeSynchDuplicateMap, Interface::coreMappings, ParamResponsePair::eval_id(), ApplicationInterface::headerFlag, ApplicationInterface::historyDuplicateMap, ApplicationInterface::ieMessagePass, Dakota::lookup_by_eval_id(), Interface::outputLevel, ParamResponsePair::prp_response(), Interface::rawResponseMap, Interface::response_mapping(), and Response::update().

43.4.2.5 void serve_evaluations () [protected, virtual]

run on evaluation servers to serve the iterator master Invoked by the serve() function in derived Model classes. Passes control to serve_evaluations_asynch(), serve_evaluations_peer(), or serve_evaluations_synch() according to specified concurrency and self/static scheduler configuration.

Reimplemented from Interface.

References ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::evalServerId, Interface::ieDedMasterFlag, ApplicationInterface::serve_evaluations_asynch(), ApplicationInterface::serve_evaluations_peer(), and ApplicationInterface::serve_evaluations_synch().

43.4.2.6 void stop_evaluation_servers () [protected, virtual]

used by the iterator master to terminate evaluation servers This code is executed on the iteratorComm rank 0 processor when iteration on a particular model is complete. It sends a termination signal (tag = 0 instead of a valid fn_eval_id) to each of the slave analysis servers. NOTE: This function is called from the Strategy layer even when in serial mode. Therefore, use iteratorCommSize to provide appropriate fall through behavior.

Reimplemented from Interface.

References ParallelLibrary::bcast_e(), ParallelLibrary::free(), Interface::ieDedMasterFlag, ParallelLibrary::isend_ie(), ApplicationInterface::iteratorCommSize, Interface::multiProcEvalFlag, ApplicationInterface::numEvalServers, and ApplicationInterface::parallelLib.

43.4.2.7 void self_schedule_analyses () [protected]

blocking self-schedule of all analyses within a function evaluation using message passing This code is called from derived classes to provide the master portion of a master-slave algorithm for the dynamic self-scheduling of analyses among slave servers. It is patterned after self_schedule_evaluations(). It performs no analyses locally and matches either serve_analyses_synch() or serve_analyses_asynch() on the slave servers, depending on the value of asynchLocalAnalysisConcurrency. Self-scheduling approach assigns jobs in 2 passes. The 1st pass gives each server the same number of jobs (equal to asynchLocalAnalysisConcurrency). The 2nd pass assigns the remaining jobs to slave servers as previous jobs are completed. Single- and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within ParallelLibrary.

References ApplicationInterface::asynchLocalAnalysisConcurrency, ParallelLibrary::free(), ParallelLibrary::irecv_ea(), ParallelLibrary::isend_ea(), ApplicationInterface::numAnalysisDrivers, ApplicationInterface::numAnalysisServers, ApplicationInterface::parallelLib, ParallelLibrary::waitall(), and ParallelLibrary::waitsome().

Referenced by DirectApplicInterface::derived_map(), ForkApplicInterface::fork_application(), and SysCallApplicInterface::spawn_application().

43.4.2.8 void serve_analyses_synch () [protected]

serve the master analysis scheduler and manage one synchronous analysis job at a time This code is called from derived classes to run synchronous analyses on slave processors. The slaves receive requests (blocking receive), do local derived_map_ac's, and return codes. This is done continuously until a termination signal is received from the master. It is patterned after serve_evaluations_synch().

References ApplicationInterface::analysisCommRank, ParallelLibrary::bcast_a(), ApplicationInterface::derived_synchronous_local_analysis(), ParallelLibrary::isend_ea(), ApplicationInterface::multiProcAnalysisFlag, ApplicationInterface::parallelLib, ParallelLibrary::recv_ea(), and ParallelLibrary::wait().

Referenced by DirectApplicInterface::derived_map(), ForkApplicInterface::fork_application(), and SysCallApplicInterface::spawn_application().

43.4.2.9 bool duplication_detect (const Variables & *vars*, Response & *response*, const bool *asynch_flag*) [private]

checks data_pairs and beforeSynchCorePRPQueue to see if the current evaluation request has already been performed or queued Called from map() to check incoming evaluation request for duplication with content of data_pairs and beforeSynchCorePRPQueue. If duplication is detected, return true, else return false. Manage bookkeeping with historyDuplicateMap and beforeSynchDuplicateMap. Note that the list searches can get very expensive if a long list is searched on every new function evaluation (either from a large number of previous jobs, a large number of pending jobs, or both). For this reason, a user request for deactivation of the evaluation cache results in a complete bypass of duplication_detect(), even though a beforeSynchCorePRPQueue search would still be meaningful. Since the intent of this request is to streamline operations, both list searches are bypassed.

References Response::active_set(), ApplicationInterface::beforeSynchCorePRPQueue, ApplicationInterface::beforeSynchDuplicateMap, Response::copy(), Dakota::data_pairs, Interface::evalIdCntr, Dakota::hashedQueueEnd(), ApplicationInterface::historyDuplicateMap, Interface::interfaceId, Dakota::lookup_by_val(), and Response::update().

Referenced by ApplicationInterface::map().

43.4.2.10 void self_schedule_evaluations () [private]

blocking self-schedule of all evaluations in beforeSynchCorePRPQueue using message passing; executes on iteratorComm master This code is called from synch() to provide the master portion of a master-slave algorithm for the dynamic self-scheduling of evaluations among slave servers. It performs no evaluations locally and matches either serve_evaluations_synch() or serve_evaluations_asynch() on the slave servers, depending on the value of asynchLocalEvalConcurrency. Self-scheduling approach assigns jobs in 2 passes. The 1st pass gives each server the same number of jobs (equal to asynchLocalEvalConcurrency). The 2nd pass assigns the remaining jobs to slave servers as previous jobs are completed. Single- and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within ParallelLibrary.

References ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::beforeSynchCorePRPQueue, Dakota::data_pairs, ApplicationInterface::evalCacheFlag, ParallelLibrary::free(), ParallelLibrary::irecv_ie(), ParallelLibrary::isend_ie(), ApplicationInterface::lenResponseMessage, Dakota::lookup_by_eval_id(), ApplicationInterface::numEvalServers, Interface::outputLevel, ApplicationInterface::parallelLib, Interface::rawResponseMap, MPIUnpackBuffer::reset(), MPIPackBuffer::reset(), MPIUnpackBuffer::resize(), ApplicationInterface::restartFileFlag, ParallelLibrary::waitall(), ParallelLibrary::waitsome(), and Dakota::write_restart.

Referenced by ApplicationInterface::synch().

43.4.2.11 void static_schedule_evaluations () [private]

blocking static schedule of all evaluations in beforeSynchCorePRPQueue using message passing; executes on iteratorComm master This code runs on the iteratorCommRank 0 processor (the iterator) and is called from synch() in order to assign a static schedule. It matches serve_evaluations_peer() for any other processors within the first evaluation partition and serve_evaluations_{synch,asynch}() for all other evaluation partitions (depending on asynchLocalEvalConcurrency). It performs function evaluations locally for its portion of the static schedule using either asynchronous_local_evaluations() or synchronous_local_evaluations(). Single-level and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within ParallelLibrary. The iteratorCommRank 0 processor assigns the static schedule since it is the only processor with access to beforeSynchCorePRPQueue (it runs the iterator and calls synchronize). The alternate design of each peer selecting its own jobs using the modulus operator would be applicable if execution of this function (and therefore the job list) were distributed.

References ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::asynchLocalEvalStatic, ApplicationInterface::asynchronous_local_evaluations(), ApplicationInterface::beforeSynchCorePRPQueue, Dakota::data_pairs, ApplicationInterface::evalCacheFlag, ParallelLibrary::free(), ParallelLibrary::irecv_ie(), ParallelLibrary::isend_ie(), ApplicationInterface::lenResponseMessage, ApplicationInterface::numEvalServers, Interface::outputLevel, ApplicationInterface::parallelLib, Interface::rawResponseMap, MPIUnpackBuffer::resize(), ApplicationInterface::restartFileFlag, ApplicationInterface::synchronous_local_evaluations(), ParallelLibrary::waitall(), and Dakota::write_restart.

Referenced by ApplicationInterface::synch().

43.4.2.12 void asynchronous_local_evaluations (PRPQueue & *prp_queue*) [private]

perform all jobs in prp_queue using asynchronous approaches on the local processor This function provides blocking synchronization for the local asynch case (background system call, nonblocking fork, or threads). It can be called from synch() for a complete local scheduling of all asynchronous jobs or from static_schedule_evaluations() to perform a local portion of the total job set. It uses the derived_map_asynch() to initiate asynchronous evaluations and derived_synch() to capture completed jobs, and mirrors the self_schedule_evaluations() message passing scheduler as much as possible (derived_synch() is modeled after MPI_Waitsome()).

References ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::completionSet, Dakota::data_pairs, ApplicationInterface::derived_map_asynch(), ApplicationInterface::derived_synch(), ApplicationInterface::evalCacheFlag, Dakota::lookup_by_eval_id(), Interface::outputLevel, Interface::rawResponseMap, ApplicationInterface::restartFileFlag, and Dakota::write_restart.

Referenced by ApplicationInterface::static_schedule_evaluations(), and ApplicationInterface::synch().

43.4.2.13 void asynchronous_local_evaluations_static (PRPQueue & *prp_queue*) [private]

perform all the jobs in prp_queue using asynchronous approaches on the local processor, but schedule statically such that eval_id is always replaced with an equivalent one, modulo asynchLocalEvalConcurrency Locally statically-scheduled counterpart to asynchronous_local_evaluations. This scheduling policy specifically ensures that a completed asynchronous evaluation eval_id is replaced with an equivalent one, modulo asynchLocalEvalConcurrency. Designed to help with parallel tiling. A disadvantage of this scheduling policy is that is could leave local asynchronous worker "servers" idle in parsing the prp_queue, e.g., when restarting and some evals are already complete. In fact, anytime this function is called with non-contiguous eval_id's the full possible concurrency won't be leveraged.

This is currently only supported when DAKOTA is running in serial. Supporting in the MPI static / asynch local hybrid mode would require MPI static schedule that is either fully round-robin or fully block scheduled, not the present hybrid. It is not clear how to support this in the MPI self scheduled / asynch local hybrid mode.

If local evaluation concurrency is unlimited, this function is not needed.

References ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::completionSet, Dakota::data_pairs, ApplicationInterface::derived_map_asynch(), ApplicationInterface::derived_synch(), ApplicationInterface::evalCacheFlag, ApplicationInterface::localServerJobMap, Dakota::lookup_by_eval_id(), Interface::outputLevel, Interface::rawResponseMap, ApplicationInterface::restartFileFlag, ApplicationInterface::runningSet, and Dakota::write_restart.

Referenced by ApplicationInterface::synch().

43.4.2.14 void synchronous_local_evaluations (PRPQueue & *prp_queue*) [private]

perform all jobs in prp_queue using synchronous approaches on the local processor This function provides blocking synchronization for the local synchronous case (foreground system call, blocking fork, or procedure call from derived_map()). It is called from static_schedule_evaluations() to perform a local portion of the total job set.

References ParallelLibrary::bcast_e(), Interface::currEvalId, Dakota::data_pairs, ApplicationInterface::derived_map(), ApplicationInterface::evalCacheFlag, ApplicationInterface::lenVarsActSetMessage, ApplicationInterface::manage_failure(), Interface::multiProcEvalFlag, Interface::outputLevel, ApplicationInterface::parallelLib, Interface::rawResponseMap, ApplicationInterface::restartFileFlag, and Dakota::write_restart.

Referenced by ApplicationInterface::static_schedule_evaluations().

43.4.2.15 void asynchronous_local_evaluations_nowait (PRPQueue & *prp_queue*) [private]

launch new jobs in prp_queue asynchronously (if capacity is available), perform nonblocking query of all running jobs, and process any completed jobs (handles both local self- and local static-scheduling cases) This function provides nonblocking synchronization for the local asynch case (background system call, nonblocking fork, or threads). It is called from synch_nowait() and passed the complete set of all asynchronous jobs (beforeSynchCorePRPQueue). It uses derived_map_asynch() to initiate asynchronous evaluations and derived_synch_nowait() to capture completed jobs in nonblocking mode. It mirrors a nonblocking message passing scheduler as much as possible (derived_synch_nowait() modeled after MPI_Testsome()). The result of this function is rawResponseMap, which uses eval_id as a key. It is assumed that the incoming prp_queue contains only active and new jobs - i.e., all completed jobs are cleared by synch_nowait().

Also supports asynchronous local evaluations with static scheduling. This scheduling policy specifically ensures that a completed asynchronous evaluation eval_id is replaced with an equivalent one, modulo asynchLocalEvalConcurrency. In the nowait case, this could render some servers idle if evaluations don't come in eval_id order or some evaluations are cancelled by the caller in between calls. If this function is called with unlimited local eval concurrency, the static scheduling request is ignored.

References ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::asynchLocalEvalStatic, ApplicationInterface::completionSet, Dakota::data_pairs, ApplicationInterface::derived_map_asynch(), ApplicationInterface::derived_synch_nowait(), ApplicationInterface::evalCacheFlag, ApplicationInterface::localServerJobMap, Dakota::lookup_by_eval_id(), Interface::outputLevel, Interface::rawResponseMap, ApplicationInterface::restartFileFlag, ApplicationInterface::runningSet, and Dakota::write_restart.

Referenced by ApplicationInterface::synch_nowait().

43.4.2.16 void serve_evaluations_synch () [private]

serve the evaluation message passing schedulers and perform one synchronous evaluation at a time This code is invoked by serve_evaluations() to perform one synchronous job at a time on each slave/peer server. The servers receive requests (blocking receive), do local synchronous maps, and return results. This is done continuously until a termination signal is received from the master (sent via stop_evaluation_servers()).

References ParallelLibrary::bcast_e(), Interface::currEvalId, ApplicationInterface::derived_map(), ApplicationInterface::evalCommRank, ParallelLibrary::isend_ie(), ApplicationInterface::lenResponseMessage, ApplicationInterface::lenVarsActSetMessage, ApplicationInterface::manage_failure(), Interface::multiProcEvalFlag, ApplicationInterface::parallelLib, ParallelLibrary::recv_ie(), MPIPackBuffer::reset(), ParallelLibrary::wait(), and Variables::write_annotated().

Referenced by ApplicationInterface::serve_evaluations().

43.4.2.17 void serve_evaluations_asynch () [private]

serve the evaluation message passing schedulers and manage multiple asynchronous evaluations This code is invoked by serve_evaluations() to perform multiple asynchronous jobs on each slave/peer server. The servers test for any incoming jobs, launch any new jobs, process any completed jobs, and return any results. Each of these components is nonblocking, although the server loop continues until a termination signal is received from the master (sent via stop_evaluation_servers()). In the master-slave case, the master maintains the correct number of

jobs on each slave. In the static scheduling case, each server is responsible for limiting concurrency (since the entire static schedule is sent to the peers at start up).

References Dakota::abort_handler(), ApplicationInterface::asynchLocalEvalConcurrency, ApplicationInterface::completionSet, ApplicationInterface::derived_map_asynch(), ApplicationInterface::derived_synch_-nowait(), Interface::interfaceId, ParallelLibrary::irecv_ie(), ApplicationInterface::lenResponseMessage, ApplicationInterface::lenVarsActSetMessage, Dakota::lookup_by_eval_id(), ApplicationInterface::parallelLib, ParallelLibrary::recv_ie(), MPIUnpackBuffer::reset(), ParallelLibrary::send_ie(), and ParallelLibrary::test().

Referenced by ApplicationInterface::serve_evaluations().

43.4.2.18 void serve_evaluations_peer () [private]

serve the evaluation message passing schedulers and perform one synchronous evaluation at a time as part of the 1st peer This code is invoked by serve_evaluations() to perform a synchronous evaluation in coordination with the iteratorCommRank 0 processor (the iterator) for static schedules. The bcast() matches either the bcast() in synchronous_local_evaluations(), which is invoked by static_schedule_evaluations()), or the bcast() in map().

References ParallelLibrary::bcast_e(), Interface::currEvalId, ApplicationInterface::derived_map(), ApplicationInterface::lenVarsActSetMessage, ApplicationInterface::manage_failure(), ApplicationInterface::parallelLib, and Variables::write_annotated().

Referenced by ApplicationInterface::serve_evaluations().

The documentation for this class was generated from the following files:

- ApplicationInterface.H
- ApplicationInterface.C

43.5 Approximation Class Reference

Base class for the approximation class hierarchy. Inheritance diagram for Approximation::

Public Member Functions

- Approximation ()

  default constructor

- Approximation (ProblemDescDB &problem_db, size_t num_vars)

  standard constructor for envelope

- Approximation (const String &approx_type, const UShortArray &approx_order, size_t num_vars, short data_order)

  alternate constructor

- Approximation (const Approximation &approx)

  copy constructor

- virtual ~Approximation ()

  destructor

- Approximation operator= (const Approximation &approx)

  assignment operator

- virtual void build ()

  builds the approximation from scratch

- virtual void rebuild ()

  rebuilds the approximation incrementally

- virtual void pop (bool save_data)

  removes entries from end of SurrogateData::{vars,resp}Data (last points appended)

- virtual void restore ()

  restores state prior to previous append()

- virtual bool restore_available ()

  queries availability of restoration for trial set

- virtual size_t restoration_index ()

return index of trial set within restorable bookkeeping sets

- virtual void finalize ()

  finalize approximation by applying all remaining trial sets

- virtual size_t finalization_index (size_t i)

  return index of i-th trailing trial set within restorable bookkeeping sets

- virtual void store ()

  store current approximation for later combination

- virtual void combine (short corr_type)

  combine current approximation with previously stored approximation

- virtual Real get_value (const RealVector &x)

  retrieve the approximate function value for a given parameter vector

- virtual const RealVector & get_gradient (const RealVector &x)

  retrieve the approximate function gradient for a given parameter vector

- virtual const RealSymMatrix & get_hessian (const RealVector &x)

  retrieve the approximate function Hessian for a given parameter vector

- virtual Real get_prediction_variance (const RealVector &x)

  retrieve the variance of the predicted value for a given parameter vector

- virtual Real get_diagnostic (const String &metric_type)

  retrieve the diagnostic metric for the diagnostic type specified

- virtual const RealVector & approximation_coefficients () const

  return the coefficient array computed by build()/rebuild()

- virtual void approximation_coefficients (const RealVector &approx_coeffs)

  set the coefficient array from external sources, rather than computing with build()/rebuild()

- virtual void print_coefficients (std::ostream &s) const

  print the coefficient array computed in build()/rebuild()

- virtual int min_coefficients () const

  return the minimum number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- virtual int recommended_coefficients () const

  return the recommended number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- virtual int num_constraints () const

  return the number of constraints to be enforced via an anchor point

- virtual void clear_current ()

  clear current build data in preparation for next build

- virtual const bool diagnostics_available ()

  check if diagnostics are available for this approximation type

- int min_points (bool constraint_flag) const

  return the minimum number of points required to build the approximation type in numVars dimensions. Uses ∗_-coefficients() and num_constraints().

- int recommended_points (bool constraint_flag) const

  return the recommended number of samples to build the approximation type in numVars dimensions (default same as min_points)

- int num_variables () const

  return the number of variables used in the approximation

- const Pecos::SurrogateData & approximation_data () const

  return approxData

- void add (const Pecos::SurrogateDataVars &sdv, bool anchor_flag)

  append to SurrogateData::varsData or assign to SurrogateData::anchorVars

- void add (const Variables &vars, bool anchor_flag, bool deep_copy)

  extract the relevant RealVector from Variables and invoke add(RealVector&)

- void add (const Real ∗sample_c_vars, bool anchor_flag, bool deep_copy)

  create a RealVector view and invoke add(RealVector&)

- void add (const RealVector &sample_c_vars, bool anchor_flag, bool deep_copy)

  shared code among add(Variables&) and add(Real∗); adds a new data point by either appending to SurrogateData::varsData or assigning to SurrogateData::anchorVars, as dictated by anchor_flag. Uses add_point() and add_anchor().

- void add (const Pecos::SurrogateDataResp &sdr, bool anchor_flag)

  append to SurrogateData::respData or assign to SurrogateData::anchorResp

- void add (const Response &response, int fn_index, bool anchor_flag, bool deep_copy)

  adds a new data point by either appending to SurrogateData::respData or assigning to SurrogateData::anchorResp, as dictated by anchor_flag. Uses add_point() and add_anchor().

- void pop_count (size_t count)

  sets popCount (number of entries to pop from end of SurrogateData::{vars,resp}Data, based on size of last data set appended)

- size_t pop_count () const

  returns popCount (number of entries to pop from end of SurrogateData::{vars,resp}Data, based on size of last data set appended)

- void clear_all ()

  clear all build data (current and history) to restore original state

- void clear_anchor ()

  clear SurrogateData::anchor{Vars,Resp}

- void clear_data ()

  clear SurrogateData::{vars,resp}Data

- void set_bounds (const RealVector &lower, const RealVector &upper)

  set approximation lower and upper bounds (currently only used by graphics)

- Approximation * approx_rep () const

  returns approxRep for access to derived class member functions that are not mapped to the top Approximation level

Protected Member Functions

- Approximation (BaseConstructor, const ProblemDescDB &problem_db, size_t num_vars)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- Approximation (BaseConstructor, const String &approx_type, size_t num_vars, short data_order)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

Protected Attributes

- short outputLevel

  output verbosity level: {SILENT,QUIET,NORMAL,VERBOSE,DEBUG}_OUTPUT

- int numVars

  number of variables in the approximation

- String approxType

  approximation type identifier

- short buildDataOrder

  order of the data used for surrogate construction, in ActiveSet request vector 3-bit format.

- RealVector approxGradient

  gradient of the approximation returned by get_gradient()

- RealSymMatrix approxHessian

  Hessian of the approximation returned by get_hessian().

- Pecos::SurrogateData approxData

  contains the variables/response data for constructing a single approximation model (one response function)

- RealVector approxLowerBounds

  approximation lower bounds (used by 3D graphics and Surfpack KrigingModel)

- RealVector approxUpperBounds

  approximation upper bounds (used by 3D graphics and Surfpack KrigingModel)

Private Member Functions

- Approximation * get_approx (ProblemDescDB &problem_db, size_t num_vars)

  Used only by the standard envelope constructor to initialize approxRep to the appropriate derived type.

- Approximation * get_approx (const String &approx_type, const UShortArray &approx_order, size_t num_-vars, short data_order)

  Used only by the alternate envelope constructor to initialize approxRep to the appropriate derived type.

Private Attributes

- size_t popCount

  number of points previously added by append() to be removed by pop()

- Approximation * approxRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing approxRep

43.5.1 Detailed Description

Base class for the approximation class hierarchy. The Approximation class is the base class for the response data fit approximation class hierarchy in DAKOTA. One instance of an Approximation must be created for each function to be approximated (a vector of Approximations is contained in ApproximationInterface). For memory efficiency and enhanced polymorphism, the approximation hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Approximation) serves as the envelope and one of the derived classes (selected in Approximation::get_approx()) serves as the letter.

43.5.2 Constructor & Destructor Documentation

43.5.2.1 Approximation ()

default constructor The default constructor is used in Array<Approximation> instantiations and by the alternate envelope constructor. approxRep is NULL in this case (problem_db is needed to build a meaningful Approximation object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

43.5.2.2 Approximation (ProblemDescDB & *problem_db*, size_t *num_vars*)

standard constructor for envelope Envelope constructor only needs to extract enough data to properly execute get_approx, since Approximation(BaseConstructor, problem_db) builds the actual base class data for the derived approximations.

References Dakota::abort_handler(), Approximation::approxRep, and Approximation::get_approx().

43.5.2.3 Approximation (const String & *approx_type*, const UShortArray & *approx_order*, size_t *num_vars*, short *data_order*)

alternate constructor This is the alternate envelope constructor for instantiations on the fly. Since it does not have access to problem_db, the letter class is not fully populated. This constructor executes get_approx(type), which invokes the default constructor of the derived letter class, which in turn invokes the default constructor of the base class.

References Dakota::abort_handler(), Approximation::approxRep, and Approximation::get_approx().

43.5.2.4 Approximation (const Approximation & *approx*)

copy constructor Copy constructor manages sharing of approxRep and incrementing of referenceCount.

References Approximation::approxRep, and Approximation::referenceCount.

43.5.2.5 ∼Approximation () `[virtual]`

destructor Destructor decrements referenceCount and only deletes approxRep when referenceCount reaches zero.

References Approximation::approxRep, and Approximation::referenceCount.

43.5.2.6 Approximation (BaseConstructor, const ProblemDescDB & *problem_db*, size_t *num_vars*) `[protected]`

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_approx() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling get_approx() again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ∼Approximation).

approxType.ends("_interpolation_polynomial") && approxType.ends("_orthogonal_polynomial"))

References Approximation::approxType, and ProblemDescDB::get_bool().

43.5.2.7 Approximation (BaseConstructor, const String & *approx_type*, size_t *num_vars*, short *data_order*) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_approx() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling get_approx() again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ∼Approximation).

43.5.3 Member Function Documentation

43.5.3.1 Approximation operator= (const Approximation & *approx*)

assignment operator Assignment operator decrements referenceCount for old approxRep, assigns new approxRep, and increments referenceCount for new approxRep.

References Approximation::approxRep, and Approximation::referenceCount.

43.5.3.2 void build () [virtual]

builds the approximation from scratch This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented in GaussProcApproximation, PecosApproximation, SurfpackApproximation, TANA3Approximation, and TaylorApproximation.

References Dakota::abort_handler(), Approximation::approxData, Approximation::approxRep, Approximation::build(), Approximation::min_points(), and Approximation::numVars.

Referenced by Approximation::build(), and Approximation::rebuild().

43.5.3.3 void rebuild () [virtual]

rebuilds the approximation incrementally This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented in PecosApproximation.

References Approximation::approxRep, Approximation::build(), and Approximation::rebuild().

Referenced by Approximation::rebuild().

43.5.3.4 void pop (bool *save_data*) [virtual]

removes entries from end of SurrogateData::{vars,resp}Data (last points appended) This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented in PecosApproximation.

References Approximation::approxData, Approximation::approxRep, Approximation::pop(), and Approximation::popCount.

Referenced by Approximation::pop().

43.5.3.5 void restore () [virtual]

restores state prior to previous append() This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented in PecosApproximation.

References Approximation::approxData, Approximation::approxRep, Approximation::popCount, Approximation::restoration_index(), and Approximation::restore().

Referenced by Approximation::restore().

43.5.3.6 void finalize () [virtual]

finalize approximation by applying all remaining trial sets This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented in PecosApproximation.

References Approximation::approxData, Approximation::approxRep, Approximation::finalization_index(), and Approximation::finalize().

Referenced by Approximation::finalize().

43.5.3.7 void clear_current () [inline, virtual]

clear current build data in preparation for next build Redefined by TANA3Approximation to clear current data but preserve history.

Reimplemented in TANA3Approximation.

References Approximation::approxRep, Approximation::clear_all(), and Approximation::clear_current().

Referenced by Approximation::clear_current().

43.5.3.8 void clear_all () [inline]

clear all build data (current and history) to restore original state Clears out any history (e.g., TANA3Approximation use for a different response function in NonDReliability).

References Approximation::approxData, Approximation::approxRep, and Approximation::clear_all().

Referenced by Approximation::clear_all(), and Approximation::clear_current().

43.5.3.9 Approximation * get_approx (ProblemDescDB & *problem_db*, size_t *num_vars*) [private]

Used only by the standard envelope constructor to initialize approxRep to the appropriate derived type. Used only by the envelope constructor to initialize approxRep to the appropriate derived type.

References String::ends(), and ProblemDescDB::get_string().

Referenced by Approximation::Approximation().

43.5.3.10 Approximation * get_approx (const String & *approx_type*, const UShortArray & *approx_order*, size_t *num_vars*, short *data_order*) [private]

Used only by the alternate envelope constructor to initialize approxRep to the appropriate derived type. Used only by the envelope constructor to initialize approxRep to the appropriate derived type.

References String::ends().

43.5.4 Member Data Documentation

43.5.4.1 short buildDataOrder [protected]

order of the data used for surrogate construction, in ActiveSet request vector 3-bit format. This setting distinguishes derivative data intended for use in construction (includes derivatives w.r.t. the build variables) from derivative data that may be approximated separately (excludes derivatives w.r.t. auxilliary variables). This setting should also not be inferred directly from the responses specification, since we may need gradient support for evaluating gradients at a single point (e.g., the center of a trust region), but not require gradient evaluations at every point.

Referenced by SurfpackApproximation::add_sdp_to_surfdata(), TaylorApproximation::build(), TANA3Approximation::build(), GaussProcApproximation::GaussProcApproximation(), TaylorApproximation::get_gradient(), TaylorApproximation::get_hessian(), TaylorApproximation::get_value(), TaylorApproximation::min_coefficients(), Approximation::min_points(), PecosApproximation::PecosApproximation(), Approximation::recommended_points(), SurfpackApproximation::SurfpackApproximation(), SurfpackApproximation::surrogates_to_surf_data(), TANA3Approximation::TANA3Approximation(), and TaylorApproximation::TaylorApproximation().

The documentation for this class was generated from the following files:

- DakotaApproximation.H
- DakotaApproximation.C

43.6 ApproximationInterface Class Reference

Derived class within the interface class hierarchy for supporting approximations to simulation-based results. Inheritance diagram for ApproximationInterface::

Public Member Functions

- ApproximationInterface (ProblemDescDB &problem_db, const Variables &am_vars, bool am_cache, const String &am_interface_id, size_t num_fns)

  primary constructor

- ApproximationInterface (const String &approx_type, const UShortArray &approx_order, const Variables &am_vars, bool am_cache, const String &am_interface_id, size_t num_fns, short data_order, short output_level)

  alternate constructor for instantiations on the fly

- ~ApproximationInterface ()

  destructor

Protected Member Functions

- void map (const Variables &vars, const ActiveSet &set, Response &response, const bool asynch_flag=false)

  the function evaluator: provides an approximate "mapping" from the variables to the responses using functionSurfaces

- int minimum_points (bool constraint_flag) const

  returns the minimum number of samples required to build the functionSurfaces

- int recommended_points (bool constraint_flag) const

  returns the recommended number of samples recommended to build the functionSurfaces

- void approximation_function_indices (const IntSet &approx_fn_indices)

  set the (currently active) approximation function index set

- void update_approximation (const Variables &vars, const IntResponsePair &response_pr)
- void update_approximation (const RealMatrix &samples, const IntResponseMap &resp_map)
- void update_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map)

- void append_approximation (const Variables &vars, const IntResponsePair &response_pr)
- void append_approximation (const RealMatrix &samples, const IntResponseMap &resp_map)
- void append_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map)
- void build_approximation (const RealVector &lower_bnds, const RealVector &upper_bnds)
- void rebuild_approximation (const BoolDeque &rebuild_deque)
- void pop_approximation (bool save_surr_data)
- void restore_approximation ()
- bool restore_available ()
- void finalize_approximation ()

  finalizes the approximation by applying all trial increments

- void store_approximation ()

  move the current approximation into storage for later combination

- void combine_approximation (short corr_type)

  combine the current approximation with one previously stored

- void clear_current ()

  clears current data from an approximation interface

- void clear_all ()

  clears all data from an approximation interface

- std::vector< Approximation > & approximations ()

  retrieve the Approximations within an ApproximationInterface

- const Pecos::SurrogateData & approximation_data (size_t index)

  retrieve the approximation data from a particular Approximation within an ApproximationInterface

- const RealVectorArray & approximation_coefficients ()

  retrieve the approximation coefficients from each Approximation within an ApproximationInterface

- void approximation_coefficients (const RealVectorArray &approx_coeffs)

  set the approximation coefficients within each Approximation within an ApproximationInterface

- const RealVector & approximation_variances (const RealVector &c_vars)

  retrieve the approximation variances from each Approximation within an ApproximationInterface

- const IntResponseMap & synch ()

  recovers data from a series of asynchronous evaluations (blocking)

- const IntResponseMap & synch_nowait ()

  recovers data from a series of asynchronous evaluations (nonblocking)

Private Member Functions

- void mixed_add (const Variables &vars, const Response &response, bool anchor)

  add variables/response data to functionSurfaces using a mixture of shallow and deep copies

- void mixed_add (const Real ∗c_vars, const Response &response, bool anchor)

  add variables/response data to functionSurfaces using a mixture of shallow and deep copies

- void shallow_add (const Variables &vars, const Response &response, bool anchor)

  add variables/response data to functionSurfaces using a shallow copy

- void update_pop_counts (const IntResponseMap &resp_map)

  update the popCount within each of the functionSurfaces based on the active set definitions within resp_map

Private Attributes

- IntSet approxFnIndices

  for incomplete approximation sets, this array specifies the response function subset that is approximated

- std::vector< Approximation > functionSurfaces

  list of approximations, one per response function

- RealVectorArray functionSurfaceCoeffs

  array of approximation coefficient vectors, one vector per response function

- RealVector functionSurfaceVariances

  vector of approximation variances, one value per response function

- StringArray diagnosticSet

  set of diagnostic metrics

- Variables actualModelVars

  copy of the actualModel variables object used to simplify conversion among differing variable views

- bool actualModelCache

  indicates usage of an evaluation cache by the actualModel

- String actualModelInterfaceId

  the interface id from the actualModel used for ordered PRPCache lookups

- IntResponseMap beforeSynchResponseMap

  bookkeeping map to catalogue responses generated in map() for use in synch() and synch_nowait(). This supports pseudo-asynchronous operations (approximate responses are always computed synchronously, but asynchronous virtual functions are supported through bookkeeping).

43.6.1 Detailed Description

Derived class within the interface class hierarchy for supporting approximations to simulation-based results. ApproximationInterface provides an interface class for building a set of global/local/multipoint approximations and performing approximate function evaluations using them. It contains a list of Approximation objects, one for each response function.

43.6.2 Member Function Documentation

43.6.2.1 void update_approximation (const Variables & *vars*, const IntResponsePair & *response_pr*) [protected, virtual]

This function populates/replaces each Approximation::anchorPoint with the incoming variables/response data point.

Reimplemented from Interface.

References ApproximationInterface::actualModelCache, ApproximationInterface::actualModelInterfaceId, Dakota::data_pairs, Dakota::lookup_by_ids(), ApproximationInterface::mixed_add(), and ApproximationInterface::shallow_add().

43.6.2.2 void update_approximation (const RealMatrix & *samples*, const IntResponseMap & *resp_map*) [protected, virtual]

This function populates/replaces each Approximation::currentPoints with the incoming variables/response arrays.

Reimplemented from Interface.

References Dakota::abort_handler(), ApproximationInterface::actualModelCache, ApproximationInterface::actualModelInterfaceId, ApproximationInterface::approxFnIndices, Dakota::data_pairs, ApproximationInterface::functionSurfaces, Dakota::lookup_by_ids(), ApproximationInterface::mixed_add(), and ApproximationInterface::shallow_add().

43.6.2.3 void update_approximation (const VariablesArray & *vars_array*, const IntResponseMap & *resp_map*) [protected, virtual]

This function populates/replaces each Approximation::currentPoints with the incoming variables/response arrays.

Reimplemented from Interface.

References Dakota::abort_handler(), ApproximationInterface::actualModelCache, ApproximationInterface::actualModelInterfaceId, ApproximationInterface::approxFnIndices, Dakota::data_pairs, ApproximationInterface::functionSurfaces, Dakota::lookup_by_ids(), ApproximationInterface::mixed_add(), and ApproximationInterface::shallow_add().

43.6.2.4 void append_approximation (const Variables & *vars*, const IntResponsePair & *response_pr*) [protected, virtual]

This function appends to each Approximation::currentPoints with one incoming variables/response data point.

Reimplemented from Interface.

References ApproximationInterface::actualModelCache, ApproximationInterface::actualModelInterfaceId, ApproximationInterface::approxFnIndices, Dakota::data_pairs, ApproximationInterface::functionSurfaces, Dakota::lookup_by_ids(), ApproximationInterface::mixed_add(), and ApproximationInterface::shallow_add().

43.6.2.5 void append_approximation (const RealMatrix & *samples*, const IntResponseMap & *resp_map*) [protected, virtual]

This function appends to each Approximation::currentPoints with multiple incoming variables/response data points.

Reimplemented from Interface.

References Dakota::abort_handler(), ApproximationInterface::actualModelCache, ApproximationInterface::actualModelInterfaceId, Dakota::data_pairs, Dakota::lookup_by_ids(), ApproximationInterface::mixed_add(), ApproximationInterface::shallow_add(), and ApproximationInterface::update_pop_counts().

43.6.2.6 void append_approximation (const VariablesArray & *vars_array*, const IntResponseMap & *resp_map*) [protected, virtual]

This function appends to each Approximation::currentPoints with multiple incoming variables/response data points.

Reimplemented from Interface.

References Dakota::abort_handler(), ApproximationInterface::actualModelCache, ApproximationInterface::actualModelInterfaceId, Dakota::data_pairs, Dakota::lookup_by_ids(), ApproximationInterface::mixed_add(), ApproximationInterface::shallow_add(), and ApproximationInterface::update_pop_counts().

43.6.2.7 void build_approximation (const RealVector & *lower_bnds*, const RealVector & *upper_bnds*) [protected, virtual]

This function finds the coefficients for each Approximation based on the data passed through update_approximation() calls. The bounds are used only for graphics visualization.

Reimplemented from Interface.

References ApproximationInterface::approxFnIndices, ApproximationInterface::diagnosticSet, ApproximationInterface::functionSurfaces, and Interface::outputLevel.

43.6.2.8 void rebuild_approximation (const BoolDeque & *rebuild_deque*) [protected, virtual]

This function updates the coefficients for each Approximation based on data increments provided by {update,append}_approximation().

Reimplemented from Interface.

References ApproximationInterface::approxFnIndices, and ApproximationInterface::functionSurfaces.

43.6.2.9 void pop_approximation (bool *save_surr_data*) [protected, virtual]

This function removes data provided by a previous call to append_approximation().

Reimplemented from Interface.

References ApproximationInterface::approxFnIndices, and ApproximationInterface::functionSurfaces.

43.6.2.10 void restore_approximation () [protected, virtual]

This function updates the coefficients for each Approximation based on data increments provided by {update,append}_approximation().

Reimplemented from Interface.

References ApproximationInterface::approxFnIndices, and ApproximationInterface::functionSurfaces.

43.6.2.11 bool restore_available () [protected, virtual]

This function updates the coefficients for each Approximation based on data increments provided by {update,append}_approximation().

Reimplemented from Interface.

References ApproximationInterface::approxFnIndices, and ApproximationInterface::functionSurfaces.

43.6.3 Member Data Documentation

43.6.3.1 std::vector<Approximation> functionSurfaces [private]

list of approximations, one per response function This formulation allows the use of mixed approximations (i.e., different approximations used for different response functions), although the input specification is not currently general enough to support it.

Referenced by ApproximationInterface::append_approximation(), ApproximationInterface::approximation_coefficients(), ApproximationInterface::approximation_data(), ApproximationInterface::approximation_variances(), ApproximationInterface::ApproximationInterface(), ApproximationInterface::approximations(), ApproximationInterface::build_approximation(), ApproximationInterface::clear_all(), ApproximationInterface::clear_current(), ApproximationInterface::combine_approximation(), ApproximationInterface::finalize_approximation(), ApproximationInterface::map(), ApproximationInterface::minimum_points(), ApproximationInterface::mixed_add(), ApproximationInterface::pop_approximation(), ApproximationInterface::rebuild_approximation(), ApproximationInterface::recommended_points(), ApproximationInterface::restore_approximation(), ApproximationInterface::restore_available(), ApproximationInterface::shallow_add(), ApproximationInterface::store_approximation(), ApproximationInterface::update_approximation(), and ApproximationInterface::update_pop_counts().

The documentation for this class was generated from the following files:

- ApproximationInterface.H
- ApproximationInterface.C

43.7 APPSEvalMgr Class Reference

Evaluation manager class for APPSPACK.

Public Member Functions

- APPSEvalMgr (Model &model)

  constructor

- ~APPSEvalMgr ()

  destructor

- bool isReadyForWork () const

  tells APPS whether or not there is a processor available to perform a function evaluation

- bool submit (const int apps_tag, const HOPSPACK::Vector &apps_xtrial, HOPSPACK::EvalRequestType apps_request)

  performs a function evaluation at APPS-provided x_in

- int recv (int &apps_tag, HOPSPACK::Vector &apps_f, HOPSPACK::Vector &apps_cEqs, HOPSPACK::Vector &apps_cIneqs, string &apps_msg)

  returns a function value to APPS

- void printDebugInfo (void) const

  empty implementation of debug info needed to complete the interface

- void printTimingInfo (void) const

  empty implementation of timing info needed to complete the interface

- void set_asynch_flag (const bool dakotaAsynchFlag)

  publishes whether or not to do asynchronous evaluations

- void set_blocking_synch (const bool blockingSynchFlag)

  publishes whether or not APPS is operating synchronously

- void set_total_workers (const int numDakotaWorkers)

  publishes the number of processors available for function evaluations

- void set_constraint_map (std::vector< int > constraintMapIndices, std::vector< double > constraintMapMultipliers, std::vector< double > constraintMapOffsets)

  publishes constraint transformation

Private Attributes

- Model & iteratedModel

  reference to the APPSOptimizer's model passed in the constructor

- bool modelAsynchFlag

  flag for asynchronous function evaluations

- bool blockingSynch

  flag for APPS synchronous behavior

- int numWorkersUsed

  number of processors actively performing function evaluations

- int numWorkersTotal

  total number of processors available for performing function evaluations

- std::vector< int > constrMapIndices

  map from Dakota constraint number to APPS constraint number

- std::vector< double > constrMapMultipliers

  multipliers for constraint transformations

- std::vector< double > constrMapOffsets

  offsets for constraint transformations

- RealVector xTrial

  trial iterate

- std::map< int, int > tagList

  map of DAKOTA eval id to APPS eval id (for asynchronous evaluations)

- std::map< int, RealVector > functionList

  map of APPS eval id to responses (for synchronous evaluations)

- IntResponseMap dakotaResponseMap

  map of DAKOTA responses returned by synchronize_nowait()

43.7.1 Detailed Description

Evaluation manager class for APPSPACK. The APPSEvalMgr class is derived from APPSPACK's Executor class. It implements the methods of that class in such away that allows DAKOTA to manage the computation of responses instead of APPS. Iterate and response values are passed between Dakota and APPSPACK via this interface.

43.7.2 Constructor & Destructor Documentation

43.7.2.1 APPSEvalMgr (Model & *model*)

constructor Evaluation manager class for APPSPACK.

The APPSEvalMgr class is derived from APPSPACK's Executor class. It implements the methods of that class in such away that allows DAKOTA to manage the computation of responses instead of APPS. Iterate and response values are passed between Dakota and APPSPACK via this interface.

43.7.3 Member Function Documentation

43.7.3.1 bool isReadyForWork () const

tells APPS whether or not there is a processor available to perform a function evaluation Check to see if all processors available for function evaluations are being used. If not, tell APPS that one is available.

References APPSEvalMgr::numWorkersTotal, and APPSEvalMgr::numWorkersUsed.

43.7.3.2 bool submit (const int *apps_tag*, const HOPSPACK::Vector & *apps_xtrial*, HOPSPACK::EvalRequestType *apps_request*)

performs a function evaluation at APPS-provided x_in Convert APPSPACK vector of variables to DAKOTA vector of variables and perform function evaluation asynchronously or not as specified in the DAKOTA input deck. If evaluation is asynchronous, map the dakota id to the APPS tag. If evaluation is synchronous, map the responses to the APPS tag.

References Model::asynch_compute_response(), Model::compute_response(), Model::continuous_variables(), Model::current_response(), Model::evaluation_id(), Response::function_values(), APPSEvalMgr::functionList, APPSEvalMgr::iteratedModel, APPSEvalMgr::modelAsynchFlag, APPSEvalMgr::numWorkersTotal, APPSEvalMgr::numWorkersUsed, APPSEvalMgr::tagList, and APPSEvalMgr::xTrial.

43.7.3.3 int recv (int & *apps_tag*, HOPSPACK::Vector & *apps_f*, HOPSPACK::Vector & *apps_cEqs*, HOPSPACK::Vector & *apps_cIneqs*, string & *apps_msg*)

returns a function value to APPS Retrieve a set of reponse values, convert to APPS data structures, and return them to APPS. APPS tags are tied to corresponding responses using the appropriate (i.e., asynchronous or synchronous) map.

References APPSEvalMgr::blockingSynch, APPSEvalMgr::constrMapIndices, APPSEvalMgr::constrMapMultipliers, APPSEvalMgr::constrMapOffsets, APPSEvalMgr::dakotaResponseMap, APPSEvalMgr::functionList, APPSEvalMgr::iteratedModel, APPSEvalMgr::modelAsynchFlag, Model::num_nonlinear_eq_constraints(), APPSEvalMgr::numWorkersUsed, Model::synchronize(), Model::synchronize_nowait(), and APPSEvalMgr::tagList.

The documentation for this class was generated from the following files:

- APPSEvalMgr.H
- APPSEvalMgr.C

43.8 APPSOptimizer Class Reference

Wrapper class for APPSPACK. Inheritance diagram for APPSOptimizer::

Public Member Functions

- APPSOptimizer (Model &model)

  constructor

- APPSOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor for on-the-fly instantiations

- ∼APPSOptimizer ()

  destructor

- void find_optimum ()

  Performs the iterations to determine the optimal solution.

Protected Member Functions

- void set_apps_parameters ()

  sets options for specific methods based on user specifications

- void initialize_variables_and_constraints ()

  initializes problem variables and constraints

Protected Attributes

- HOPSPACK::ParameterList params

  Pointer to APPS parameter list.

- HOPSPACK::ParameterList ∗ problemParams

Pointer to APPS problem parameter sublist.

- HOPSPACK::ParameterList * linearParams

  Pointer to APPS linear constraint parameter sublist.

- HOPSPACK::ParameterList * mediatorParams

  Pointer to APPS mediator parameter sublist.

- HOPSPACK::ParameterList * citizenParams

  Pointer to APPS citizen/algorithm parameter sublist.

- APPSEvalMgr * evalMgr

  Pointer to the APPS evaluation manager object.

- std::vector< int > constraintMapIndices

  map from Dakota constraint number to APPS constraint number

- std::vector< double > constraintMapMultipliers

  multipliers for constraint transformations

- std::vector< double > constraintMapOffsets

  offsets for constraint transformations

43.8.1 Detailed Description

Wrapper class for APPSPACK. The APPSOptimizer class provides a wrapper for APPSPACK, a Sandia-developed C++ library for generalized pattern search. APPSPACK defaults to a coordinate pattern search but also allows for augmented search patterns. It can solve problems with bounds, linear constraints, and general nonlinear constraints. APPSOptimizer uses an APPSEvalMgr object to manage the function evaluations.

The user input mappings are as follows: `output` `max_function_evaluations`, `constraint_tol` `initial_delta`, `contraction_factor`, `threshold_delta`, `solution_target`, `synchronization`, `merit_function`, `constraint_penalty`, and `smoothing_factor` are mapped into APPS's `"Debug"`, "Maximum Evaluations", "Bounds Tolerance"/"Machine Epsilon"/"Constraint Tolerance", "Initial Step", "Contraction Factor", "Step Tolerance", "Function Tolerance", "Synchronous", "Method", "Initial Penalty Value", and "Initial Smoothing Value" data attributes. Refer to the APPS web site (`http://software.sandia.gov/appspack`) for additional information on APPS objects and controls.

43.8.2 Constructor & Destructor Documentation

43.8.2.1 APPSOptimizer (Model & *model*)

constructor Wrapper class for HOPSPACK.

The APPSOptimizer class provides a wrapper for HOPSPACK, a Sandia-developed C++ library for generalized pattern search. HOPSPACK defaults to a coordinate pattern search but also allows for augmented search patterns.

It can solve problems with bounds, linear constraints, and general nonlinear constraints. APPSOptimizer uses an APPSEvalMgr object to manage the function evaluations.

The user input mappings are as follows: `output` `max_function_evaluations`, `constraint_tol` `initial_delta`, `contraction_factor`, `threshold_delta`, `solution_target`, `synchronization`, `merit_function`, `constraint_penalty`, and `smoothing_factor` are mapped into HOPS's `"Display"`, "Maximum Evaluations", "Active Tolerance"/"Nonlinear Active Tolerance", "Initial Step", "Contraction Factor", "Step Tolerance", "Objective Target", "Synchronous Evaluations", "Penalty Function", "Penalty Parameter", and "Penalty Smoothing Value" data attributes. Refer to the HOPS web site (`https://software.sandia.gov/trac/hopspack`) for additional information on HOPS objects and controls.

References APPSOptimizer::evalMgr, Model::init_communicators(), Iterator::iteratedModel, Optimizer::localObjectiveRecast, Iterator::maxConcurrency, Minimizer::scaleFlag, and APPSOptimizer::set_apps_parameters().

43.8.3 Member Function Documentation

43.8.3.1 void find_optimum () [virtual]

Performs the iterations to determine the optimal solution. find_optimum redefines the Optimizer virtual function to perform the optimization using HOPS. It first sets up the problem data, then executes minimize() on the HOPS optimizer, and finally catalogues the results.

Implements Optimizer.

References Model::asynch_flag(), Iterator::bestResponseArray, Iterator::bestVariablesArray, APPSOptimizer::constraintMapIndices, APPSOptimizer::constraintMapMultipliers, APPSOptimizer::constraintMapOffsets, APPSOptimizer::evalMgr, Model::evaluation_capacity(), APPSOptimizer::initialize_variables_and_constraints(), Iterator::iteratedModel, Optimizer::localObjectiveRecast, Iterator::numContinuousVars, Iterator::numFunctions, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, APPSOptimizer::params, APPSEvalMgr::set_asynch_flag(), and APPSEvalMgr::set_total_workers().

43.8.3.2 void set_apps_parameters () [protected]

sets options for specific methods based on user specifications Set all of the HOPS algorithmic parameters as specified in the DAKOTA input deck. This is called at construction time.

References APPSOptimizer::citizenParams, Minimizer::constraintTol, APPSOptimizer::evalMgr, ProblemDescDB::get_real(), ProblemDescDB::get_string(), ProblemDescDB::is_null(), APPSOptimizer::linearParams, Iterator::maxConcurrency, Iterator::maxFunctionEvals, APPSOptimizer::mediatorParams, Iterator::numContinuousVars, Minimizer::numNonlinearConstraints, Iterator::outputLevel, APPSOptimizer::params, Iterator::probDescDB, APPSOptimizer::problemParams, and APPSEvalMgr::set_blocking_synch().

Referenced by APPSOptimizer::APPSOptimizer().

43.8.3.3 void initialize_variables_and_constraints () [protected]

initializes problem variables and constraints Set the variables and constraints as specified in the DAKOTA input deck. This is done at run time.

References Minimizer::bigRealBoundSize, APPSOptimizer::constraintMapIndices, APPSOptimizer::constraintMapMultipliers, APPSOptimizer::constraintMapOffsets, Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::continuous_variables(), APPSOptimizer::evalMgr, Iterator::iteratedModel, Model::linear_eq_constraint_coeffs(), Model::linear_eq_constraint_targets(), Model::linear_ineq_constraint_coeffs(), Model::linear_ineq_constraint_lower_bounds(), Model::linear_ineq_constraint_upper_bounds(), APPSOptimizer::linearParams, Model::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Iterator::numContinuousVars, Minimizer::numLinearEqConstraints, Minimizer::numLinearIneqConstraints, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, APPSOptimizer::problemParams, and APPSEvalMgr::set_constraint_map().

Referenced by APPSOptimizer::find_optimum().

The documentation for this class was generated from the following files:

- APPSOptimizer.H
- APPSOptimizer.C

43.9 BaseConstructor Struct Reference

Dummy struct for overloading letter-envelope constructors.

Public Member Functions

- BaseConstructor (int=0)

  C++ structs can have constructors.

43.9.1 Detailed Description

Dummy struct for overloading letter-envelope constructors. BaseConstructor is used to overload the constructor for the base class portion of letter objects. It avoids infinite recursion (Coplien p.139) in the letter-envelope idiom by preventing the letter from instantiating another envelope. Putting this struct here avoids circular dependencies.

The documentation for this struct was generated from the following file:

- global_defs.h

43.10 BiStream Class Reference

The binary input stream class. Overloads the >> operator for all data types.

Public Member Functions

- BiStream ()

  Default constructor, need to open.

- BiStream (const char *s)

  Constructor takes name of input file.

- BiStream (const char *s, std::ios_base::openmode mode)

  Constructor takes name of input file, mode.

- ~BiStream ()

  Destructor, calls xdr_destroy to delete xdr stream.

- BiStream & operator>> (std::string &ds)

  Binary Input stream operator>>.

- BiStream & operator>> (char *s)

  Input operator, reads char from binary stream BiStream.*

- BiStream & operator>> (char &c)

  Input operator, reads char from binary stream BiStream.

- BiStream & operator>> (int &i)

  Input operator, reads int from binary stream BiStream.*

- BiStream & operator>> (long &l)

  Input operator, reads long from binary stream BiStream.

- BiStream & operator>> (short &s)

  Input operator, reads short from binary stream BiStream.

- BiStream & operator>> (bool &b)

  Input operator, reads bool from binary stream BiStream.

- BiStream & operator>> (double &d)

  Input operator, reads double from binary stream BiStream.

- BiStream & operator>> (float &f)

  Input operator, reads float from binary stream BiStream.

- BiStream & operator>> (unsigned char &c)

  Input operator, reads unsigned char∗ from binary stream BiStream.

- BiStream & operator>> (unsigned int &i)

  Input operator, reads unsigned int from binary stream BiStream.

- BiStream & operator>> (unsigned long &l)

  Input operator, reads unsigned long from binary stream BiStream.

- BiStream & operator>> (unsigned short &s)

  Input operator, reads unsigned short from binary stream BiStream.

Private Attributes

- XDR xdrInBuf

  XDR input stream buffer.

- char inBuf [MAX_NETOBJ_SZ]

  Buffer to hold data as it is read in.

43.10.1 Detailed Description

The binary input stream class. Overloads the >> operator for all data types. The Dakota::BiStream class is a binary input class which overloads the >> operator for all standard data types (int, char, float, etc). The class relies on the methods within the ifstream base class. The Dakota::BiStream class inherits from the ifstream class. If available, the class utilize rpc/xdr to construct machine independent binary files. These Dakota restart files can be moved from host to host. The motivation to develop these classes was to replace the Rogue wave classes which Dakota historically used for binary I/O.

43.10.2 Constructor & Destructor Documentation

43.10.2.1 BiStream ()

Default constructor, need to open. Default constructor, allocates xdr stream , but does not call the open method. The open method must be called before stream can be read.

References BiStream::inBuf, and BiStream::xdrInBuf.

43.10.2.2 BiStream (const char ∗ *s*)

Constructor takes name of input file. Constructor which takes a char∗ filename. Calls the base class open method with the filename and no other arguments. Also allocates the xdr stream.

References BiStream::inBuf, and BiStream::xdrInBuf.

43.10.2.3 BiStream (const char ∗ *s*, std::ios_base::openmode *mode*)

Constructor takes name of input file, mode. Constructor which takes a char∗ filename and int flags. Calls the base class open method with the filename and flags as arguments. Also allocates xdr stream.

References BiStream::inBuf, and BiStream::xdrInBuf.

43.10.2.4 ∼BiStream ()

Destructor, calls xdr_destroy to delete xdr stream. Destructor, destroys the xdr stream allocated in constructor

References BiStream::xdrInBuf.

43.10.3 Member Function Documentation

43.10.3.1 BiStream & operator>> (std::string & *ds*)

Binary Input stream operator>>. The std::string input operator must first read both the xdr buffer size and the size of the string written. Once these are read it can then read and convert the std::string correctly.

References BiStream::inBuf, and BiStream::xdrInBuf.

43.10.3.2 BiStream & operator>> (char ∗ *s*)

Input operator, reads char∗ from binary stream BiStream. Reading char array is a special case. The method has no way of knowing if the length to the input array is large enough, it assumes it is one char longer than actual string, (Null terminator added). As with the std::string the size of the xdr buffer as well as the char array size written must be read from the stream prior to reading and converting the char array.

References BiStream::inBuf, and BiStream::xdrInBuf.

The documentation for this class was generated from the following files:

- DakotaBinStream.H
- DakotaBinStream.C

43.11 BoStream Class Reference

The binary output stream class. Overloads the << operator for all data types.

Public Member Functions

- BoStream ()

  Default constructor, need to open.

- BoStream (const char *s)

  Constructor takes name of input file.

- BoStream (const char *s, std::ios_base::openmode mode)

  Constructor takes name of input file, mode.

- ~BoStream () throw ()

  Destructor, calls xdr_destroy to delete xdr stream.

- BoStream & operator<< (const std::string &ds)

  Binary Output stream operator<<.

- BoStream & operator<< (const char *s)

  Output operator, writes char TO binary stream BoStream.*

- BoStream & operator<< (const char &c)

  Output operator, writes char to binary stream BoStream.

- BoStream & operator<< (const int &i)

  Output operator, writes int to binary stream BoStream.

- BoStream & operator<< (const long &l)

  Output operator, writes long to binary stream BoStream.

- BoStream & operator<< (const short &s)

  Output operator, writes short to binary stream BoStream.

- BoStream & operator<< (const bool &b)

  Output operator, writes bool to binary stream BoStream.

- BoStream & operator<< (const double &d)

  Output operator, writes double to binary stream BoStream.

- BoStream & operator<< (const float &f)

  Output operator, writes float to binary stream BoStream.

- BoStream & operator<< (const unsigned char &c)

  Output operator, writes unsigned char to binary stream BoStream.

- BoStream & operator<< (const unsigned int &i)

  Output operator, writes unsigned int to binary stream BoStream.

- BoStream & operator<< (const unsigned long &l)

  Output operator, writes unsigned long to binary stream BoStream.

- BoStream & operator<< (const unsigned short &s)

  Output operator, writes unsigned short to binary stream BoStream.

Private Attributes

- XDR xdrOutBuf

  XDR output stream buffer.

- char outBuf [MAX_NETOBJ_SZ]

  Buffer to hold converted data before it is written.

43.11.1 Detailed Description

The binary output stream class. Overloads the << operator for all data types. The Dakota::BoStream class is a binary output classes which overloads the << operator for all standard data types (int, char, float, etc). The class relies on the built in write methods within the ostream base classes. Dakota::BoStream inherits from the ofstream class. The motivation to develop this class was to replace the Rogue wave class which Dakota historically used for binary I/O. If available, the class utilize rpc/xdr to construct machine independent binary files. These Dakota restart files can be moved between hosts.

43.11.2 Constructor & Destructor Documentation

43.11.2.1 BoStream ()

Default constructor, need to open. Default constructor allocates the xdr stream but does not call the open() method. The open() method must be called before stream can be written to.

References BoStream::outBuf, and BoStream::xdrOutBuf.

43.11.2.2 BoStream (const char $*$ *s*)

Constructor takes name of input file. Constructor, takes char $*$ filename as argument. Calls base class open method with filename and no other arguments. Also allocates xdr stream

References BoStream::outBuf, and BoStream::xdrOutBuf.

43.11.2.3 BoStream (const char * *s*, std::ios_base::openmode *mode*)

Constructor takes name of input file, mode. Constructor, takes char * filename and int flags as arguments. Calls base class open method with filename and flags as arguments. Also allocates xdr stream. Note : If no rpc/xdr support xdr calls are #ifdef'd out.

References BoStream::outBuf, and BoStream::xdrOutBuf.

43.11.3 Member Function Documentation

43.11.3.1 BoStream & operator<< (const std::string & *ds*)

Binary Output stream operator<<. The std::string operator<< must first write the xdr buffer size and the original string size to the stream. The input operator needs this information to be able to correctly read and convert the std::string.

References BoStream::outBuf, and BoStream::xdrOutBuf.

43.11.3.2 BoStream & operator<< (const char * *s*)

Output operator, writes char* TO binary stream BoStream. The output of char* is the same as the output of the std::string. The size of the xdr buffer and the size of the string must be written first, then the string itself.

References BoStream::outBuf, and BoStream::xdrOutBuf.

The documentation for this class was generated from the following files:

- DakotaBinStream.H
- DakotaBinStream.C

43.12 COLINApplication Class Reference

Public Member Functions

- COLINApplication ()

  Default constructor. Rrequired by COLIN's ApplicationHandle creation.

- COLINApplication (Model &model)

  Constructor with Model (not presently used).

- ~COLINApplication ()

  Destructor.

- void set_problem (Model &model)

  Helper function called after default construction to extract problem information from the Model and set it for COLIN.

- void set_blocking_synch (const bool blockingSynchFlag)

  publishes whether or not COLIN is operating synchronously

- virtual utilib::Any spawn_evaluation_impl (const utilib::Any &domain, const colin::AppRequest::request_map_t &requests, utilib::seed_t &seed)

  Schedule one or more requests at specified domain point, returning a DAKOTA-specific evaluation tracking ID.

- virtual bool evaluation_available ()

  Check to see if there are any function values ready to be collected.

- virtual void perform_evaluation_impl (const utilib::Any &domain, const colin::AppRequest::request_map_t &requests, utilib::seed_t &seed, colin::AppResponse::response_map_t &colin_responses)

  Perform a function evaluation at t given point.

- virtual utilib::Any collect_evaluation_impl (colin::AppResponse::response_map_t &responses, utilib::seed_t &seed)

  Collect a completed evaluation from DAKOTA.

- virtual void colin_request_to_dakota_request (const utilib::Any &domain, const colin::AppRequest::request_map_t &requests, utilib::seed_t &seed)

  Helper function to convert evaluation request data from COLIN structures to DAKOTA structures.

- virtual void dakota_response_to_colin_response (const Response &dakota_response, colin::AppResponse::response_map_t &colin_responses)

  Gelper function to convert evaluation response data from DAKOTA structures to COLIN structures.

- virtual bool map_domain (const utilib::Any &src, utilib::Any &native, bool forward=true) const

  Map the domain point into data type desired by this application context.

Protected Attributes

- Model iteratedModel

  Shallow copy of the model on which COLIN will iterate.

- bool blockingSynch

  Flag for COLIN synchronous behavior (Pattern Search only).

- ActiveSet activeSet

  Local copy of model's active set for convenience.

- std::vector< int > requestedEvals

  Evaluations queued for asynch evaluation.

- IntResponseMap dakota_responses

  eval_id to response mapping to cache completed jobs.

43.12.1 Detailed Description

COLINApplication is a DAKOTA class that is derived from COLIN's Application hierarchy. It redefines a variety of virtual COLIN functions to use the corresponding DAKOTA functions. This is a more flexible algorithm library interfacing approach than can be obtained with the function pointer approaches used by NPSOLOptimizer and SNLLOptimizer.

43.12.2 Member Function Documentation

43.12.2.1 void set_problem (Model & *model*)

Helper function called after default construction to extract problem information from the Model and set it for COLIN. Set variable bounds and linear and nonlinear constraints. This avoids using probDescDB, so it is called by both the standard and the on-the-fly COLINOptimizer constructors.

References Response::active_set(), COLINApplication::activeSet, Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::current_response(), Model::cv(), Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Model::discrete_int_lower_bounds(), Model::discrete_int_upper_bounds(), Model::div(), Model::drv(), COLINApplication::iteratedModel, Model::linear_eq_constraint_coeffs(), Model::linear_eq_constraint_targets(), Model::linear_ineq_constraint_coeffs(), Model::linear_ineq_constraint_lower_bounds(), Model::linear_ineq_constraint_upper_bounds(), Model::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Model::num_functions(), Model::num_linear_eq_constraints(), Model::num_linear_ineq_constraints(), Model::num_nonlinear_eq_constraints(), and Model::num_nonlinear_ineq_constraints().

Referenced by COLINApplication::COLINApplication().

43.12.2.2 utilib::Any spawn_evaluation_impl (const utilib::Any & *domain*, const colin::AppRequest::request_map_t & *requests*, utilib::seed_t & *seed*) `[virtual]`

Schedule one or more requests at specified domain point, returning a DAKOTA-specific evaluation tracking ID. Schedule one or more requests at specified domain point, returning a DAKOTA-specific evaluation tracking ID. This is only called by COLIN's concurrent evaluator, which is only instantiated when the Model supports asynch evals. The domain point is guaranteed to be compatible with data type specified by map_domain(...)

References COLINApplication::activeSet, Model::asynch_compute_response(), COLINApplication::colin_-request_to_dakota_request(), Model::evaluation_id(), and COLINApplication::iteratedModel.

43.12.2.3 bool evaluation_available () `[virtual]`

Check to see if there are any function values ready to be collected. Check to see if any asynchronous evaluations have finished. This is only called by COLIN's concurrent evaluator, which is only instantiated when the Model supports asynch evals.

References COLINApplication::dakota_responses, COLINApplication::iteratedModel, and Model::synchronize().

43.12.2.4 void perform_evaluation_impl (const utilib::Any & *domain*, const colin::AppRequest::request_map_t & *requests*, utilib::seed_t & *seed*, colin::AppResponse::response_map_t & *colin_responses*) `[virtual]`

Perform a function evaluation at t given point. Perform an evaluation at a specified domain point. Wait for and return the response. This is only called by COLIN's serial evaluator, which is only instantiated when the Model does not support asynch evals. The domain point is guaranteed to be compatible with data type specified by map_domain(...)

References COLINApplication::activeSet, COLINApplication::colin_request_to_dakota_request(), Model::compute_response(), Model::current_response(), COLINApplication::dakota_response_to_colin_-response(), and COLINApplication::iteratedModel.

43.12.2.5 utilib::Any collect_evaluation_impl (colin::AppResponse::response_map_t & *colin_responses*, utilib::seed_t & *seed*) `[virtual]`

Collect a completed evaluation from DAKOTA. Collect the next completed evaluation from DAKOTA. Always returns the evalid of the response returned.

References COLINApplication::dakota_response_to_colin_response(), and COLINApplication::dakota_-responses.

43.12.2.6 void colin_request_to_dakota_request (const utilib::Any & *domain*, const colin::AppRequest::request_map_t & *requests*, utilib::seed_t & *seed*) `[virtual]`

Helper function to convert evaluation request data from COLIN structures to DAKOTA structures. Map COLIN info requests to DAKOTA objectives and constraints.

References Model::continuous_variables(), Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Model::discrete_int_variables(), Model::discrete_real_variables(), Model::div(), Model::drv(), COLINApplication::iteratedModel, Model::num_functions(), and Dakota::set_index_to_value().

Referenced by COLINApplication::perform_evaluation_impl(), and COLINApplication::spawn_evaluation_impl().

43.12.2.7 void dakota_response_to_colin_response (const Response & *dakota_response*, colin::AppResponse::response_map_t & *colin_responses*) `[virtual]`

Gelper function to convert evaluation response data from DAKOTA structures to COLIN structures. Map DAKOTA objective and constraint values to COLIN response.

References Response::active_set_request_vector(), and Response::function_value().

Referenced by COLINApplication::collect_evaluation_impl(), and COLINApplication::perform_evaluation_impl().

43.12.2.8 bool map_domain (const utilib::Any & *src*, utilib::Any & *native*, bool *forward* = `true`) const `[virtual]`

Map the domain point into data type desired by this application context. Map the domain point into data type desired by this application context (utilib::MixedIntVars). This data type can be exposed from the Any &domain presented to spawn and collect.

The documentation for this class was generated from the following files:

- COLINApplication.H
- COLINApplication.C

43.13 COLINOptimizer Class Reference

Wrapper class for optimizers defined using COLIN. Inheritance diagram for COLINOptimizer::

Public Member Functions

- COLINOptimizer (Model &model)

  constructor

- COLINOptimizer (Model &model, int seed)

  alternate constructor for on-the-fly instantiations

- COLINOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor for Iterator instantiations by name

- ~COLINOptimizer ()

  destructor

- void find_optimum ()

  iterates the COLIN solver to determine the optimal solution

- bool returns_multiple_points () const

  some COLIN methods can return multiple points

Protected Member Functions

- void solver_setup (Model &model)

  convenience function for setting up the particular COLIN solver and appropriate Application

- void set_rng (int seed)

  sets up the random number generator for stochastic methods

- void set_solver_parameters ()

sets construct-time options for specific methods based on user specifications, including calling method-specific set functions

- void post_run (std::ostream &s)

  Get the final set of points from the solver Look up responses and sort, first according to constraint violation, then according to function value.

- void resize_final_points (size_t newsize)

  resize bestVariablesArray

- void resize_final_responses (size_t newsize)

  resize bestResponseArray

Protected Attributes

- short solverType

  COLIN solver sub-type as enumerated in COLINOptimizer.C.

- colin::SolverHandle colinSolver

  handle to the COLIN solver

- std::pair< colin::ApplicationHandle, COLINApplication * > colinProblem

  handle and pointer to the COLINApplication object

- colin::EvaluationManager_Base * colinEvalMgr

  pointer to the COLIN evalutaion manager object

- utilib::RNG * rng

  random number generator pointer

- bool blockingSynch

  the `synchronization` *setting: true if* `blocking`*, false if* `nonblocking`

- Real constraint_penalty

  Buffer to hold problem constraint_penalty parameter.

- bool constant_penalty

  Buffer to hold problem constant_penalty parameter.

43.13.1 Detailed Description

Wrapper class for optimizers defined using COLIN. The COLINOptimizer class wraps COLIN, a Sandia-developed C++ optimization interface library. A variety of COLIN optimizers are defined in COLIN and its

associated libraries, including SCOLIB which contains the optimization components from the old COLINY (formerly SGOPT) library. COLIN contains optimizers such as genetic algorithms, pattern search methods, and other nongradient-based techniques. COLINOptimizer uses a COLINApplication object to perform the function evaluations.

The user input mappings are as follows: `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, and `solution_accuracy` are mapped into COLIN's `max_iterations`, `max_function_evaluations_this_trial`, `function_value_tolerance`, `sufficient_objective_value` properties. An `outputLevel` is mapped to COLIN's `output_level` property and a setting of `debug` activates output of method initialization and sets the COLIN `debug` attribute to 10000 for the DEBUG output level. Refer to [Hart, W.E., 2006] for additional information on COLIN objects and controls.

43.13.2 Constructor & Destructor Documentation

43.13.2.1 COLINOptimizer (Model & *model*)

constructor Default constructor.

References ProblemDescDB::get_int(), Model::init_communicators(), Iterator::iteratedModel, Optimizer::localObjectiveRecast, Iterator::maxConcurrency, Iterator::probDescDB, Minimizer::scaleFlag, COLINOptimizer::set_rng(), COLINOptimizer::set_solver_parameters(), and COLINOptimizer::solver_setup().

43.13.2.2 COLINOptimizer (Model & *model*, int *seed*)

alternate constructor for on-the-fly instantiations Alternate constructor for on-the-fly instantiations.

References COLINOptimizer::set_rng(), COLINOptimizer::set_solver_parameters(), and COLINOptimizer::solver_setup().

43.13.2.3 COLINOptimizer (NoDBBaseConstructor, Model & *model*)

alternate constructor for Iterator instantiations by name Alternate constructor for Iterator instantiations by name.

References COLINOptimizer::set_solver_parameters(), and COLINOptimizer::solver_setup().

43.13.3 Member Function Documentation

43.13.3.1 void find_optimum () `[virtual]`

iterates the COLIN solver to determine the optimal solution find_optimum redefines the Optimizer virtual function to perform the optimization using COLIN. It first sets up the problem data, then executes optimize() on the COLIN solver and finally catalogues the results.

Implements Optimizer.

References Model::asynch_flag(), COLINOptimizer::colinEvalMgr, COLINOptimizer::colinProblem, COLINOptimizer::colinSolver, COLINOptimizer::constant_penalty, COLINOptimizer::constraint_penalty, Model::continuous_variables(), Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Model::discrete_int_variables(), Model::discrete_real_variables(), Model::evaluation_capacity(),

Iterator::iteratedModel, Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, Iterator::outputLevel, Dakota::set_value_to_index(), and COLINOptimizer::solverType.

43.13.3.2 bool returns_multiple_points () const [virtual]

some COLIN methods can return multiple points Designate which solvers can return multiple final points.

Reimplemented from Iterator.

References COLINOptimizer::solverType.

43.13.3.3 void solver_setup (Model & *model*) [protected]

convenience function for setting up the particular COLIN solver and appropriate Application This convenience function is called by the constructors in order to instantiate the solver.

References COLINOptimizer::colinProblem, COLINOptimizer::colinSolver, COLINOptimizer::constant_penalty, COLINOptimizer::constraint_penalty, ProblemDescDB::get_string(), Iterator::method_name(), Iterator::probDescDB, and COLINOptimizer::solverType.

Referenced by COLINOptimizer::COLINOptimizer().

43.13.3.4 void set_rng (int *seed*) [protected]

sets up the random number generator for stochastic methods Instantiate random number generator (RNG).

References COLINOptimizer::colinSolver, and COLINOptimizer::rng.

Referenced by COLINOptimizer::COLINOptimizer().

43.13.3.5 void set_solver_parameters () [protected]

sets construct-time options for specific methods based on user specifications, including calling method-specific set functions Sets solver properties based on user specifications. Called at construction time.

References Model::asynch_flag(), COLINOptimizer::blockingSynch, COLINOptimizer::colinSolver, COLINOptimizer::constant_penalty, COLINOptimizer::constraint_penalty, Iterator::convergenceTol, ProblemDescDB::get_bool(), ProblemDescDB::get_dsa(), ProblemDescDB::get_int(), ProblemDescDB::get_real(), ProblemDescDB::get_string(), ProblemDescDB::is_null(), Iterator::iteratedModel, Iterator::maxConcurrency, Iterator::maxFunctionEvals, Iterator::maxIterations, Iterator::numContinuousVars, Iterator::outputLevel, Iterator::probDescDB, and COLINOptimizer::solverType.

Referenced by COLINOptimizer::COLINOptimizer().

43.13.3.6 void post_run (std::ostream & *s*) [protected, virtual]

Get the final set of points from the solver Look up responses and sort, first according to constraint violation, then according to function value. This overrides Optimizer::post_run(). Do this because we need to unscale variables in order to look responses up in the database.

Reimplemented from Optimizer.

References Iterator::bestResponseArray, Iterator::bestVariablesArray, COLINOptimizer::colinProblem, COLINOptimizer::colinSolver, Variables::continuous_variables(), Response::copy(), Variables::copy(), Model::current_response(), Model::current_variables(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Dakota::data_pairs, Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Variables::discrete_int_variable(), Variables::discrete_real_variable(), Response::function_values(), Model::interface_id(), Iterator::iteratedModel, Optimizer::localObjectiveRecast, Dakota::lookup_by_val(), Minimizer::modify_s2n(), Model::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Model::num_nonlinear_eq_constraints(), Model::num_nonlinear_ineq_constraints(), Iterator::numContinuousVars, Iterator::numDiscreteIntVars, Iterator::numFinalSolutions, Iterator::numFunctions, Minimizer::numUserPrimaryFns, Minimizer::objective(), Model::primary_response_fn_weights(), COLINOptimizer::resize_final_points(), COLINOptimizer::resize_final_responses(), Minimizer::scaleFlag, Dakota::set_index_to_value(), Model::subordinate_model(), and Minimizer::varsScaleFlag.

The documentation for this class was generated from the following files:

- COLINOptimizer.H
- COLINOptimizer.C

43.14 CollaborativeHybridStrategy Class Reference

Strategy for hybrid minimization using multiple collaborating optimization and nonlinear least squares methods. Inheritance diagram for CollaborativeHybridStrategy::

Public Member Functions

- CollaborativeHybridStrategy (ProblemDescDB &problem_db)

  constructor

- ~CollaborativeHybridStrategy ()

  destructor

Protected Member Functions

- void run_strategy ()

  Performs the collaborative hybrid minimization strategy.

- const Variables & variables_results () const

  return the final solution from the collaborative minimization (variables)

- const Response & response_results () const

  return the final solution from the collaborative minimization (response)

Private Attributes

- String hybridCollabType

  abo or hops

- Variables bestVariables

  best variables found in minimization

- Response bestResponse

  best response found in minimization

43.14.1 Detailed Description

Strategy for hybrid minimization using multiple collaborating optimization and nonlinear least squares methods. This strategy has two approaches to hybrid minimization: (1) agent-based using the ABO framework; (2) nonagent-based using the HOPSPACK framework.

The documentation for this class was generated from the following files:

- CollaborativeHybridStrategy.H
- CollaborativeHybridStrategy.C

43.15 CommandLineHandler Class Reference

Utility class for managing command line inputs to DAKOTA. Inheritance diagram for CommandLineHandler::

Public Member Functions

- CommandLineHandler ()

  default constructor, requires check_usage() call for parsing

- CommandLineHandler (int argc, char **argv)

  constructor with parsing

- ~CommandLineHandler ()

  destructor

- void check_usage (int argc, char **argv)

  Verifies that DAKOTA is called with the correct command usage. Prints a descriptive message and exits the program if incorrect.

- int read_restart_evals () const

  Returns the number of evaluations to be read from the restart file (as specified on the DAKOTA command line) as an integer instead of a const char.*

- bool instantiate_flag () const

  Whether command line args dictate instantiation of objects for run.

Private Member Functions

- void initialize_options ()

  enrolls the supported command line inputs.

- void output_version (std::ostream &s) const

  outputs the DAKOTA version

43.15.1 Detailed Description

Utility class for managing command line inputs to DAKOTA. CommandLineHandler provides additional functionality that is specific to DAKOTA's needs for the definition and parsing of command line options. Inheritance is used to allow the class to have all the functionality of the base class, GetLongOpt.

43.15.2 Member Function Documentation

43.15.2.1 bool instantiate_flag () const [inline]

Whether command line args dictate instantiation of objects for run. Instantiate objects if not just getting help or version

References GetLongOpt::retrieve().

Referenced by main().

The documentation for this class was generated from the following files:

- CommandLineHandler.H
- CommandLineHandler.C

43.16 CommandShell Class Reference

Utility class which defines convenience operators for spawning processes with system calls.

Public Member Functions

- CommandShell (const std::string &work_dir)

  constructor

- ~CommandShell ()

  destructor

- CommandShell & operator<< (const char *cmd)

  appends cmd to sysCommand

- CommandShell & operator<< (const std::string &cmd)

  convenient operator: appends string to the commandString to be executed

- CommandShell & operator<< (CommandShell &(*f)(CommandShell &))

  allows passing of the flush function to the shell using <<

- CommandShell & flush ()

  "flushes" the shell; i.e. executes the sysCommand

- void asynch_flag (const bool flag)

  set the asynchFlag

- bool asynch_flag () const

  get the asynchFlag

- void suppress_output_flag (const bool flag)

  set the suppressOutputFlag

- bool suppress_output_flag () const

  get the suppressOutputFlag

Private Attributes

- const std::string & workDir

  To convey working directory when useWorkdir is true:.

- std::string sysCommand

  The command string that is constructed through one or more << insertions and then executed by flush.

- bool asynchFlag

  flags nonblocking operation (background system calls)

- bool suppressOutputFlag

  flags suppression of shell output (no command echo)

43.16.1 Detailed Description

Utility class which defines convenience operators for spawning processes with system calls. The CommandShell class wraps the C system() utility and defines convenience operators for building a command string and then passing it to the shell.

43.16.2 Member Function Documentation

43.16.2.1 CommandShell & operator<< (const char * *cmd*) `[inline]`

appends cmd to sysCommand convenient operator: appends string to the commandString to be executed

References CommandShell::sysCommand.

43.16.2.2 CommandShell & operator<< (CommandShell &(*)(CommandShell &) *f*) `[inline]`

allows passing of the flush function to the shell using << convenience operator: allows passing of the flush func to the shell via <<

43.16.2.3 CommandShell & flush ()

"flushes" the shell; i.e. executes the sysCommand Executes the sysCommand by passing it to system(). Appends an "&" if asynchFlag is set (background system call) and echos the sysCommand to Cout if suppressOutputFlag is not set.

References Dakota::abort_handler(), CommandShell::asynchFlag, CommandShell::suppressOutputFlag, CommandShell::sysCommand, and CommandShell::workDir.

The documentation for this class was generated from the following files:

- CommandShell.H
- CommandShell.C

43.17 ConcurrentStrategy Class Reference

Strategy for multi-start iteration or pareto set optimization. Inheritance diagram for ConcurrentStrategy::

Public Member Functions

- ConcurrentStrategy (ProblemDescDB &problem_db)

  constructor

- ~ConcurrentStrategy ()

  destructor

Protected Member Functions

- void run_strategy ()

  Performs the concurrent strategy by executing selectedIterator on userDefinedModel multiple times in parallel for different settings within the iterator or model.

- void initialize_iterator (int job_index)

  initialize the iterator about to be executed within a parallel iterator scheduling function (serve_iterators() or static_schedule_iterators())

- void pack_parameters_buffer (MPIPackBuffer &send_buffer, int job_index)

  pack a send_buffer for assigning an iterator job to a server

- void unpack_parameters_buffer (MPIUnpackBuffer &recv_buffer)

  unpack a recv_buffer for accepting an iterator job from the scheduler

- void pack_results_buffer (MPIPackBuffer &send_buffer, int job_index)

  pack a send_buffer for returning iterator results from a server

- void unpack_results_buffer (MPIUnpackBuffer &recv_buffer, int job_index)

  unpack a recv_buffer for accepting iterator results from a server

- void update_local_results (int job_index)

  update local PRP results arrays with current iteration results

Private Member Functions

- void initialize_iterator (const RealVector &param_set)

  called by unpack_parameters_buffer(MPIUnpackBuffer) and initialize_iterator(int) to update userDefinedModel and selectedIterator

- void print_results () const

  prints the concurrent iteration results summary (called by run_strategy())

Private Attributes

- Model userDefinedModel

  the model used by the iterator

- Iterator selectedIterator

  the iterator used by the concurrent strategy

- bool multiStartFlag

  distinguishes multi-start from Pareto-set

- RealVector initialPt

  the initial continuous variables for restoring the starting point in the Pareto set strategy

- RealVectorArray parameterSets

  an array of parameter set vectors (either multistart variable sets or pareto multi-objective/least squares weighting sets) to be performed.

- PRPArray prpResults

  1-d array of ParamResponsePair results corresponding to numIteratorJobs

43.17.1 Detailed Description

Strategy for multi-start iteration or pareto set optimization. This strategy maintains two concurrent iterator capabilities. First, a general capability for running an iterator multiple times from different starting points is provided (often used for multi-start optimization, but not restricted to optimization). Second, a simple capability for mapping the "pareto frontier" (the set of optimal solutions in multiobjective formulations) is provided. This pareto set is mapped through running an optimizer multiple times for different sets of multiobjective weightings.

43.17.2 Member Function Documentation

43.17.2.1 void pack_parameters_buffer (MPIPackBuffer & *send_buffer*, int *job_index*) `[inline, protected, virtual]`

pack a send_buffer for assigning an iterator job to a server This virtual function redefinition is executed on the dedicated master processor for self scheduling. It is not used for peer partitions.

Reimplemented from Strategy.

References ConcurrentStrategy::parameterSets.

43.17.2.2 void unpack_parameters_buffer (MPIUnpackBuffer & *recv_buffer*) [inline, protected, virtual]

unpack a recv_buffer for accepting an iterator job from the scheduler This virtual function redefinition is executed on an iterator server for dedicated master self scheduling. It is not used for peer partitions.

Reimplemented from Strategy.

References ConcurrentStrategy::initialize_iterator().

43.17.2.3 void pack_results_buffer (MPIPackBuffer & *send_buffer*, int *job_index*) [inline, protected, virtual]

pack a send_buffer for returning iterator results from a server This virtual function redefinition is executed either on an iterator server for dedicated master self scheduling or on peers 2 through n for static scheduling.

Reimplemented from Strategy.

References ConcurrentStrategy::prpResults.

43.17.2.4 void unpack_results_buffer (MPIUnpackBuffer & *recv_buffer*, int *job_index*) [inline, protected, virtual]

unpack a recv_buffer for accepting iterator results from a server This virtual function redefinition is executed on an strategy master (either the dedicated master processor for self scheduling or peer 1 for static scheduling).

Reimplemented from Strategy.

References ConcurrentStrategy::prpResults.

The documentation for this class was generated from the following files:

- ConcurrentStrategy.H
- ConcurrentStrategy.C

43.18 CONMINOptimizer Class Reference

Wrapper class for the CONMIN optimization library. Inheritance diagram for CONMINOptimizer::

Public Member Functions

- CONMINOptimizer (Model &model)

  standard constructor

- CONMINOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor

- ~CONMINOptimizer ()

  destructor

- void find_optimum ()

  Used within the optimizer branch for computing the optimal solution. Redefines the run virtual function for the optimizer branch.

Protected Member Functions

- void initialize_run ()

  performs run-time set up

Private Member Functions

- void initialize ()

  Shared constructor code.

- void allocate_workspace ()

  Allocates workspace for the optimizer.

- void deallocate_workspace ()

  Releases workspace memory.

- void allocate_constraints ()

  Allocates constraint mappings.

Private Attributes

- int conminInfo

  INFO from CONMIN manual.

- int printControl

  IPRINT from CONMIN manual (controls output verbosity).

- int optimizationType

  MINMAX from DOT manual (minimize or maximize).

- Real objFnValue

  value of the objective function passed to CONMIN

- RealVector constraintValues

  array of nonlinear constraint values passed to CONMIN

- int numConminNlnConstr

  total number of nonlinear constraints seen by CONMIN

- int numConminLinConstr

  total number of linear constraints seen by CONMIN

- int numConminConstr

  total number of linear and nonlinear constraints seen by CONMIN

- SizetArray constraintMappingIndices

  a container of indices for referencing the corresponding Response constraints used in computing the CONMIN constraints.

- RealArray constraintMappingMultipliers

  a container of multipliers for mapping the Response constraints to the CONMIN constraints.

- RealArray constraintMappingOffsets

  a container of offsets for mapping the Response constraints to the CONMIN constraints.

- int N1

  Size variable for CONMIN arrays. See CONMIN manual.

- int N2

  Size variable for CONMIN arrays. See CONMIN manual.

- int N3

  Size variable for CONMIN arrays. See CONMIN manual.

- int N4

  Size variable for CONMIN arrays. See CONMIN manual.

- int N5

  Size variable for CONMIN arrays. See CONMIN manual.

- int NFDG

  Finite difference flag.

- int IPRINT

  Flag to control amount of output data.

- int ITMAX

  Flag to specify the maximum number of iterations.

- double FDCH

  Relative finite difference step size.

- double FDCHM

  Absolute finite difference step size.

- double CT

  Constraint thickness parameter.

- double CTMIN

  Minimum absolute value of CT used during optimization.

- double CTL

  Constraint thickness parameter for linear and side constraints.

- double CTLMIN

  Minimum value of CTL used during optimization.

- double DELFUN

  Relative convergence criterion threshold.

- double DABFUN

  Absolute convergence criterion threshold.

- double * conminDesVars

Array of design variables used by CONMIN (length N1 = numdv+2).

- double * conminLowerBnds

  Array of lower bounds used by CONMIN (length N1 = numdv+2).

- double * conminUpperBnds

  Array of upper bounds used by CONMIN (length N1 = numdv+2).

- double * S

  Internal CONMIN array.

- double * G1

  Internal CONMIN array.

- double * G2

  Internal CONMIN array.

- double * B

  Internal CONMIN array.

- double * C

  Internal CONMIN array.

- int * MS1

  Internal CONMIN array.

- double * SCAL

  Internal CONMIN array.

- double * DF

  Internal CONMIN array.

- double * A

  Internal CONMIN array.

- int * ISC

  Internal CONMIN array.

- int * IC

  Internal CONMIN array.

43.18.1 Detailed Description

Wrapper class for the CONMIN optimization library. The CONMINOptimizer class provides a wrapper for CONMIN, a Public-domain Fortran 77 optimization library written by Gary Vanderplaats under contract to NASA Ames Research Center. The CONMIN User's Manual is contained in NASA Technical Memorandum X-62282, 1978. CONMIN uses a reverse communication mode, which avoids the static member function issues that arise with function pointer designs (see NPSOLOptimizer and SNLLOptimizer).

The user input mappings are as follows: `max_iterations` is mapped into CONMIN's `ITMAX` parameter, `max_function_evaluations` is implemented directly in the find_optimum() loop since there is no CONMIN parameter equivalent, `convergence_tolerance` is mapped into CONMIN's `DELFUN` and `DABFUN` parameters, `output` verbosity is mapped into CONMIN's `IPRINT` parameter (verbose: `IPRINT` = 4; quiet: `IPRINT` = 2), gradient mode is mapped into CONMIN's `NFDG` parameter, and finite difference step size is mapped into CONMIN's `FDCH` and `FDCHM` parameters. Refer to [Vanderplaats, 1978] for additional information on CONMIN parameters.

43.18.2 Member Data Documentation

43.18.2.1 int conminInfo [private]

INFO from CONMIN manual. Information requested by CONMIN: 1 = evaluate objective and constraints, 2 = evaluate gradients of objective and constraints.

Referenced by CONMINOptimizer::find_optimum(), and CONMINOptimizer::initialize().

43.18.2.2 int printControl [private]

IPRINT from CONMIN manual (controls output verbosity). Values range from 0 (nothing) to 4 (most output). 0 = nothing, 1 = initial and final function information, 2 = all of #1 plus function value and design vars at each iteration, 3 = all of #2 plus constraint values and direction vectors, 4 = all of #3 plus gradients of the objective function and constraints, 5 = all of #4 plus proposed design vector, plus objective and constraint functions from the 1-D search

Referenced by CONMINOptimizer::initialize().

43.18.2.3 int optimizationType [private]

MINMAX from DOT manual (minimize or maximize). Values of 0 or -1 (minimize) or 1 (maximize).

43.18.2.4 RealVector constraintValues [private]

array of nonlinear constraint values passed to CONMIN This array must be of nonzero length and must contain only one-sided inequality constraints which are <= 0 (which requires a transformation from 2-sided inequalities and equalities).

Referenced by CONMINOptimizer::allocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.5 SizetArray constraintMappingIndices [private]

a container of indices for referencing the corresponding Response constraints used in computing the CONMIN constraints. The length of the container corresponds to the number of CONMIN constraints, and each entry in the container points to the corresponding DAKOTA constraint.

Referenced by CONMINOptimizer::allocate_constraints(), and CONMINOptimizer::find_optimum().

43.18.2.6 RealArray constraintMappingMultipliers [private]

a container of multipliers for mapping the Response constraints to the CONMIN constraints. The length of the container corresponds to the number of CONMIN constraints, and each entry in the container stores a multiplier for the DAKOTA constraint identified with constraintMappingIndices. These multipliers are currently +1 or -1.

Referenced by CONMINOptimizer::allocate_constraints(), and CONMINOptimizer::find_optimum().

43.18.2.7 RealArray constraintMappingOffsets [private]

a container of offsets for mapping the Response constraints to the CONMIN constraints. The length of the container corresponds to the number of CONMIN constraints, and each entry in the container stores an offset for the DAKOTA constraint identified with constraintMappingIndices. These offsets involve inequality bounds or equality targets, since CONMIN assumes constraint allowables = 0.

Referenced by CONMINOptimizer::allocate_constraints(), and CONMINOptimizer::find_optimum().

43.18.2.8 int N1 [private]

Size variable for CONMIN arrays. See CONMIN manual. N1 = number of variables + 2

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::find_optimum(), and CONMINOptimizer::initialize_run().

43.18.2.9 int N2 [private]

Size variable for CONMIN arrays. See CONMIN manual. N2 = number of constraints + 2∗(number of variables)

Referenced by CONMINOptimizer::allocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.10 int N3 [private]

Size variable for CONMIN arrays. See CONMIN manual. N3 = Maximum possible number of active constraints.

Referenced by CONMINOptimizer::allocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.11 int N4 [private]

Size variable for CONMIN arrays. See CONMIN manual. N4 = Maximum(N3,number of variables)

Referenced by CONMINOptimizer::allocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.12 int N5 [private]

Size variable for CONMIN arrays. See CONMIN manual. N5 = 2∗(N4)

Referenced by CONMINOptimizer::allocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.13 double CT [private]

Constraint thickness parameter. The value of CT decreases in magnitude during optimization.

Referenced by CONMINOptimizer::find_optimum(), and CONMINOptimizer::initialize().

43.18.2.14 double∗ S [private]

Internal CONMIN array. Move direction in N-dimensional space.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.15 double∗ G1 [private]

Internal CONMIN array. Temporary storage of constraint values.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.16 double∗ G2 [private]

Internal CONMIN array. Temporary storage of constraint values.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.17 double∗ B [private]

Internal CONMIN array. Temporary storage for computations involving array S.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.18 double∗ C [private]

Internal CONMIN array. Temporary storage for use with arrays B and S.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.19 int* MS1 [private]

Internal CONMIN array. Temporary storage for use with arrays B and S.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.20 double* SCAL [private]

Internal CONMIN array. Vector of scaling parameters for design parameter values.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.21 double* DF [private]

Internal CONMIN array. Temporary storage for analytic gradient data.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.22 double* A [private]

Internal CONMIN array. Temporary 2-D array for storage of constraint gradients.

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), and CONMINOptimizer::find_optimum().

43.18.2.23 int* ISC [private]

Internal CONMIN array. Array of flags to identify linear constraints. (not used in this implementation of CONMIN)

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), CONMINOptimizer::find_optimum(), and CONMINOptimizer::initialize_run().

43.18.2.24 int* IC [private]

Internal CONMIN array. Array of flags to identify active and violated constraints

Referenced by CONMINOptimizer::allocate_workspace(), CONMINOptimizer::deallocate_workspace(), CONMINOptimizer::find_optimum(), and CONMINOptimizer::initialize_run().

The documentation for this class was generated from the following files:

- CONMINOptimizer.H
- CONMINOptimizer.C

43.19 Constraints Class Reference

Base class for the variable constraints class hierarchy. Inheritance diagram for Constraints::

Public Member Functions

- Constraints ()

  default constructor

- Constraints (const ProblemDescDB &prob_db, const SharedVariablesData &svd)

  standard constructor

- Constraints (const SharedVariablesData &svd)

  alternate constructor for instantiations on the fly

- Constraints (const Constraints &con)

  copy constructor

- virtual ~Constraints ()

  destructor

- Constraints operator= (const Constraints &con)

  assignment operator

- virtual void write (std::ostream &s) const

  write a variable constraints object to an std::ostream

- virtual void read (std::istream &s)

  read a variable constraints object from an std::istream

- virtual void reshape (const SizetArray &vc_totals)

  reshape the lower/upper bound arrays within the Constraints hierarchy

- const RealVector & continuous_lower_bounds () const

  return the active continuous variable lower bounds

- void continuous_lower_bounds (const RealVector &cl_bnds)

  set the active continuous variable lower bounds

- void continuous_lower_bound (const Real &cl_bnd, const size_t &i)
  set an active continuous variable lower bound
- const RealVector & continuous_upper_bounds () const
  return the active continuous variable upper bounds
- void continuous_upper_bounds (const RealVector &cu_bnds)
  set the active continuous variable upper bounds
- void continuous_upper_bound (const Real &cu_bnd, const size_t &i)
  set an active continuous variable upper bound
- const IntVector & discrete_int_lower_bounds () const
  return the active discrete variable lower bounds
- void discrete_int_lower_bounds (const IntVector &dil_bnds)
  set the active discrete variable lower bounds
- void discrete_int_lower_bound (const int &dil_bnd, const size_t &i)
  set an active discrete variable lower bound
- const IntVector & discrete_int_upper_bounds () const
  return the active discrete variable upper bounds
- void discrete_int_upper_bounds (const IntVector &diu_bnds)
  set the active discrete variable upper bounds
- void discrete_int_upper_bound (const int &diu_bnd, const size_t &i)
  set an active discrete variable upper bound
- const RealVector & discrete_real_lower_bounds () const
  return the active discrete variable lower bounds
- void discrete_real_lower_bounds (const RealVector &drl_bnds)
  set the active discrete variable lower bounds
- void discrete_real_lower_bound (const Real &drl_bnd, const size_t &i)
  set an active discrete variable lower bound
- const RealVector & discrete_real_upper_bounds () const
  return the active discrete variable upper bounds
- void discrete_real_upper_bounds (const RealVector &dru_bnds)
  set the active discrete variable upper bounds
- void discrete_real_upper_bound (const Real &dru_bnd, const size_t &i)

set an active discrete variable upper bound

- const RealVector & inactive_continuous_lower_bounds () const

  return the inactive continuous lower bounds

- void inactive_continuous_lower_bounds (const RealVector &icl_bnds)

  set the inactive continuous lower bounds

- const RealVector & inactive_continuous_upper_bounds () const

  return the inactive continuous upper bounds

- void inactive_continuous_upper_bounds (const RealVector &icu_bnds)

  set the inactive continuous upper bounds

- const IntVector & inactive_discrete_int_lower_bounds () const

  return the inactive discrete lower bounds

- void inactive_discrete_int_lower_bounds (const IntVector &idil_bnds)

  set the inactive discrete lower bounds

- const IntVector & inactive_discrete_int_upper_bounds () const

  return the inactive discrete upper bounds

- void inactive_discrete_int_upper_bounds (const IntVector &idiu_bnds)

  set the inactive discrete upper bounds

- const RealVector & inactive_discrete_real_lower_bounds () const

  return the inactive discrete lower bounds

- void inactive_discrete_real_lower_bounds (const RealVector &idrl_bnds)

  set the inactive discrete lower bounds

- const RealVector & inactive_discrete_real_upper_bounds () const

  return the inactive discrete upper bounds

- void inactive_discrete_real_upper_bounds (const RealVector &idru_bnds)

  set the inactive discrete upper bounds

- const RealVector & all_continuous_lower_bounds () const

  returns a single array with all continuous lower bounds

- void all_continuous_lower_bounds (const RealVector &acl_bnds)

  sets all continuous lower bounds using a single array

- void all_continuous_lower_bound (const Real &acl_bnd, const size_t &i)

  set a lower bound within the all continuous lower bounds array

- const RealVector & all_continuous_upper_bounds () const
  returns a single array with all continuous upper bounds
- void all_continuous_upper_bounds (const RealVector &acu_bnds)
  sets all continuous upper bounds using a single array
- void all_continuous_upper_bound (const Real &acu_bnd, const size_t &i)
  set an upper bound within the all continuous upper bounds array
- const IntVector & all_discrete_int_lower_bounds () const
  returns a single array with all discrete lower bounds
- void all_discrete_int_lower_bounds (const IntVector &adil_bnds)
  sets all discrete lower bounds using a single array
- void all_discrete_int_lower_bound (const int &adil_bnd, const size_t &i)
  set a lower bound within the all discrete lower bounds array
- const IntVector & all_discrete_int_upper_bounds () const
  returns a single array with all discrete upper bounds
- void all_discrete_int_upper_bounds (const IntVector &adiu_bnds)
  sets all discrete upper bounds using a single array
- void all_discrete_int_upper_bound (const int &adiu_bnd, const size_t &i)
  set an upper bound within the all discrete upper bounds array
- const RealVector & all_discrete_real_lower_bounds () const
  returns a single array with all discrete lower bounds
- void all_discrete_real_lower_bounds (const RealVector &adrl_bnds)
  sets all discrete lower bounds using a single array
- void all_discrete_real_lower_bound (const Real &adrl_bnd, const size_t &i)
  set a lower bound within the all discrete lower bounds array
- const RealVector & all_discrete_real_upper_bounds () const
  returns a single array with all discrete upper bounds
- void all_discrete_real_upper_bounds (const RealVector &adru_bnds)
  sets all discrete upper bounds using a single array
- void all_discrete_real_upper_bound (const Real &adru_bnd, const size_t &i)
  set an upper bound within the all discrete upper bounds array

- size_t num_linear_ineq_constraints () const

  return the number of linear inequality constraints

- size_t num_linear_eq_constraints () const

  return the number of linear equality constraints

- const RealMatrix & linear_ineq_constraint_coeffs () const

  return the linear inequality constraint coefficients

- void linear_ineq_constraint_coeffs (const RealMatrix &lin_ineq_coeffs)

  set the linear inequality constraint coefficients

- const RealVector & linear_ineq_constraint_lower_bounds () const

  return the linear inequality constraint lower bounds

- void linear_ineq_constraint_lower_bounds (const RealVector &lin_ineq_l_bnds)

  set the linear inequality constraint lower bounds

- const RealVector & linear_ineq_constraint_upper_bounds () const

  return the linear inequality constraint upper bounds

- void linear_ineq_constraint_upper_bounds (const RealVector &lin_ineq_u_bnds)

  set the linear inequality constraint upper bounds

- const RealMatrix & linear_eq_constraint_coeffs () const

  return the linear equality constraint coefficients

- void linear_eq_constraint_coeffs (const RealMatrix &lin_eq_coeffs)

  set the linear equality constraint coefficients

- const RealVector & linear_eq_constraint_targets () const

  return the linear equality constraint targets

- void linear_eq_constraint_targets (const RealVector &lin_eq_targets)

  set the linear equality constraint targets

- size_t num_nonlinear_ineq_constraints () const

  return the number of nonlinear inequality constraints

- size_t num_nonlinear_eq_constraints () const

  return the number of nonlinear equality constraints

- const RealVector & nonlinear_ineq_constraint_lower_bounds () const

  return the nonlinear inequality constraint lower bounds

- void nonlinear_ineq_constraint_lower_bounds (const RealVector &nln_ineq_l_bnds)

set the nonlinear inequality constraint lower bounds

- const RealVector & nonlinear_ineq_constraint_upper_bounds () const

  return the nonlinear inequality constraint upper bounds

- void nonlinear_ineq_constraint_upper_bounds (const RealVector &nln_ineq_u_bnds)

  set the nonlinear inequality constraint upper bounds

- const RealVector & nonlinear_eq_constraint_targets () const

  return the nonlinear equality constraint targets

- void nonlinear_eq_constraint_targets (const RealVector &nln_eq_targets)

  set the nonlinear equality constraint targets

- Constraints copy () const

  for use when a deep copy is needed (the representation is _not_ shared)

- void reshape (const size_t &num_nln_ineq_cons, const size_t &num_nln_eq_cons, const size_t &num_lin_ineq_cons, const size_t &num_lin_eq_cons, const SizetArray &vc_totals)

  reshape the linear/nonlinear/bound constraint arrays arrays within the Constraints hierarchy

- void reshape (const size_t &num_nln_ineq_cons, const size_t &num_nln_eq_cons, const size_t &num_lin_ineq_cons, const size_t &num_lin_eq_cons)

  reshape the linear/nonlinear constraint arrays within the Constraints hierarchy

- void inactive_view (short view2)

  sets the inactive view based on higher level (nested) context

- bool is_null () const

  function to check constraintsRep (does this envelope contain a letter)

Protected Member Functions

- Constraints (BaseConstructor, const ProblemDescDB &problem_db, const SharedVariablesData &svd)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- Constraints (BaseConstructor, const SharedVariablesData &svd)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- virtual void build_active_views ()

  construct active views of all variables bounds arrays

- virtual void build_inactive_views ()

construct inactive views of all variables bounds arrays

- void build_views ()

  construct active/inactive views of all variables arrays

- void manage_linear_constraints (const ProblemDescDB &problem_db)

  perform checks on user input, convert linear constraint coefficient input to matrices, and assign defaults

Protected Attributes

- SharedVariablesData sharedVarsData

  configuration data shared from a Variables instance

- RealVector allContinuousLowerBnds

  a continuous lower bounds array combining continuous design, uncertain, and continuous state variable types (all view).

- RealVector allContinuousUpperBnds

  a continuous upper bounds array combining continuous design, uncertain, and continuous state variable types (all view).

- IntVector allDiscreteIntLowerBnds

  a discrete lower bounds array combining discrete design and discrete state variable types (all view).

- IntVector allDiscreteIntUpperBnds

  a discrete upper bounds array combining discrete design and discrete state variable types (all view).

- RealVector allDiscreteRealLowerBnds

  a discrete lower bounds array combining discrete design and discrete state variable types (all view).

- RealVector allDiscreteRealUpperBnds

  a discrete upper bounds array combining discrete design and discrete state variable types (all view).

- size_t numNonlinearIneqCons

  number of nonlinear inequality constraints

- size_t numNonlinearEqCons

  number of nonlinear equality constraints

- RealVector nonlinearIneqConLowerBnds

  nonlinear inequality constraint lower bounds

- RealVector nonlinearIneqConUpperBnds

  nonlinear inequality constraint upper bounds

- RealVector nonlinearEqConTargets

  nonlinear equality constraint targets

- size_t numLinearIneqCons

  number of linear inequality constraints

- size_t numLinearEqCons

  number of linear equality constraints

- RealMatrix linearIneqConCoeffs

  linear inequality constraint coefficients

- RealMatrix linearEqConCoeffs

  linear equality constraint coefficients

- RealVector linearIneqConLowerBnds

  linear inequality constraint lower bounds

- RealVector linearIneqConUpperBnds

  linear inequality constraint upper bounds

- RealVector linearEqConTargets

  linear equality constraint targets

- RealVector continuousLowerBnds

  the active continuous lower bounds array view

- RealVector continuousUpperBnds

  the active continuous upper bounds array view

- IntVector discreteIntLowerBnds

  the active discrete lower bounds array view

- IntVector discreteIntUpperBnds

  the active discrete upper bounds array view

- RealVector discreteRealLowerBnds

  the active discrete lower bounds array view

- RealVector discreteRealUpperBnds

  the active discrete upper bounds array view

- RealVector inactiveContinuousLowerBnds

  the inactive continuous lower bounds array view

- RealVector inactiveContinuousUpperBnds

the inactive continuous upper bounds array view

- IntVector inactiveDiscreteIntLowerBnds

  the inactive discrete lower bounds array view

- IntVector inactiveDiscreteIntUpperBnds

  the inactive discrete upper bounds array view

- RealVector inactiveDiscreteRealLowerBnds

  the inactive discrete lower bounds array view

- RealVector inactiveDiscreteRealUpperBnds

  the inactive discrete upper bounds array view

Private Member Functions

- Constraints * get_constraints (const ProblemDescDB &problem_db, const SharedVariablesData &svd)

  Used only by the constructor to initialize constraintsRep to the appropriate derived type.

- Constraints * get_constraints (const SharedVariablesData &svd) const

  Used by copy() to initialize constraintsRep to the appropriate derived type.

Private Attributes

- Constraints * constraintsRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing constraintsRep

43.19.1 Detailed Description

Base class for the variable constraints class hierarchy. The Constraints class is the base class for the class hierarchy managing bound, linear, and nonlinear constraints. Using the variable lower and upper bounds arrays from the input specification, different derived classes define different views of this data. The linear and nonlinear constraint data is consistent in all views and is managed at the base class level. For memory efficiency and enhanced polymorphism, the variable constraints hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Constraints) serves as the envelope and one of the derived classes (selected in Constraints::get_constraints()) serves as the letter.

43.19.2 Constructor & Destructor Documentation

43.19.2.1 Constraints ()

default constructor The default constructor: constraintsRep is NULL in this case (a populated problem_db is needed to build a meaningful Constraints object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

43.19.2.2 Constraints (const ProblemDescDB & *problem_db*, const SharedVariablesData & *svd*)

standard constructor The envelope constructor only needs to extract enough data to properly execute get_-constraints, since the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived classes.

References Dakota::abort_handler(), Constraints::constraintsRep, and Constraints::get_constraints().

43.19.2.3 Constraints (const SharedVariablesData & *svd*)

alternate constructor for instantiations on the fly Envelope constructor for instantiations on the fly. This constructor executes get_constraints(view), which invokes the default derived/base constructors, followed by a reshape() based on vars_comps.

References Dakota::abort_handler(), Constraints::constraintsRep, and Constraints::get_constraints().

43.19.2.4 Constraints (const Constraints & *con*)

copy constructor Copy constructor manages sharing of constraintsRep and incrementing of referenceCount.

References Constraints::constraintsRep, and Constraints::referenceCount.

43.19.2.5 ~Constraints () `[virtual]`

destructor Destructor decrements referenceCount and only deletes constraintsRep when referenceCount reaches zero.

References Constraints::constraintsRep, and Constraints::referenceCount.

43.19.2.6 Constraints (BaseConstructor, const ProblemDescDB & *problem_db*, const SharedVariablesData & *svd*) `[protected]`

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_constraints() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling get_-constraints() again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ~Constraints).

43.19.2.7 Constraints (BaseConstructor, const SharedVariablesData & *svd*) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_constraints() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling get_constraints() again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ∼Constraints).

43.19.3 Member Function Documentation

43.19.3.1 Constraints operator= (const Constraints & *con*)

assignment operator Assignment operator decrements referenceCount for old constraintsRep, assigns new constraintsRep, and increments referenceCount for new constraintsRep.

References Constraints::constraintsRep, and Constraints::referenceCount.

43.19.3.2 void reshape (const SizetArray & *vc_totals*) [virtual]

reshape the lower/upper bound arrays within the Constraints hierarchy Resizes the derived bounds arrays.

Reimplemented in MergedConstraints, and MixedConstraints.

References Constraints::constraintsRep, Constraints::continuousLowerBnds, Constraints::discreteIntLowerBnds, Constraints::discreteRealLowerBnds, Constraints::linearEqConCoeffs, Constraints::linearIneqConCoeffs, Constraints::numLinearEqCons, Constraints::numLinearIneqCons, and Constraints::reshape().

Referenced by DataFitSurrModel::DataFitSurrModel(), RecastModel::RecastModel(), and Constraints::reshape().

43.19.3.3 Constraints copy () const

for use when a deep copy is needed (the representation is _not_ shared) Deep copies are used for history mechanisms that catalogue permanent copies (should not change as the representation within userDefinedConstraints changes).

References Constraints::allContinuousLowerBnds, Constraints::allContinuousUpperBnds, Constraints::allDiscreteIntLowerBnds, Constraints::allDiscreteIntUpperBnds, Constraints::allDiscreteRealLowerBnds, Constraints::allDiscreteRealUpperBnds, Constraints::build_views(), Constraints::constraintsRep, Constraints::get_constraints(), Constraints::linearEqConCoeffs, Constraints::linearEqConTargets, Constraints::linearIneqConCoeffs, Constraints::linearIneqConLowerBnds, Constraints::linearIneqConUpperBnds, Constraints::nonlinearEqConTargets, Constraints::nonlinearIneqConLowerBnds, Constraints::nonlinearIneqConUpperBnds, Constraints::numLinearEqCons, Constraints::numLinearIneqCons, Constraints::numNonlinearEqCons, Constraints::numNonlinearIneqCons, and Constraints::sharedVarsData.

Referenced by SurrogateModel::force_rebuild(), and RecastModel::RecastModel().

43.19.3.4 void reshape (const size_t & *num_nln_ineq_cons*, const size_t & *num_nln_eq_cons*, const size_t & *num_lin_ineq_cons*, const size_t & *num_lin_eq_cons*)

reshape the linear/nonlinear constraint arrays within the Constraints hierarchy Resizes the linear and nonlinear constraint arrays at the base class. Does NOT currently resize the derived bounds arrays.

References Constraints::constraintsRep, Constraints::linearEqConTargets, Constraints::linearIneqConLowerBnds, Constraints::linearIneqConUpperBnds, Constraints::nonlinearEqConTargets, Constraints::nonlinearIneqConLowerBnds, Constraints::nonlinearIneqConUpperBnds, Constraints::numLinearEqCons, Constraints::numLinearIneqCons, Constraints::numNonlinearEqCons, Constraints::numNonlinearIneqCons, and Constraints::reshape().

43.19.3.5 void build_views () `[inline, protected]`

construct active/inactive views of all variables arrays

= EMPTY)

= EMPTY)

References Constraints::build_active_views(), Constraints::build_inactive_views(), Constraints::sharedVarsData, and SharedVariablesData::view().

Referenced by Constraints::copy(), MergedConstraints::MergedConstraints(), MixedConstraints::MixedConstraints(), MixedConstraints::reshape(), and MergedConstraints::reshape().

43.19.3.6 void manage_linear_constraints (const ProblemDescDB & *problem_db*) `[protected]`

perform checks on user input, convert linear constraint coefficient input to matrices, and assign defaults Convenience function called from derived class constructors. The number of variables active for applying linear constraints is currently defined to be the number of active continuous variables plus the number of active discrete variables (the most general case), even though very few optimizers can currently support mixed variable linear constraints.

References Dakota::abort_handler(), Constraints::continuousLowerBnds, Dakota::copy_data(), Constraints::discreteIntLowerBnds, Constraints::discreteRealLowerBnds, ProblemDescDB::get_rdv(), Constraints::linearEqConCoeffs, Constraints::linearEqConTargets, Constraints::linearIneqConCoeffs, Constraints::linearIneqConLowerBnds, Constraints::linearIneqConUpperBnds, Constraints::numLinearEqCons, and Constraints::numLinearIneqCons.

Referenced by MergedConstraints::MergedConstraints(), and MixedConstraints::MixedConstraints().

43.19.3.7 Constraints * get_constraints (const ProblemDescDB & *problem_db*, const SharedVariablesData & *svd*) `[private]`

Used only by the constructor to initialize constraintsRep to the appropriate derived type. Initializes constraintsRep to the appropriate derived type, as given by the variables view.

References SharedVariablesData::view().

Referenced by Constraints::Constraints(), and Constraints::copy().

43.19.3.8 Constraints * get_constraints (const SharedVariablesData & *svd*) const [private]

Used by copy() to initialize constraintsRep to the appropriate derived type. Initializes constraintsRep to the appropriate derived type, as given by the variables view. The default derived class constructors are invoked.

References SharedVariablesData::view().

The documentation for this class was generated from the following files:

- DakotaConstraints.H
- DakotaConstraints.C

43.20 DataFitSurrModel Class Reference

Derived model class within the surrogate model branch for managing data fit surrogates (global and local). Inheritance diagram for DataFitSurrModel::

Public Member Functions

- DataFitSurrModel (ProblemDescDB &problem_db)

  constructor

- DataFitSurrModel (Iterator &dace_iterator, Model &actual_model, const String &approx_type, const UShortArray &approx_order, short corr_type, short corr_order, short data_order, const String &point_reuse, short output_level=NORMAL_OUTPUT)

  alternate constructor for instantiations on the fly

- ~DataFitSurrModel ()

  destructor

- void total_points (int points)

  set pointsTotal and pointsManagement mode

Protected Member Functions

- void derived_compute_response (const ActiveSet &set)

  portion of compute_response() specific to DataFitSurrModel

- void derived_asynch_compute_response (const ActiveSet &set)

  portion of asynch_compute_response() specific to DataFitSurrModel

- const IntResponseMap & derived_synchronize ()

  portion of synchronize() specific to DataFitSurrModel

- const IntResponseMap & derived_synchronize_nowait ()

  portion of synchronize_nowait() specific to DataFitSurrModel

- Iterator & subordinate_iterator ()

return daceIterator

- Model & surrogate_model ()

  return this model instance

- Model & truth_model ()

  return actualModel

- void derived_subordinate_models (ModelList &ml, bool recurse_flag)

  return actualModel (and optionally its sub-models)

- void update_from_subordinate_model (bool recurse_flag=true)

  pass request to actualModel if recursing and then update from it

- Interface & interface ()

  return approxInterface

- void primary_response_fn_weights (const RealVector &wts, bool recurse_flag=true)

  set the relative weightings for multiple objective functions or least squares terms and optionally recurses into actualModel

- void surrogate_response_mode (short mode)

  set responseMode and pass any bypass request on to actualModel for any lower-level surrogates.

- void surrogate_function_indices (const IntSet &surr_fn_indices)

  (re)set the surrogate index set in SurrogateModel::surrogateFnIndices and ApproximationInterface::approxFnIndices

- void build_approximation ()

  Builds the local/multipoint/global approximation using daceIterator/actualModel to generate new data points.

- bool build_approximation (const Variables &vars, const IntResponsePair &response_pr)

  Builds the local/multipoint/global approximation using daceIterator/actualModel to generate new data points that augment the vars/response anchor point.

- void update_approximation (bool rebuild_flag)

  replaces the approximation data with daceIterator results and rebuilds the approximation if requested

- void update_approximation (const Variables &vars, const IntResponsePair &response_pr, bool rebuild_flag)

  replaces the anchor point, and rebuilds the approximation if requested

- void update_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map, bool rebuild_flag)

  replaces the current points array and rebuilds the approximation if requested

- void append_approximation (bool rebuild_flag)

appends daceIterator results to a global approximation and rebuilds it if requested

- void append_approximation (const Variables &vars, const IntResponsePair &response_pr, bool rebuild_flag)

  appends a point to a global approximation and rebuilds it if requested

- void append_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map, bool rebuild_flag)

  appends an array of points to a global approximation and rebuilds it if requested

- void pop_approximation (bool save_surr_data)

  remove approximation data added on previous append_approximation() call

- void restore_approximation ()

  restore a previous approximation data state

- bool restore_available ()

  query for whether a trial increment is restorable

- void finalize_approximation ()

  finalize data fit by applying all previous trial increments

- void store_approximation ()

  store the current data fit approximation for later combination

- void combine_approximation (short corr_type)

  combine the current data fit approximation with one previously stored

- std::vector< Approximation > & approximations ()

  retrieve the set of Approximations from approxInterface

- const RealVectorArray & approximation_coefficients ()

  return the approximation coefficients from each Approximation (request forwarded to approxInterface)

- void approximation_coefficients (const RealVectorArray &approx_coeffs)

  set the approximation coefficients within each Approximation (request forwarded to approxInterface)

- const RealVector & approximation_variances (const RealVector &c_vars)

  return the approximation variance from each Approximation (request forwarded to approxInterface)

- const Pecos::SurrogateData & approximation_data (size_t index)

  return the approximation data from a particular Approximation (request forwarded to approxInterface)

- void component_parallel_mode (short mode)

  update component parallel mode for supporting parallelism in actualModel

- void derived_init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set up actualModel for parallel operations

- void derived_init_serial ()

  set up actualModel for serial operations.

- void derived_set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set active parallel configuration within actualModel

- void derived_free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  deallocate communicator partitions for the DataFitSurrModel (request forwarded to actualModel)

- void serve ()

  Service actualModel job requests received from the master. Completes when a termination message is received from stop_servers().

- void stop_servers ()

  Executed by the master to terminate actualModel server operations when DataFitSurrModel iteration is complete.

- void inactive_view (short view, bool recurse_flag=true)

  update the Model's inactive view based on higher level (nested) context and optionally recurse into actualModel

- const String & interface_id () const

  return the approxInterface identifier

- int evaluation_id () const

  return the current evaluation id for the DataFitSurrModel

- void set_evaluation_reference ()

  set the evaluation counter reference points for the DataFitSurrModel (request forwarded to approxInterface and actualModel)

- void fine_grained_evaluation_counters ()

  request fine-grained evaluation reporting within approxInterface and actualModel

- void print_evaluation_summary (std::ostream &s, bool minimal_header=false, bool relative_count=true) const

  print the evaluation summary for the DataFitSurrModel (request forwarded to approxInterface and actualModel)

Private Member Functions

- void derived_synchronize_approx (const IntResponseMap &approx_resp_map, IntResponseMap &approx_resp_map_rekey)

  Common code for processing of approximate response maps shared by derived_synchronize() and derived_synchronize_nowait().

- void update_global ()

  Updates fit arrays for global approximations.

- void update_local_multipoint ()

  Updates fit arrays for local or multipoint approximations.

- void build_global ()

  Builds a global approximation using daceIterator.

- void build_local_multipoint ()

  Builds a local or multipoint approximation using actualModel.

- void update_actual_model ()

  update actualModel with data from current variables/labels/bounds/targets

- void update_from_actual_model ()

  update current variables/labels/bounds/targets with data from actualModel

- bool inside (const RealVector &c_vars, const IntVector &di_vars, const RealVector &dr_vars)

  test if c_vars and d_vars are within [c_l_bnds,c_u_bnds] and [d_l_bnds,d_u_bnds]

Private Attributes

- int surrModelEvalCntr

  number of calls to derived_compute_response()/ derived_asynch_compute_response()

- int pointsTotal

  total points the user specified to construct the surrogate

- short pointsManagement

  configuration for points management in build_global()

- String pointReuse

  type of point reuse for approximation builds: `all`, `region` (default if points file), or `none` (default if no points file)

- String pointReuseFile

  file name for `points_file` specification

- VariablesList reuseFileVars

  array of variables sets read from the `points_file`

- ResponseList reuseFileResponses

  array of response sets read from the `points_file`

- Interface approxInterface

  manages the building and subsequent evaluation of the approximations (required for both global and local)

- Model actualModel

  the truth model which provides evaluations for building the surrogate (optional for global, required for local)

- Iterator daceIterator

  selects parameter sets on which to evaluate actualModel in order to generate the necessary data for building global approximations (optional for global since restart data may also be used)

43.20.1 Detailed Description

Derived model class within the surrogate model branch for managing data fit surrogates (global and local). The DataFitSurrModel class manages global or local approximations (surrogates that involve data fits) that are used in place of an expensive model. The class contains an approxInterface (required for both global and local) which manages the approximate function evaluations, an actualModel (optional for global, required for local) which provides truth evaluations for building the surrogate, and a daceIterator (optional for global, not used for local) which selects parameter sets on which to evaluate actualModel in order to generate the necessary data for building global approximations.

43.20.2 Member Function Documentation

43.20.2.1 void derived_compute_response (const ActiveSet & *set*) `[protected, virtual]`

portion of compute_response() specific to DataFitSurrModel Compute the response synchronously using actualModel, approxInterface, or both (mixed case). For the approxInterface portion, build the approximation if needed, evaluate the approximate response, and apply correction (if active) to the results.

Reimplemented from Model.

References DiscrepancyCorrection::active(), Response::active_set(), DataFitSurrModel::actualModel, DiscrepancyCorrection::apply(), SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, SurrogateModel::asv_mapping(), DataFitSurrModel::build_approximation(), DataFitSurrModel::component_parallel_mode(), DiscrepancyCorrection::compute(), Model::compute_response(), Variables::continuous_variables(), Response::copy(), Model::current_response(), Model::currentResponse, Model::currentVariables, SurrogateModel::deltaCorr, SurrogateModel::force_rebuild(), Interface::map(), Model::outputLevel, ActiveSet::request_vector(), SurrogateModel::response_mapping(), SurrogateModel::responseMode, DataFitSurrModel::surrModelEvalCntr, Response::update(), and DataFitSurrModel::update_actual_model().

43.20.2.2 void derived_asynch_compute_response (const ActiveSet & *set*) `[protected, virtual]`

portion of asynch_compute_response() specific to DataFitSurrModel Compute the response asynchronously using actualModel, approxInterface, or both (mixed case). For the approxInterface portion, build the approximation if needed and evaluate the approximate response in a quasi-asynchronous approach (ApproximationInterface::map() performs the map synchronously and bookkeeps the results for return in derived_synchronize() below).

Reimplemented from Model.

References DataFitSurrModel::actualModel, SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, SurrogateModel::asv_mapping(), Model::asynch_compute_response(), DataFitSurrModel::build_approximation(), Variables::continuous_variables(), Dakota::copy_data(), Model::currentResponse, Model::currentVariables, Interface::evaluation_id(), Model::evaluation_id(), SurrogateModel::force_rebuild(), Interface::map(), SurrogateModel::rawCVarsMap, ActiveSet::request_vector(), SurrogateModel::responseMode, SurrogateModel::surrIdMap, DataFitSurrModel::surrModelEvalCntr, SurrogateModel::truthIdMap, and DataFitSurrModel::update_actual_model().

43.20.2.3 const IntResponseMap & derived_synchronize () [protected, virtual]

portion of synchronize() specific to DataFitSurrModel Blocking retrieval of asynchronous evaluations from actualModel, approxInterface, or both (mixed case). For the approxInterface portion, apply correction (if active) to each response in the array. derived_synchronize() is designed for the general case where derived_asynch_compute_response() may be inconsistent in its use of actual evaluations, approximate evaluations, or both.

Reimplemented from Model.

References DataFitSurrModel::actualModel, DataFitSurrModel::approxInterface, DataFitSurrModel::component_parallel_mode(), DiscrepancyCorrection::compute(), SurrogateModel::deltaCorr, DataFitSurrModel::derived_synchronize_approx(), Model::outputLevel, SurrogateModel::response_mapping(), SurrogateModel::responseMode, SurrogateModel::surrIdMap, SurrogateModel::surrResponseMap, Interface::synch(), Model::synchronize(), and SurrogateModel::truthIdMap.

43.20.2.4 const IntResponseMap & derived_synchronize_nowait () [protected, virtual]

portion of synchronize_nowait() specific to DataFitSurrModel Nonblocking retrieval of asynchronous evaluations from actualModel, approxInterface, or both (mixed case). For the approxInterface portion, apply correction (if active) to each response in the map. derived_synchronize_nowait() is designed for the general case where derived_asynch_compute_response() may be inconsistent in its use of actual evals, approx evals, or both.

Reimplemented from Model.

References Dakota::abort_handler(), DataFitSurrModel::actualModel, DataFitSurrModel::approxInterface, SurrogateModel::cachedApproxRespMap, DataFitSurrModel::component_parallel_mode(), DiscrepancyCorrection::compute(), SurrogateModel::deltaCorr, DataFitSurrModel::derived_synchronize_approx(), Model::outputLevel, SurrogateModel::response_mapping(), SurrogateModel::responseMode, SurrogateModel::surrIdMap, SurrogateModel::surrResponseMap, Interface::synch_nowait(), Model::synchronize_nowait(), and SurrogateModel::truthIdMap.

43.20.2.5 void build_approximation () [protected, virtual]

Builds the local/multipoint/global approximation using daceIterator/actualModel to generate new data points. This function constructs a new approximation, discarding any previous data. It constructs any required data for SurrogateData::{vars,resp}Data and does not define an anchor point for SurrogateData::anchor{Vars,Resp}.

Reimplemented from Model.

References DataFitSurrModel::actualModel, SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Interface::build_approximation(), DataFitSurrModel::build_global(),

DataFitSurrModel::build_local_multipoint(), Interface::clear_current(), Model::continuous_lower_bounds(), Constraints::continuous_lower_bounds(), Model::continuous_upper_bounds(), Constraints::continuous_upper_bounds(), Model::is_null(), Model::surrogateType, DataFitSurrModel::update_actual_model(), DataFitSurrModel::update_global(), DataFitSurrModel::update_local_multipoint(), and Model::userDefinedConstraints.

Referenced by DataFitSurrModel::derived_asynch_compute_response(), and DataFitSurrModel::derived_compute_response().

43.20.2.6 bool build_approximation (const Variables & *vars*, const IntResponsePair & *response_pr*) `[protected, virtual]`

Builds the local/multipoint/global approximation using daceIterator/actualModel to generate new data points that augment the vars/response anchor point. This function constructs a new approximation, discarding any previous data. It uses the passed data to populate SurrogateData::anchor{Vars,Resp} and constructs any required data points for SurrogateData::{vars,resp}Data.

Reimplemented from Model.

References DataFitSurrModel::actualModel, SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Interface::build_approximation(), DataFitSurrModel::build_global(), Interface::clear_current(), Model::continuous_lower_bounds(), Constraints::continuous_lower_bounds(), Model::continuous_upper_bounds(), Constraints::continuous_upper_bounds(), Model::is_null(), Model::surrogateType, DataFitSurrModel::update_actual_model(), Interface::update_approximation(), DataFitSurrModel::update_global(), DataFitSurrModel::update_local_multipoint(), and Model::userDefinedConstraints.

43.20.2.7 void update_approximation (bool *rebuild_flag*) `[protected, virtual]`

replaces the approximation data with daceIterator results and rebuilds the approximation if requested This function populates/replaces SurrogateData::anchor{Vars,Resp} and rebuilds the approximation, if requested. It does not clear other data (i.e., SurrogateData::{vars,resp}Data) and does not update the actualModel with revised bounds, labels, etc. Thus, it updates data from a previous call to build_approximation(), and is not intended to be used in isolation.

Reimplemented from Model.

References Iterator::all_responses(), Iterator::all_samples(), Iterator::all_variables(), SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Iterator::compact_mode(), DataFitSurrModel::daceIterator, Model::numFns, Interface::rebuild_approximation(), Model::surrogateType, and Interface::update_approximation().

43.20.2.8 void update_approximation (const Variables & *vars*, const IntResponsePair & *response_pr*, bool *rebuild_flag*) `[protected, virtual]`

replaces the anchor point, and rebuilds the approximation if requested This function populates/replaces SurrogateData::anchor{Vars,Resp} and rebuilds the approximation, if requested. It does not clear other data (i.e., SurrogateData::{vars,resp}Data) and does not update the actualModel with revised bounds, labels, etc. Thus, it updates data from a previous call to build_approximation(), and is not intended to be used in isolation.

Reimplemented from Model.

References SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Model::numFns, Interface::rebuild_approximation(), Model::surrogateType, and Interface::update_approximation().

43.20.2.9 void update_approximation (const VariablesArray & *vars_array*, const IntResponseMap & *resp_map*, bool *rebuild_flag*) `[protected, virtual]`

replaces the current points array and rebuilds the approximation if requested This function populates/replaces SurrogateData::{vars,resp}Data and rebuilds the approximation, if requested. It does not clear other data (i.e., SurrogateData::anchor{Vars,Resp}) and does not update the actualModel with revised bounds, labels, etc. Thus, it updates data from a previous call to build_approximation(), and is not intended to be used in isolation.

Reimplemented from Model.

References SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Model::numFns, Interface::rebuild_approximation(), Model::surrogateType, and Interface::update_approximation().

43.20.2.10 void append_approximation (bool *rebuild_flag*) `[protected, virtual]`

appends daceIterator results to a global approximation and rebuilds it if requested This function appends one point to SurrogateData::{vars,resp}Data and rebuilds the approximation, if requested. It does not modify other data (i.e., SurrogateData::anchor{Vars,Resp}) and does not update the actualModel with revised bounds, labels, etc. Thus, it appends to data from a previous call to build_approximation(), and is not intended to be used in isolation.

Reimplemented from Model.

References Iterator::all_responses(), Iterator::all_samples(), Iterator::all_variables(), Interface::append_approximation(), SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Iterator::compact_mode(), DataFitSurrModel::daceIterator, Model::numFns, Interface::rebuild_approximation(), and Model::surrogateType.

43.20.2.11 void append_approximation (const Variables & *vars*, const IntResponsePair & *response_pr*, bool *rebuild_flag*) `[protected, virtual]`

appends a point to a global approximation and rebuilds it if requested This function appends one point to SurrogateData::{vars,resp}Data and rebuilds the approximation, if requested. It does not modify other data (i.e., SurrogateData::anchor{Vars,Resp}) and does not update the actualModel with revised bounds, labels, etc. Thus, it appends to data from a previous call to build_approximation(), and is not intended to be used in isolation.

Reimplemented from Model.

References Interface::append_approximation(), SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Model::numFns, Interface::rebuild_approximation(), and Model::surrogateType.

43.20.2.12 void append_approximation (const VariablesArray & *vars_array*, const IntResponseMap & *resp_map*, bool *rebuild_flag*) `[protected, virtual]`

appends an array of points to a global approximation and rebuilds it if requested This function appends multiple points to SurrogateData::{vars,resp}Data and rebuilds the approximation, if requested. It does not modify other data (i.e., SurrogateData::anchor{Vars,Resp}) and does not update the actualModel with revised bounds, labels,

etc. Thus, it appends to data from a previous call to build_approximation(), and is not intended to be used in isolation.

Reimplemented from Model.

References Interface::append_approximation(), SurrogateModel::approxBuilds, DataFitSurrModel::approxInterface, Model::numFns, Interface::rebuild_approximation(), and Model::surrogateType.

43.20.2.13 void derived_init_communicators (const int & *max_iterator_concurrency*, bool *recurse_flag* = true) [inline, protected, virtual]

set up actualModel for parallel operations asynchronous flags need to be initialized for the sub-models. In addition, max_iterator_concurrency is the outer level iterator concurrency, not the DACE concurrency that actualModel will see, and recomputing the message_lengths on the sub-model is probably not a bad idea either. Therefore, recompute everything on actualModel using init_communicators.

Reimplemented from Model.

References DataFitSurrModel::actualModel, DataFitSurrModel::approxInterface, DataFitSurrModel::daceIterator, Model::derivative_concurrency(), Model::init_communicators(), Iterator::is_null(), Model::is_null(), Iterator::maximum_concurrency(), and Interface::minimum_points().

43.20.2.14 int evaluation_id () const [inline, protected, virtual]

return the current evaluation id for the DataFitSurrModel return the DataFitSurrModel evaluation count. Due to possibly intermittent use of surrogate bypass, this is not the same as either the approxInterface or actualModel model evaluation counts. It also does not distinguish duplicate evals.

Reimplemented from Model.

References DataFitSurrModel::surrModelEvalCntr.

43.20.2.15 void build_global () [private]

Builds a global approximation using daceIterator. Determine points to use in building the approximation and then evaluate them on actualModel using daceIterator. Any changes to the bounds should be performed by setting them at a higher level (e.g., SurrBasedOptStrategy).

References Dakota::abort_handler(), Iterator::active_set(), DataFitSurrModel::actualModel, Iterator::all_responses(), Iterator::all_samples(), Iterator::all_variables(), Interface::append_approximation(), Interface::approximation_data(), DataFitSurrModel::approxInterface, SurrogateModel::asv_mapping(), Iterator::compact_mode(), DataFitSurrModel::component_parallel_mode(), Variables::continuous_variables(), Model::currentVariables, Model::cv(), Variables::cv(), DataFitSurrModel::daceIterator, Dakota::data_pairs, Variables::discrete_int_variables(), Variables::discrete_real_variables(), Model::div(), Variables::div(), Model::drv(), Variables::drv(), DataFitSurrModel::inside(), Model::interface_id(), Iterator::is_null(), Model::is_null(), Interface::minimum_points(), Iterator::num_samples(), Model::outputLevel, DataFitSurrModel::pointReuse, DataFitSurrModel::pointReuseFile, DataFitSurrModel::pointsManagement, DataFitSurrModel::pointsTotal, Interface::recommended_points(), DataFitSurrModel::reuseFileResponses, DataFitSurrModel::reuseFileVars, Iterator::run_iterator(), Iterator::sampling_reset(), and SurrogateModel::surrogateFnIndices.

Referenced by DataFitSurrModel::build_approximation().

43.20.2.16 void build_local_multipoint () [private]

Builds a local or multipoint approximation using actualModel. Evaluate the value, gradient, and possibly Hessian needed for a local or multipoint approximation using actualModel.

References Response::active_set(), DataFitSurrModel::actualModel, DataFitSurrModel::approxInterface, SurrogateModel::asv_mapping(), String::begins(), DataFitSurrModel::component_parallel_mode(), Model::compute_response(), Model::continuous_variable_ids(), Model::current_response(), Model::current_variables(), Model::evaluation_id(), Model::hessian_type(), Model::numFns, ActiveSet::request_vector(), Model::surrogateType, and Interface::update_approximation().

Referenced by DataFitSurrModel::build_approximation().

43.20.2.17 void update_actual_model () [private]

update actualModel with data from current variables/labels/bounds/targets Update variables and constraints data within actualModel using values and labels from currentVariables and bound/linear/nonlinear constraints from userDefinedConstraints.

References Dakota::abort_handler(), DataFitSurrModel::actualModel, Model::all_continuous_lower_bounds(), Constraints::all_continuous_lower_bounds(), Model::all_continuous_upper_bounds(), Constraints::all_continuous_upper_bounds(), Model::all_continuous_variable_labels(), Variables::all_continuous_variable_labels(), Model::all_continuous_variables(), Variables::all_continuous_variables(), Model::all_discrete_int_lower_bounds(), Constraints::all_discrete_int_lower_bounds(), Model::all_discrete_int_upper_bounds(), Constraints::all_discrete_int_upper_bounds(), Model::all_discrete_int_variable_labels(), Variables::all_discrete_int_variable_labels(), Model::all_discrete_int_variables(), Variables::all_discrete_int_variables(), Model::all_discrete_real_lower_bounds(), Constraints::all_discrete_real_lower_bounds(), Model::all_discrete_real_upper_bounds(), Constraints::all_discrete_real_upper_bounds(), Model::all_discrete_real_variable_labels(), Variables::all_discrete_real_variable_labels(), Model::all_discrete_real_variables(), Variables::all_discrete_real_variables(), SurrogateModel::approxBuilds, Constraints::continuous_lower_bounds(), Model::continuous_lower_bounds(), Constraints::continuous_upper_bounds(), Model::continuous_upper_bounds(), Variables::continuous_variable_labels(), Model::continuous_variable_labels(), Variables::continuous_variables(), Model::continuous_variables(), Model::current_variables(), Model::currentResponse, Model::currentVariables, Model::cv(), Variables::cv(), Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Constraints::discrete_int_lower_bounds(), Model::discrete_int_lower_bounds(), Constraints::discrete_int_upper_bounds(), Model::discrete_int_upper_bounds(), Variables::discrete_int_variable_labels(), Model::discrete_int_variable_labels(), Variables::discrete_int_variables(), Model::discrete_int_variables(), Constraints::discrete_real_lower_bounds(), Model::discrete_real_lower_bounds(), Constraints::discrete_real_upper_bounds(), Model::discrete_real_upper_bounds(), Variables::discrete_real_variable_labels(), Model::discrete_real_variable_labels(), Variables::discrete_real_variables(), Model::discrete_real_variables(), Model::discrete_state_set_int_values(), Model::discrete_state_set_real_values(), Model::discreteDesignSetIntValues, Model::discreteDesignSetRealValues, Model::discreteStateSetIntValues, Model::discreteStateSetRealValues, Model::distParams, Model::distribution_parameters(), Model::div(), Variables::div(), Model::drv(), Variables::drv(), Response::function_labels(), Variables::inactive_continuous_variable_labels(), Model::inactive_continuous_variable_labels(), Variables::inactive_discrete_int_variable_labels(), Model::inactive_discrete_int_variable_labels(), Variables::inactive_discrete_real_variable_labels(), Model::inactive_discrete_real_variable_labels(), Model::is_null(), Constraints::linear_eq_constraint_coeffs(), Model::linear_eq_constraint_coeffs(), Constraints::linear_eq_constraint_targets(), Model::linear_eq_constraint_targets(), Constraints::linear_ineq_constraint_coeffs(), Model::linear_ineq_constraint_coeffs(), Constraints::linear_ineq_constraint_lower_bounds(), Model::linear_-

ineq_constraint_lower_bounds(), Constraints::linear_ineq_constraint_upper_bounds(), Model::linear_ineq_constraint_upper_bounds(), Constraints::nonlinear_eq_constraint_targets(), Model::nonlinear_eq_constraint_targets(), Constraints::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_lower_bounds(), Constraints::nonlinear_ineq_constraint_upper_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Constraints::num_linear_eq_constraints(), Constraints::num_linear_ineq_constraints(), Constraints::num_nonlinear_eq_constraints(), Constraints::num_nonlinear_ineq_constraints(), Model::response_labels(), Model::userDefinedConstraints, and Variables::view().

Referenced by DataFitSurrModel::build_approximation(), DataFitSurrModel::derived_asynch_compute_response(), and DataFitSurrModel::derived_compute_response().

43.20.2.18 void update_from_actual_model () [private]

update current variables/labels/bounds/targets with data from actualModel Update values and labels in currentVariables and bound/linear/nonlinear constraints in userDefinedConstraints from variables and constraints data within actualModel.

References Dakota::abort_handler(), DataFitSurrModel::actualModel, Model::all_continuous_lower_bounds(), Constraints::all_continuous_lower_bounds(), Model::all_continuous_upper_bounds(), Constraints::all_continuous_upper_bounds(), Model::all_continuous_variable_labels(), Variables::all_continuous_variable_labels(), Model::all_continuous_variables(), Variables::all_continuous_variables(), Model::all_discrete_int_lower_bounds(), Constraints::all_discrete_int_lower_bounds(), Model::all_discrete_int_upper_bounds(), Constraints::all_discrete_int_upper_bounds(), Model::all_discrete_int_variable_labels(), Variables::all_discrete_int_variable_labels(), Model::all_discrete_int_variables(), Variables::all_discrete_int_variables(), Model::all_discrete_real_lower_bounds(), Constraints::all_discrete_real_lower_bounds(), Model::all_discrete_real_upper_bounds(), Constraints::all_discrete_real_upper_bounds(), Model::all_discrete_real_variable_labels(), Variables::all_discrete_real_variable_labels(), Model::all_discrete_real_variables(), Variables::all_discrete_real_variables(), SurrogateModel::approxBuilds, Model::currentResponse, Model::currentVariables, Variables::cv(), Model::cv(), Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Model::discrete_state_set_int_values(), Model::discrete_state_set_real_values(), Model::discreteDesignSetIntValues, Model::discreteDesignSetRealValues, Model::discreteStateSetIntValues, Model::discreteStateSetRealValues, Model::distParams, Model::distribution_parameters(), Variables::div(), Model::div(), Variables::drv(), Model::drv(), Response::function_labels(), Model::linear_eq_constraint_coeffs(), Constraints::linear_eq_constraint_coeffs(), Model::linear_eq_constraint_targets(), Constraints::linear_eq_constraint_targets(), Model::linear_ineq_constraint_coeffs(), Constraints::linear_ineq_constraint_coeffs(), Model::linear_ineq_constraint_lower_bounds(), Constraints::linear_ineq_constraint_lower_bounds(), Model::linear_ineq_constraint_upper_bounds(), Constraints::linear_ineq_constraint_upper_bounds(), Model::nonlinear_eq_constraint_targets(), Constraints::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Constraints::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Constraints::nonlinear_ineq_constraint_upper_bounds(), Model::num_linear_eq_constraints(), Model::num_linear_ineq_constraints(), Model::num_nonlinear_eq_constraints(), Model::num_nonlinear_ineq_constraints(), Model::response_labels(), and Model::userDefinedConstraints.

Referenced by DataFitSurrModel::DataFitSurrModel(), and DataFitSurrModel::update_from_subordinate_model().

43.20.3 Member Data Documentation

43.20.3.1 Model actualModel [private]

the truth model which provides evaluations for building the surrogate (optional for global, required for local) actualModel is unrestricted in type; arbitrary nestings are possible.

Referenced by DataFitSurrModel::build_approximation(), DataFitSurrModel::build_global(), DataFitSurrModel::build_local_multipoint(), DataFitSurrModel::component_parallel_mode(), DataFitSurrModel::DataFitSurrModel(), DataFitSurrModel::derived_asynch_compute_response(), DataFitSurrModel::derived_compute_response(), DataFitSurrModel::derived_free_communicators(), DataFitSurrModel::derived_init_communicators(), DataFitSurrModel::derived_init_serial(), DataFitSurrModel::derived_set_communicators(), DataFitSurrModel::derived_subordinate_models(), DataFitSurrModel::derived_synchronize(), DataFitSurrModel::derived_synchronize_nowait(), DataFitSurrModel::fine_grained_evaluation_counters(), DataFitSurrModel::inactive_view(), DataFitSurrModel::inside(), DataFitSurrModel::primary_response_fn_weights(), DataFitSurrModel::print_evaluation_summary(), DataFitSurrModel::serve(), DataFitSurrModel::stop_servers(), DataFitSurrModel::surrogate_response_mode(), DataFitSurrModel::truth_model(), DataFitSurrModel::update_actual_model(), DataFitSurrModel::update_from_actual_model(), DataFitSurrModel::update_from_subordinate_model(), DataFitSurrModel::update_global(), and DataFitSurrModel::update_local_multipoint().

The documentation for this class was generated from the following files:

- DataFitSurrModel.H
- DataFitSurrModel.C

43.21 DataInterface Class Reference

Handle class for interface specification data.

Public Member Functions

- DataInterface ()

  constructor

- DataInterface (const DataInterface &)

  copy constructor

- ~DataInterface ()

  destructor

- DataInterface & operator= (const DataInterface &)

  assignment operator

- void write (std::ostream &s) const

  write a DataInterface object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataInterface object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataInterface object to a packed MPI buffer

Static Public Member Functions

- static bool id_compare (const DataInterface &di, const std::string &id)

  compares the idInterface attribute of DataInterface objects

Private Attributes

- DataInterfaceRep * dataIfaceRep

  pointer to the body (handle-body idiom)

Friends

- class **ProblemDescDB**
- class **NIDRProblemDescDB**

- void run_dakota_data ()

  library_mode default data initializer

43.21.1 Detailed Description

Handle class for interface specification data. The DataInterface class is used to provide a memory management handle for the data in DataInterfaceRep. It is populated by IDRProblemDescDB::interface_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataInterface objects is maintained in ProblemDescDB::dataInterfaceList, one for each interface specification in an input file.

The documentation for this class was generated from the following files:

- DataInterface.H
- DataInterface.C

43.22 DataMethod Class Reference

Handle class for method specification data.

Public Member Functions

- DataMethod ()

  constructor

- DataMethod (const DataMethod &)

  copy constructor

- ~DataMethod ()

  destructor

- DataMethod & operator= (const DataMethod &)

  assignment operator

- void write (std::ostream &s) const

  write a DataMethod object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataMethod object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataMethod object to a packed MPI buffer

Static Public Member Functions

- static bool id_compare (const DataMethod &dm, const std::string &id)

  compares the idMethod attribute of DataMethod objects

Private Attributes

- DataMethodRep * dataMethodRep

  pointer to the body (handle-body idiom)

Friends

- class **ProblemDescDB**
- class **NIDRProblemDescDB**

- void run_dakota_data ()

  library_mode default data initializer

43.22.1 Detailed Description

Handle class for method specification data. The DataMethod class is used to provide a memory management handle for the data in DataMethodRep. It is populated by IDRProblemDescDB::method_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataMethod objects is maintained in ProblemDescDB::dataMethodList, one for each method specification in an input file.

The documentation for this class was generated from the following files:

- DataMethod.H
- DataMethod.C

43.23 DataMethodRep Class Reference

Body class for method specification data.

Public Attributes

- String idMethod

  string identifier for the method specification data set (from the `id_method` *specification in* **MethodIndControl***)*

- String modelPointer

  string pointer to the model specification to be used by this method (from the `model_pointer` *specification in* **MethodIndControl***)*

- short methodOutput

  method verbosity control: {SILENT,QUIET,NORMAL,VERBOSE,DEBUG}_OUTPUT (from the `output` *specification in* **MethodIndControl***)*

- int maxIterations

  maximum number of iterations allowed for the method (from the `max_iterations` *specification in* **MethodIndControl***)*

- int maxFunctionEvaluations

  maximum number of function evaluations allowed for the method (from the `max_function_evaluations` *specification in* **MethodIndControl***)*

- bool speculativeFlag

  flag for use of speculative gradient approaches for maintaining parallel load balance during the line search portion of optimization algorithms (from the `speculative` *specification in* **MethodIndControl***)*

- bool methodUseDerivsFlag

  flag for usage of derivative data to enhance the computation of surrogate models (PCE/SC expansions, GP models for EGO/EGRA/EGIE) based on the `use_derivatives` *specification*

- Real convergenceTolerance

  iteration convergence tolerance for the method (from the `convergence_tolerance` *specification in* **MethodIndControl***)*

- Real constraintTolerance

  tolerance for controlling the amount of infeasibility that is allowed before an active constraint is considered to be violated (from the `constraint_tolerance` *specification in* **MethodIndControl***)*

- bool methodScaling

  flag indicating scaling status (from the `scaling` *specification in* **MethodIndControl***)*

- size_t numFinalSolutions

  number of final solutions returned from the iterator

- RealVector linearIneqConstraintCoeffs

  coefficient matrix for the linear inequality constraints (from the `linear_inequality_constraint_matrix` *specification in* **MethodIndControl***)*

- RealVector linearIneqLowerBnds

  lower bounds for the linear inequality constraints (from the `linear_inequality_lower_bounds` *specification in* **MethodIndControl***)*

- RealVector linearIneqUpperBnds

  upper bounds for the linear inequality constraints (from the `linear_inequality_upper_bounds` *specification in* **MethodIndControl***)*

- StringArray linearIneqScaleTypes

  scaling types for the linear inequality constraints (from the `linear_inequality_scale_types` *specification in* **MethodIndControl***)*

- RealVector linearIneqScales

  scaling factors for the linear inequality constraints (from the `linear_inequality_scales` *specification in* **MethodIndControl***)*

- RealVector linearEqConstraintCoeffs

  coefficient matrix for the linear equality constraints (from the `linear_equality_constraint_matrix` *specification in* **MethodIndControl***)*

- RealVector linearEqTargets

  targets for the linear equality constraints (from the `linear_equality_targets` *specification in* **MethodIndControl***)*

- StringArray linearEqScaleTypes

  scaling types for the linear equality constraints (from the `linear_equality_scale_types` *specification in* **MethodIndControl***)*

- RealVector linearEqScales

  scaling factors for the linear equality constraints (from the `linear_equality_scales` *specification in* **MethodIndControl***)*

- String methodName

  the method selection: one of the optimizer, least squares, nond, dace, or parameter study methods

- String subMethodName

  string identifier for a sub-method within a multi-option method specification (e.g., from `sub_method_name` *in SBL/SBG,* `dace` *option, or* `richardson_extrap` *option)*

- String subMethodPointer

  string pointer for a sub-method specification used by a multi-component method (from the `sub_method_pointer` *specification in SBL/SBG)*

- int surrBasedLocalSoftConvLimit

number of consecutive iterations with change less than convergenceTolerance required to trigger convergence within the surrogate-based local method (from the `soft_convergence_limit` *specification in* **MethodSBL***)*

- bool surrBasedLocalLayerBypass

  flag to indicate user-specification of a bypass of any/all layerings in evaluating truth response values in SBL.

- Real surrBasedLocalTRInitSize

  initial trust region size in the surrogate-based local method (from the `initial_size` *specification in* **MethodSBL***) note: this is a relative value, e.g., 0.1 = 10% of global bounds distance (upper bound - lower bound) for each variable*

- Real surrBasedLocalTRMinSize

  minimum trust region size in the surrogate-based local method (from the `minimum_size` *specification in* **MethodSBL***), if the trust region size falls below this threshold the SBL iterations are terminated (note: if kriging is used with SBL, the min trust region size is set to 1.0e-3 in attempt to avoid ill-conditioned matrixes that arise in kriging over small trust regions)*

- Real surrBasedLocalTRContractTrigger

  trust region minimum improvement level (ratio of actual to predicted decrease in objective fcn) in the surrogate-based local method (from the `contract_threshold` *specification in* **MethodSBL***), the trust region shrinks or is rejected if the ratio is below this value ("eta_1" in the Conn-Gould-Toint trust region book)*

- Real surrBasedLocalTRExpandTrigger

  trust region sufficient improvement level (ratio of actual to predicted decrease in objective fn) in the surrogate-based local method (from the `expand_threshold` *specification in* **MethodSBL***), the trust region expands if the ratio is above this value ("eta_2" in the Conn-Gould-Toint trust region book)*

- Real surrBasedLocalTRContract

  trust region contraction factor in the surrogate-based local method (from the `contraction_factor` *specification in* **MethodSBL***)*

- Real surrBasedLocalTRExpand

  trust region expansion factor in the surrogate-based local method (from the `expansion_factor` *specification in* **MethodSBL***)*

- short surrBasedLocalSubProbObj

  SBL approximate subproblem objective: ORIGINAL_PRIMARY, SINGLE_OBJECTIVE, LAGRANGIAN_OBJECTIVE, or AUGMENTED_LAGRANGIAN_OBJECTIVE.

- short surrBasedLocalSubProbCon

  SBL approximate subproblem constraints: NO_CONSTRAINTS, LINEARIZED_CONSTRAINTS, or ORIGINAL_CONSTRAINTS.

- short surrBasedLocalMeritFn

  SBL merit function type: BASIC_PENALTY, ADAPTIVE_PENALTY, BASIC_LAGRANGIAN, or AUGMENTED_LAGRANGIAN.

- short surrBasedLocalAcceptLogic

SBL iterate acceptance logic: TR_RATIO or FILTER.

- short surrBasedLocalConstrRelax

  SBL constraint relaxation method: NO_RELAX or HOMOTOPY.

- bool surrBasedGlobalReplacePts

  user-specified method for adding points to the set upon which the next surrogate is based in the `surrogate_based_global` *strategy.*

- String minMaxType

  the `optimization_type` *specification in* **MethodDOTDC**

- String dlDetails

  string of options for a dynamically linked solver

- void * dlLib

  handle to dynamically loaded library

- int verifyLevel

  the `verify_level` *specification in* **MethodNPSOLDC**

- Real functionPrecision

  the `function_precision` *specification in* **MethodNPSOLDC**

- Real lineSearchTolerance

  the `linesearch_tolerance` *specification in* **MethodNPSOLDC**

- Real absConvTol

  absolute function convergence tolerance

- Real xConvTol

  x-convergence tolerance

- Real singConvTol

  singular convergence tolerance

- Real singRadius

  radius for singular convergence test

- Real falseConvTol

  false-convergence tolerance

- Real initTRRadius

  initial trust radius

- int covarianceType

kind of covariance required

- bool regressDiag

  whether to print the regression diagnostic vector

- String searchMethod

  the `search_method` *specification for Newton and nonlinear interior-point methods in* **MethodOPTPPDC**

- Real gradientTolerance

  the `gradient_tolerance` *specification in* **MethodOPTPPDC**

- Real maxStep

  the `max_step` *specification in* **MethodOPTPPDC**

- String meritFn

  the `merit_function` *specification for nonlinear interior-point methods in* **MethodOPTPPDC**

- String centralPath

  the `central_path` *specification for nonlinear interior-point methods in* **MethodOPTPPDC**

- Real stepLenToBoundary

  the `steplength_to_boundary` *specification for nonlinear interior-point methods in* **MethodOPTPPDC**

- Real centeringParam

  the `centering_parameter` *specification for nonlinear interior-point methods in* **MethodOPTPPDC**

- int searchSchemeSize

  the `search_scheme_size` *specification for PDS methods in* **MethodOPTPPDC**

- Real initStepLength

  the `initStepLength` *choice for nonlinearly constrained APPS in* **MethodAPPSDC**

- Real contractStepLength

  the `contractStepLength` *choice for nonlinearly constrained APPS in* **MethodAPPSDC**

- Real threshStepLength

  the `threshStepLength` *choice for nonlinearly constrained APPS in* **MethodAPPSDC**

- String evalSynchronize

  the `synchronization` *choice for nonlinearly constrained APPS in* **MethodAPPSDC**

- String meritFunction

  the `meritFunction` *choice for nonlinearly constrained APPS in* **MethodAPPSDC**

- Real constrPenalty

  the `constrPenalty` *choice for nonlinearly constrained APPS in* **MethodAPPSDC**

- Real smoothFactor

  the initial `smoothFactor` *value for nonlinearly constrained APPS in* **MethodAPPSDC**

- String evalSynchronization

  the `synchronization` *setting for parallel pattern search methods in MethodCOLINYPS and* **MethodAPPS**

- Real constraintPenalty

  the initial `constraint_penalty` *for COLINY methods in* **MethodAPPS**, *MethodCOLINYDIR, MethodCOLINYPS, MethodCOLINYSW and MethodCOLINYEA*

- bool constantPenalty

  the `constant_penalty` *flag for COLINY methods in MethodCOLINYPS and MethodCOLINYSW*

- Real globalBalanceParam

  the `global_balance_parameter` *for the DIRECT method in MethodCOLINYDIR*

- Real localBalanceParam

  the `local_balance_parameter` *for the DIRECT method in MethodCOLINYDIR*

- Real maxBoxSize

  the `max_boxsize_limit` *for the DIRECT method in MethodCOLINYDIR*

- Real minBoxSize

  the `min_boxsize_limit` *for the DIRECT method in MethodCOLINYDIR and* **MethodNCSUDC**

- String boxDivision

  the `division` *setting (*`major_dimension` *or* `all_dimensions`*) for the DIRECT method in MethodCOLINYDIR*

- bool mutationAdaptive

  the `non_adaptive` *specification for the coliny_ea method in MethodCOLINYEA*

- bool showMiscOptions

  the `show_misc_options` *specification in MethodCOLINYDC*

- StringArray miscOptions

  the `misc_options` *specification in MethodCOLINYDC*

- Real solnTarget

  the `solution_target` *specification in MethodCOLINYDC*

- Real crossoverRate

  the `crossover_rate` *specification for EA methods in MethodCOLINYEA*

- Real mutationRate

the `mutation_rate` *specification for EA methods in MethodCOLINYEA*

- Real mutationScale

  the `mutation_scale` *specification for EA methods in MethodCOLINYEA*

- Real mutationMinScale

  the `min_scale` *specification for mutation in EA methods in MethodCOLINYEA*

- Real initDelta

  the `initial_delta` *specification for APPS/COBYLA/PS/SW methods in* **MethodAPPS**, *MethodCOLINYCOB, MethodCOLINYPS, and MethodCOLINYSW*

- Real threshDelta

  the `threshold_delta` *specification for APPS/COBYLA/PS/SW methods in* **MethodAPPS**, *MethodCOLINYCOB, MethodCOLINYPS, and MethodCOLINYSW*

- Real contractFactor

  the `contraction_factor` *specification for APPS/PS/SW methods in* **MethodAPPS**, *MethodCOLINYPS, and MethodCOLINYSW*

- int newSolnsGenerated

  the `new_solutions_generated` *specification for GA/EPSA methods in MethodCOLINYEA*

- int numberRetained

  the integer assignment to random, chc, or elitist in the `replacement_type` *specification for GA/EPSA methods in MethodCOLINYEA*

- bool expansionFlag

  the `no_expansion` *specification for APPS/PS/SW methods in* **MethodAPPS**, *MethodCOLINYPS, and MethodCOLINYSW*

- int expandAfterSuccess

  the `expand_after_success` *specification for PS/SW methods in MethodCOLINYPS and MethodCOLINYSW*

- int contractAfterFail

  the `contract_after_failure` *specification for the SW method in MethodCOLINYSW*

- int mutationRange

  the `mutation_range` *specification for the pga_int method in MethodCOLINYEA*

- int totalPatternSize

  the `total_pattern_size` *specification for PS methods in MethodCOLINYPS*

- bool randomizeOrderFlag

  the `stochastic` *specification for the PS method in MethodCOLINYPS*

- String selectionPressure

the `fitness_type` *specification for EA methods in MethodCOLINYEA*

- String replacementType

  the `replacement_type` *specification for EA methods in MethodCOLINYEA*

- String crossoverType

  the `crossover_type` *specification for EA methods in MethodCOLINYEA*

- String mutationType

  the `mutation_type` *specification for EA methods in MethodCOLINYEA*

- String exploratoryMoves

  the `exploratory_moves` *specification for the PS method in MethodCOLINYPS*

- String patternBasis

  the `pattern_basis` *specification for APPS/PS methods in* **MethodAPPS** *and MethodCOLINYPS*

- size_t numCrossPoints

  The number of crossover points or multi-point schemes.

- size_t numParents

  The number of parents to use in a crossover operation.

- size_t numOffspring

  The number of children to produce in a crossover operation.

- String fitnessType

  the fitness assessment operator to use.

- String convergenceType

  The means by which this JEGA should converge.

- Real percentChange

  The minimum percent change before convergence for a fitness tracker converger.

- size_t numGenerations

  The number of generations over which a fitness tracker converger should track.

- Real fitnessLimit

  The cutoff value for survival in fitness limiting selectors (e.g., below_limit selector).

- Real shrinkagePercent

  The minimum percentage of the requested number of selections that must take place on each call to the selector (0, 1).

- String nichingType

The niching type.

- RealVector nicheVector

  The discretization percentage along each objective.

- String postProcessorType

  The post processor type.

- RealVector distanceVector

  The discretization percentage along each objective.

- String initializationType

  The means by which the JEGA should initialize the population.

- String flatFile

  The filename to use for initialization.

- String logFile

  The filename to use for logging.

- int populationSize

  the `population_size` *specification for GA methods in MethodCOLINYEA*

- bool printPopFlag

  The `print_each_pop` *flag to set the printing of the population at each generation.*

- Real volBoxSize

  the `volume_boxsize_limit` *for the DIRECT method in* **MethodNCSUDC**

- int numSymbols

  the `symbols` *specification for DACE methods*

- bool mainEffectsFlag

  the `main_effects` *specification for sampling methods in* **MethodDDACE***)*

- bool latinizeFlag

  the `latinize` *specification for FSU QMC and CVT methods in* **MethodFSUDACE**

- bool volQualityFlag

  the `quality_metrics` *specification for sampling methods (FSU QMC and CVT methods in* **MethodFSUDACE***)*

- IntVector sequenceStart

  the `sequenceStart` *specification in* **MethodFSUDACE**

- IntVector sequenceLeap

the `sequenceLeap` *specification in* **MethodFSUDACE**

- IntVector primeBase

  the `primeBase` *specification in* **MethodFSUDACE**

- int numTrials

  the `numTrials` *specification in* **MethodFSUDACE**

- String trialType

  the `trial_type` *specification in* **MethodFSUDACE**

- int randomSeed

  the `seed` *specification for COLINY, NonD, & DACE methods*

- int numSamples

  the `samples` *specification for NonD & DACE methods*

- bool fixedSeedFlag

  flag for fixing the value of the seed among different NonD/DACE sample sets. This results in the use of the same sampling stencil/pattern throughout a strategy with repeated sampling.

- bool fixedSequenceFlag

  flag for fixing the sequence for Halton or Hammersley QMC sample sets. This results in the use of the same sampling stencil/pattern throughout a strategy with repeated sampling.

- int previousSamples

  the number of previous samples when augmenting a LHS sample

- bool vbdFlag

  the `var_based_decomp` *specification for a variety of sampling methods*

- Real vbdDropTolerance

  the `var_based_decomp` *tolerance for omitting index output*

- short vbdControl

  a sub-specification of vbdFlag: {NO,UNIVARIATE,ALL}_VBD. When vbdFlag is on, controls granularity of calculation/output of main/interaction/total effects

- String rngName

  the `basic` *random-number generator for NonD*

- short refinementType

  refinement type for stochastic expansions from dimension refinement keyword group

- short refinementControl

  refinement control for stochastic expansions from dimension refinement keyword group

- short nestingOverride

  override for default point nesting policy: NO_NESTING_OVERRIDE, NESTED, or NON_NESTED

- short growthOverride

  override for default point growth restriction policy: NO_GROWTH_OVERRIDE, RESTRICTED, or UNRESTRICTED

- short expansionType

  enumeration for u-space type that defines u-space variable targets for probability space transformations: EXTENDED_U (default), ASKEY_U, STD_NORMAL_U, or STD_UNIFORM_U

- bool piecewiseBasis

  boolean indicating presence of `piecewise` *keyword*

- short piecewiseBasisType

  enumeration for type of basis in piecewise interpolants: NODAL_INTERPOLANT or HIERARCHICAL_INTERPOLANT

- int expansionTerms

  the `expansion_terms` *specification in* **MethodNonDPCE**

- UShortArray expansionOrder

  the `expansion_order` *specification in* **MethodNonDPCE**

- int expansionSamples

  the `expansion_samples` *specification in* **MethodNonDPCE**

- String expansionSampleType

  allows for incremental PCE construction using the `incremental_lhs` *specification in* **MethodNonDPCE**

- UShortArray quadratureOrder

  the `quadrature_order` *specification in* **MethodNonDPCE** *and* **MethodNonDSC**

- UShortArray sparseGridLevel

  the `sparse_grid_level` *specification in* **MethodNonDPCE**, **MethodNonDSC**, *and other stochastic expansion-enabled methods*

- RealVector anisoGridDimPref

  the `dimension_preference` *specification for tensor and sparse grids in* **MethodNonDPCE** *and* **MethodNonDSC**

- unsigned short cubIntOrder

  the `cubature_integrand` *specification in* **MethodNonDPCE**

- int collocationPoints

  the `collocation_points` *specification in* **MethodNonDPCE**

- Real collocationRatio

  the `collocation_ratio` *specification in* **MethodNonDPCE**

- Real collocRatioTermsOrder

  order applied to the number of expansion terms when applying or computing the collocation ratio within regression PCE; based on the `ratio_order` *specification in* **MethodNonDPCE**

- String collocPtReuse

  allows for incremental PCE construction using the `reuse_points` *specification in* **MethodNonDPCE**

- bool probCollocFlag

  flag for usage of a filtered set of tensor-product grid points within regression PCE; based on the `tensor_grid` *specification in* **MethodNonDPCE**

- String expansionImportFile

  the `expansion_import_file` *specification in* **MethodNonDPCE**

- String sampleType

  the `sample_type` *specification in* **MethodNonDMC**, **MethodNonDPCE**, *and* **MethodNonDSC**

- String reliabilitySearchType

  the type of limit state search in **MethodNonDLocalRel** *(*`x_taylor_mean`*,* `x_taylor_mpp`*,* `x_two_point`*,* `u_taylor_mean`*,* `u_taylor_mpp`*,* `u_two_point`*, or* `no_approx`*) or* **MethodNonDGlobalRel** *(*`x_gaussian_process` *or* `u_gaussian_process`*)*

- String reliabilityIntegration

  the `first_order` *or* `second_order` *integration selection in* **MethodNonDLocalRel**

- String integrationRefine

  the `import`*,* `adapt_import`*, or* `mm_adapt_import` *integration refinement selection in* **MethodNonDLocalRel**, **MethodNonDPCE**, *and* **MethodNonDSC**

- String nondOptAlgorithm

  the algorithm selection `sqp` *or* `nip` *used for computing the MPP in* **MethodNonDLocalRel** *or the interval in* **MethodNonDLocalIntervalEst**

- String distributionType

  the `distribution` `cumulative` *or* `complementary` *specification in* **MethodNonDMC**, **MethodNonDPCE**, **MethodNonDLocalRel**, *and* **MethodNonDGlobalRel**

- String responseLevelMappingType

  the `compute` `probabilities`*,* `reliabilities`*, or* `gen_reliabilities` *specification in* **MethodNonDMC**, **MethodNonDPCE**, **MethodNonDLocalRel**, *and* **MethodNonDGlobalRel**

- RealVectorArray responseLevels

  the `response_levels` *specification in* **MethodNonDMC**, **MethodNonDPCE**, **MethodNonDLocalRel**, *and* **MethodNonDGlobalRel**

- RealVectorArray probabilityLevels

  the `probability_levels` *specification in* **MethodNonDMC**, **MethodNonDPCE**, **MethodNonDLocalRel**, *and* **MethodNonDGlobalRel**

- RealVectorArray reliabilityLevels

  the `reliability_levels` *specification in* **MethodNonDMC**, **MethodNonDPCE**, *and* **MethodNonDLocalRel**

- RealVectorArray genReliabilityLevels

  the `gen_reliability_levels` *specification in* **MethodNonDMC**, **MethodNonDPCE**, **MethodNonDLocalRel**, *and* **MethodNonDGlobalRel**

- bool allVarsFlag

  the `all_variables` *specification in* **MethodNonDMC**

- int emulatorSamples

  *the number of samples to construct a GP emulator for Bayesian calibration methods (***MethodNonDBayesCalib***)*

- short emulatorType

  the `emulator` *specification in* **MethodNonDBayesCalib**

- String rejectionType

  the `rejection` *type specification in* **MethodNonDBayesCalib**

- String metropolisType

  the `metropolis` *type specification in* **MethodNonDBayesCalib**

- Real proposalCovScale

  the `proposal` *covariance scale factor in* **MethodNonDBayesCalib**

- Real likelihoodScale

  the `likelihood` *scale factor in* **MethodNonDBayesCalib**

- RealVector finalPoint

  the `final_point` *specification in* **MethodPSVPS**

- RealVector stepVector

  the `step_vector` *specification in* **MethodPSVPS** *and* **MethodPSCPS**

- int numSteps

  the `num_steps` *specification in* **MethodPSVPS**

- IntVector stepsPerVariable

  the `deltas_per_variable` *specification in* **MethodPSCPS**

- RealVector listOfPoints

  the `list_of_points` specification in **MethodPSLPS**

- UShortArray varPartitions

  the `partitions` specification for PStudy method in **MethodPSMPS**

- Real refinementRate

  rate of mesh refinement in Richardson extrapolation

Private Member Functions

- DataMethodRep ()

  constructor

- ~DataMethodRep ()

  destructor

- void write (std::ostream &s) const

  write a DataInterfaceRep object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataInterfaceRep object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataInterfaceRep object to a packed MPI buffer

Private Attributes

- int referenceCount

  number of handle objects sharing this dataMethodRep

Friends

- class DataMethod

  the handle class can access attributes of the body class directly

43.23.1 Detailed Description

Body class for method specification data. The DataMethodRep class is used to contain the data from a method keyword specification. Default values are managed in the DataMethodRep constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::dataMethodList is private.

The documentation for this class was generated from the following files:

- DataMethod.H
- DataMethod.C

43.24 DataModel Class Reference

Handle class for model specification data.

Public Member Functions

- DataModel ()

  constructor

- DataModel (const DataModel &)

  copy constructor

- ~DataModel ()

  destructor

- DataModel & operator= (const DataModel &)

  assignment operator

- void write (std::ostream &s) const

  write a DataModel object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataModel object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataModel object to a packed MPI buffer

Static Public Member Functions

- static bool id_compare (const DataModel &dm, const std::string &id)

  compares the idModel attribute of DataModel objects

Private Attributes

- DataModelRep * dataModelRep

  pointer to the body (handle-body idiom)

Friends

- class **ProblemDescDB**
- class **NIDRProblemDescDB**

43.24.1 Detailed Description

Handle class for model specification data. The DataModel class is used to provide a memory management handle for the data in DataModelRep. It is populated by IDRProblemDescDB::model_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataModel objects is maintained in ProblemDescDB::dataModelList, one for each model specification in an input file.

The documentation for this class was generated from the following files:

- DataModel.H
- DataModel.C

43.25 DataModelRep Class Reference

Body class for model specification data.

Public Attributes

- String idModel

  string identifier for the model specification data set (from the `id_model` *specification in* **ModelIndControl***)*

- String modelType

  model type selection: single, surrogate, or nested (from the model type specification in **ModelIndControl***)*

- String variablesPointer

  string pointer to the variables specification to be used by this model (from the `variables_pointer` *specification in* **ModelIndControl***)*

- String interfacePointer

  string pointer to the interface specification to be used by this model (from the `interface_pointer` *specification in* **ModelSingle** *and the* `optional_interface_pointer` *specification in* **ModelNested***)*

- String responsesPointer

  string pointer to the responses specification to be used by this model (from the `responses_pointer` *specification in* **ModelIndControl***)*

- String subMethodPointer

  pointer to a sub-iterator used for global approximations (from the `dace_method_pointer` *specification in* **ModelSurrG***) or by nested models (from the* `sub_method_pointer` *specification in* **ModelNested***)*

- IntSet surrogateFnIndices

  array specifying the response function set that is approximated

- String surrogateType

  the selected surrogate type: local_taylor, multipoint_tana, global_(neural_network,mars,orthogonal_-polynomial,gaussian, polynomial,kriging), or hierarchical

- String truthModelPointer

  pointer to the model specification for constructing the truth model used in building local, multipoint, and hierarchical approximations (from the `actual_model_pointer` *specification in* **ModelSurrL** *and* **ModelSurrMP** *and the* `high_fidelity_model_pointer` *specification in* **ModelSurrH***)*

- String lowFidelityModelPointer

  pointer to the low fidelity model specification used in hierarchical approximations (from the `low_fidelity_-model_pointer` *specification in* **ModelSurrH***)*

- int pointsTotal

  user-specified lower bound on total points with which to build the model (if reuse_points < pointsTotal, new samples will make up the difference)

- short pointsManagement

  points management configuration for DataFitSurrModel: DEFAULT_POINTS, MINIMUM_POINTS, or RECOMMENDED_POINTS

- String approxPointReuse

  sample reuse selection for building global approximations: none, all, region, or file (from the `reuse_samples` *specification in* **ModelSurrG***)*

- String approxPointReuseFile

  the file name for the "file" setting for the `reuse_samples` *specification in* **ModelSurrG**

- bool approxPointFileAnnotated

  whether the point reuse file is annotated (default true)

- short approxCorrectionType

  correction type for global and hierarchical approximations: NO_CORRECTION, ADDITIVE_CORRECTION, MULTIPLICATIVE_CORRECTION, or COMBINED_CORRECTION (from the `correction` *specification in* **ModelSurrG** *and* **ModelSurrH***)*

- short approxCorrectionOrder

  correction order for global and hierarchical approximations: 0, 1, or 2 (from the `correction` *specification in* **ModelSurrG** *and* **ModelSurrH***)*

- bool modelUseDerivsFlag

  flags the use of derivatives in building global approximations (from the `use_derivatives` *specification in* **ModelSurrG***)*

- short polynomialOrder

  scalar integer indicating the order of the polynomial approximation (1=linear, 2=quadratic, 3=cubic; from the `polynomial` *specification in* **ModelSurrG***)*

- RealVector krigingCorrelations

  vector of correlations used in building a kriging approximation (from the `correlations` *specification in* **ModelSurrG***)*

- String krigingOptMethod

  optimization method to use in finding optimal correlation parameters: none, sampling, local, global

- short krigingMaxTrials

  maximum number of trials in optimization of kriging correlations

- RealVector krigingMaxCorrelations

  upper bound on kriging correlation vector

- RealVector krigingMinCorrelations

  lower bound on kriging correlation vector

- short mlsPolyOrder

  polynomial order for moving least squares approximation

- short mlsWeightFunction

  weight function for moving least squares approximation

- short rbfBases

  bases for radial basis function approximation

- short rbfMaxPts

  maximum number of points for radial basis function approximation

- short rbfMaxSubsets

  maximum number of subsets for radial basis function approximation

- short rbfMinPartition

  minimum partition for radial basis function approximation

- short marsMaxBases

  maximum number of bases for MARS approximation

- String marsInterpolation

  interpolation type for MARS approximation

- short annRandomWeight

  random weight for artificial neural network approximation

- short annNodes

  number of nodes for artificial neural network approximation

- Real annRange

  range for artificial neural network approximation

- String trendOrder

  scalar integer indicating the order of the Gaussian process mean (0= constant, 1=linear, 2=quadratic, 3=cubic); from the `gaussian_process` *specification in* **ModelSurrG***)*

- bool pointSelection

  flag indicating the use of point selection in the Gaussian process

- StringArray diagMetrics

  List of diagnostic metrics the user requests to assess the goodness of fit for a surrogate model.

- String optionalInterfRespPointer

string pointer to the responses specification used by the optional interface in nested models (from the `optional_interface_responses_pointer` *specification in* **ModelNested***)*

- StringArray primaryVarMaps

  the primary variable mappings used in nested models for identifying the lower level variable targets for inserting top level variable values (from the `primary_variable_mapping` *specification in* **ModelNested***)*

- StringArray secondaryVarMaps

  the secondary variable mappings used in nested models for identifying the (distribution) parameter targets within the lower level variables for inserting top level variable values (from the `secondary_variable_mapping` *specification in* **ModelNested***)*

- RealVector primaryRespCoeffs

  the primary response mapping matrix used in nested models for weighting contributions from the sub-iterator responses in the top level (objective) functions (from the `primary_response_mapping` *specification in* **ModelNested***)*

- RealVector secondaryRespCoeffs

  the secondary response mapping matrix used in nested models for weighting contributions from the sub-iterator responses in the top level (constraint) functions (from the `secondary_response_mapping` *specification in* **ModelNested***)*

Private Member Functions

- DataModelRep ()

  constructor

- ~DataModelRep ()

  destructor

- void write (std::ostream &s) const

  write a DataModelRep object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataModelRep object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataModelRep object to a packed MPI buffer

Private Attributes

- int referenceCount

  number of handle objects sharing this dataModelRep

Friends

- class DataModel

  the handle class can access attributes of the body class directly

43.25.1 Detailed Description

Body class for model specification data. The DataModelRep class is used to contain the data from a model keyword specification. Default values are managed in the DataModelRep constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::dataModelList is private.

The documentation for this class was generated from the following files:

- DataModel.H
- DataModel.C

43.26 DataResponses Class Reference

Handle class for responses specification data.

Public Member Functions

- DataResponses ()

  constructor

- DataResponses (const DataResponses &)

  copy constructor

- ~DataResponses ()

  destructor

- DataResponses & operator= (const DataResponses &)

  assignment operator

- void write (std::ostream &s) const

  write a DataResponses object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataResponses object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataResponses object to a packed MPI buffer

Static Public Member Functions

- static bool id_compare (const DataResponses &dr, const std::string &id)

  compares the idResponses attribute of DataResponses objects

Private Attributes

- DataResponsesRep * dataRespRep

  pointer to the body (handle-body idiom)

Friends

- class **ProblemDescDB**
- class **NIDRProblemDescDB**

- void run_dakota_data ()

  library_mode default data initializer

43.26.1 Detailed Description

Handle class for responses specification data. The DataResponses class is used to provide a memory management handle for the data in DataResponsesRep. It is populated by IDRProblemDescDB::responses_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataResponses objects is maintained in ProblemDescDB::dataResponsesList, one for each responses specification in an input file.

The documentation for this class was generated from the following files:

- DataResponses.H
- DataResponses.C

43.27 DataResponsesRep Class Reference

Body class for responses specification data.

Public Attributes

- size_t numObjectiveFunctions

  number of objective functions (from the `num_objective_functions` *specification in* **RespFnOpt***)*

- size_t numNonlinearIneqConstraints

  number of nonlinear inequality constraints (from the `num_nonlinear_inequality_constraints` *specification in* **RespFnOpt***)*

- size_t numNonlinearEqConstraints

  number of nonlinear equality constraints (from the `num_nonlinear_equality_constraints` *specification in* **RespFnOpt***)*

- size_t numLeastSqTerms

  number of least squares terms (from the `num_least_squares_terms` *specification in* **RespFnLS***)*

- size_t numResponseFunctions

  number of generic response functions (from the `num_response_functions` *specification in* **RespFnGen***)*

- StringArray primaryRespFnScaleTypes

  vector of primary response function scaling types (from the `objective_function_scale_types` *specification in* **RespFnOpt** *and the* `least_squares_term_scale_types` *specification in* **RespFnLS***)*

- RealVector primaryRespFnScales

  vector of primary response function scaling factors (from the `objective_function_scales` *specification in* **RespFnOpt** *and the* `least_squares_term_scales` *specification in* **RespFnLS***)*

- RealVector primaryRespFnWeights

  vector of weightings for multiobjective optimization or weighted nonlinear least squares (from the `multi_objective_weights` *specification in* **RespFnOpt** *and the* `least_squares_weights` *specification in* **RespFnLS***)*

- RealVector expConfigVars

  list of num_experiments x num_config_vars configuration variable values

- RealVector expObservations

  list of num_calibration_terms observation data

- RealVector expStdDeviations

  list of 1 or num_calibration_terms observation standard deviations

- String expDataFileName

name of experimental data file containing response data (with optional state variable and sigma data) to read

- bool expDataFileAnnotated

  whether the experimental data is in annotated format

- size_t numExperiments

  number of rows of experimental data (replicates or distinct experiments)

- size_t numExpConfigVars

  number of experimental configuration vars (state variables) in each row of data

- size_t numExpStdDeviations

  whether to read num_responses standard deviations from each row of data file

- RealVector nonlinearIneqLowerBnds

  vector of nonlinear inequality constraint lower bounds (from the `nonlinear_inequality_lower_bounds` *specification in* **RespFnOpt***)*

- RealVector nonlinearIneqUpperBnds

  vector of nonlinear inequality constraint upper bounds (from the `nonlinear_inequality_upper_bounds` *specification in* **RespFnOpt***)*

- StringArray nonlinearIneqScaleTypes

  vector of nonlinear inequality constraint scaling types (from the `nonlinear_inequality_scale_types` *specification in* **RespFnOpt***)*

- RealVector nonlinearIneqScales

  vector of nonlinear inequality constraint scaling factors (from the `nonlinear_inequality_scales` *specification in* **RespFnOpt***)*

- RealVector nonlinearEqTargets

  vector of nonlinear equality constraint targets (from the `nonlinear_equality_targets` *specification in* **RespFnOpt***)*

- StringArray nonlinearEqScaleTypes

  vector of nonlinear equality constraint scaling types (from the `nonlinear_equality_scale_types` *specification in* **RespFnOpt***)*

- RealVector nonlinearEqScales

  vector of nonlinear equality constraint scaling factors (from the `nonlinear_equality_scales` *specification in* **RespFnOpt***)*

- String gradientType

  gradient type: none, numerical, analytic, or mixed (from the `no_gradients`*,* `numerical_gradients`*,* `analytic_gradients`*, and* `mixed_gradients` *specifications in* **RespGrad***)*

- String hessianType

Hessian type: none, numerical, quasi, analytic, or mixed (from the `no_hessians`*,* `numerical_hessians`*,* `quasi_hessians`*,* `analytic_hessians`*, and* `mixed_hessians` *specifications in* **RespHess***).*

- bool ignoreBounds

  option to ignore bounds when doing finite differences (default is to honor bounds)

- bool centralHess

  Temporary(?) option to use old 2nd-order diffs when computing finite-difference Hessians; default is forward differences.

- String quasiHessianType

  quasi-Hessian type: bfgs, damped_bfgs, or sr1 (from the `bfgs` *and* `sr1` *specifications in* **RespHess***)*

- String methodSource

  numerical gradient method source: dakota or vendor (from the `method_source` *specification in* **RespGradNum** *and* **RespGradMixed***)*

- String intervalType

  numerical gradient interval type: forward or central (from the `interval_type` *specification in* **RespGradNum** *and* **RespGradMixed***)*

- RealVector fdGradStepSize

  vector of finite difference step sizes for numerical gradients, one step size per active continuous variable, used in computing 1st-order forward or central differences (from the `fd_gradient_step_size` *specification in* **RespGradNum** *and* **RespGradMixed***)*

- RealVector fdHessStepSize

  vector of finite difference step sizes for numerical Hessians, one step size per active continuous variable, used in computing 1st-order gradient-based differences and 2nd-order function-based differences (from the `fd_hessian_step_size` *specification in* **RespHessNum** *and* **RespHessMixed***)*

- IntList idNumericalGrads

  mixed gradient numerical identifiers (from the `id_numerical_gradients` *specification in* **RespGradMixed***)*

- IntList idAnalyticGrads

  mixed gradient analytic identifiers (from the `id_analytic_gradients` *specification in* **RespGradMixed***)*

- IntList idNumericalHessians

  mixed Hessian numerical identifiers (from the `id_numerical_hessians` *specification in* **RespHessMixed***)*

- IntList idQuasiHessians

  mixed Hessian quasi identifiers (from the `id_quasi_hessians` *specification in* **RespHessMixed***)*

- IntList idAnalyticHessians

  mixed Hessian analytic identifiers (from the `id_analytic_hessians` *specification in* **RespHessMixed***)*

- String idResponses

string identifier for the responses specification data set (from the `id_responses` *specification in* **RespSetId***)*

- StringArray responseLabels

  the response labels array (from the `response_descriptors` *specification in* **RespLabels***)*

Private Member Functions

- DataResponsesRep ()

  constructor

- ~DataResponsesRep ()

  destructor

- void write (std::ostream &s) const

  write a DataResponsesRep object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataResponsesRep object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataResponsesRep object to a packed MPI buffer

Private Attributes

- int referenceCount

  number of handle objects sharing this dataResponsesRep

Friends

- class DataResponses

  the handle class can access attributes of the body class directly

43.27.1 Detailed Description

Body class for responses specification data. The DataResponsesRep class is used to contain the data from a responses keyword specification. Default values are managed in the DataResponsesRep constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::dataResponsesList is private.

The documentation for this class was generated from the following files:

- DataResponses.H

- DataResponses.C

43.28 DataStrategy Class Reference

Handle class for strategy specification data.

Public Member Functions

- DataStrategy ()
  constructor
- DataStrategy (const DataStrategy &)
  copy constructor
- ~DataStrategy ()
  destructor
- DataStrategy & operator= (const DataStrategy &)
  assignment operator
- void write (std::ostream &s) const
  write a DataStrategy object to an std::ostream
- void read (MPIUnpackBuffer &s)
  read a DataStrategy object from a packed MPI buffer
- void write (MPIPackBuffer &s) const
  write a DataStrategy object to a packed MPI buffer

Private Attributes

- DataStrategyRep * dataStratRep
  pointer to the body (handle-body idiom)

Friends

- class **ProblemDescDB**
- class **NIDRProblemDescDB**

43.28.1 Detailed Description

Handle class for strategy specification data. The DataStrategy class is used to provide a memory management handle for the data in DataStrategyRep. It is populated by IDRProblemDescDB::strategy_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A single DataStrategy object is maintained in ProblemDescDB::strategySpec.

The documentation for this class was generated from the following files:

- DataStrategy.H
- DataStrategy.C

43.29 DataStrategyRep Class Reference

Body class for strategy specification data.

Public Attributes

- String strategyType

  the strategy selection: hybrid, multi_start, pareto_set, or single_method

- bool graphicsFlag

  flags use of graphics by the strategy (from the `graphics` *specification in* **StratIndControl***)*

- bool tabularDataFlag

  flags tabular data collection by the strategy (from the tabular_graphics_data specification in **StratIndControl***)*

- String tabularDataFile

  the filename used for tabular data collection by the strategy (from the tabular_graphics_file specification in **StratIndControl***)*

- int outputPrecision

  output precision for tabular and screen output

- int iteratorServers

  number of servers for concurrent iterator parallelism (from the iterator_servers specification in **StratIndControl***)*

- String iteratorScheduling

  type of scheduling (self or static) used in concurrent iterator parallelism (from the `iterator_self_scheduling` *and* `iterator_static_scheduling` *specifications in* **StratIndControl***)*

- String methodPointer

  method identifier for the strategy (from the `opt_method_pointer` *specifications in* **StratParetoSet** *and* `method_pointer` *specifications in* **StratSingle** *and* **StratMultiStart***)*

- StringArray hybridMethodList

  array of methods for the sequential and collaborative hybrid optimization strategies (from the `method_list` *specification in* **StratHybrid***)*

- String hybridType

  the type of hybrid optimization strategy: collaborative, embedded, sequential, or sequential_adaptive (from the `collaborative`*,* `embedded`*, and* `sequential` *specifications in* **StratHybrid***)*

- String hybridGlobalMethodPointer

  global method pointer for embedded hybrids (from the `global_method_pointer` *specification in* **StratHybrid***)*

- String hybridLocalMethodPointer

local method pointer for embedded hybrids (from the `local_method_pointer` *specification in* **StratHybrid***)*

- Real hybridLSProb

  local search probability for embedded hybrids (from the `local_search_probability` *specification in* **StratHybrid***)*

- int concurrentRandomJobs

  number of random jobs to perform in the concurrent strategy (from the `random_starts` *and* `random_weight_sets` *specifications in* **StratMultiStart** *and* **StratParetoSet***)*

- int concurrentSeed

  seed for the selected random jobs within the concurrent strategy (from the `seed` *specification in* **StratMultiStart** *and* **StratParetoSet***)*

- RealVector concurrentParameterSets

  user-specified (i.e., nonrandom) parameter sets to evaluate in the concurrent strategy (from the `starting_points` *and* `multi_objective_weight_sets` *specifications in* **StratMultiStart** *and* **StratParetoSet***)*

Private Member Functions

- DataStrategyRep ()

  constructor

- ~DataStrategyRep ()

  destructor

- void write (std::ostream &s) const

  write a DataStrategyRep object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataStrategyRep object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataStrategyRep object to a packed MPI buffer

Private Attributes

- int referenceCount

  number of handle objects sharing this dataStrategyRep

Friends

- class DataStrategy

  the handle class can access attributes of the body class directly

43.29.1 Detailed Description

Body class for strategy specification data. The DataStrategyRep class is used to contain the data from the strategy keyword specification. Default values are managed in the DataStrategyRep constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::strategySpec is private.

The documentation for this class was generated from the following files:

- DataStrategy.H
- DataStrategy.C

43.30 DataVariables Class Reference

Handle class for variables specification data.

Public Member Functions

- DataVariables ()

  constructor

- DataVariables (const DataVariables &)

  copy constructor

- ~DataVariables ()

  destructor

- DataVariables operator= (const DataVariables &)

  assignment operator

- bool operator== (const DataVariables &)

  equality operator

- void write (std::ostream &s) const

  write a DataVariables object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataVariables object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataVariables object to a packed MPI buffer

- size_t design ()

  return total number of design variables

- size_t aleatory_uncertain ()

  return total number of aleatory uncertain variables

- size_t epistemic_uncertain ()

  return total number of epistemic uncertain variables

- size_t uncertain ()

  return total number of uncertain variables

- size_t state ()

  return total number of state variables

- size_t continuous_variables ()

  return total number of continuous variables

- size_t discrete_variables ()

  return total number of discrete variables

- size_t total_variables ()

  return total number of variables

Static Public Member Functions

- static bool id_compare (const DataVariables &dv, const std::string &id)

  compares the idVariables attribute of DataVariables objects

Private Attributes

- DataVariablesRep * dataVarsRep

  pointer to the body (handle-body idiom)

Friends

- class **ProblemDescDB**
- class **NIDRProblemDescDB**
- void run_dakota_data ()

  library_mode default data initializer

43.30.1 Detailed Description

Handle class for variables specification data. The DataVariables class is used to provide a memory management handle for the data in DataVariablesRep. It is populated by IDRProblemDescDB::variables_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataVariables objects is maintained in ProblemDescDB::dataVariablesList, one for each variables specification in an input file.

The documentation for this class was generated from the following files:

- DataVariables.H
- DataVariables.C

43.31 DataVariablesRep Class Reference

Body class for variables specification data.

Public Attributes

- String idVariables

  string identifier for the variables specification data set (from the `id_variables` *specification in* **VarSetId***)*

- size_t numContinuousDesVars

  number of continuous design variables (from the `continuous_design` *specification in* **VarDV***)*

- size_t numDiscreteDesRangeVars

  number of discrete design variables defined by an integer range (from the `discrete_design_range` *specification in* **VarDV***)*

- size_t numDiscreteDesSetIntVars

  number of discrete design variables defined by a set of integers (from the `discrete_design_set_integer` *specification in* **VarDV***)*

- size_t numDiscreteDesSetRealVars

  number of discrete design variables defined by a set of reals (from the `discrete_design_set_real` *specification in* **VarDV***)*

- size_t numNormalUncVars

  number of normal uncertain variables (from the `normal_uncertain` *specification in* **VarAUV***)*

- size_t numLognormalUncVars

  number of lognormal uncertain variables (from the `lognormal_uncertain` *specification in* **VarAUV***)*

- size_t numUniformUncVars

  number of uniform uncertain variables (from the `uniform_uncertain` *specification in* **VarAUV***)*

- size_t numLoguniformUncVars

  number of loguniform uncertain variables (from the `loguniform_uncertain` *specification in* **VarAUV***)*

- size_t numTriangularUncVars

  number of triangular uncertain variables (from the `triangular_uncertain` *specification in* **VarAUV***)*

- size_t numExponentialUncVars

  number of exponential uncertain variables (from the `exponential_uncertain` *specification in* **VarAUV***)*

- size_t numBetaUncVars

  number of beta uncertain variables (from the `beta_uncertain` *specification in* **VarAUV***)*

- size_t numGammaUncVars

number of gamma uncertain variables (from the `gamma_uncertain` *specification in* **VarAUV***)*

- size_t numGumbelUncVars

  number of gumbel uncertain variables (from the `gumbel_uncertain` *specification in* **VarAUV***)*

- size_t numFrechetUncVars

  number of frechet uncertain variables (from the `frechet_uncertain` *specification in* **VarAUV***)*

- size_t numWeibullUncVars

  number of weibull uncertain variables (from the `weibull_uncertain` *specification in* **VarAUV***)*

- size_t numHistogramBinUncVars

  number of histogram bin uncertain variables (from the `histogram_bin_uncertain` *specification in* **VarAUV***)*

- size_t numPoissonUncVars

  number of Poisson uncertain variables (from the `poisson_uncertain` *specification in* **VarAUV***)*

- size_t numBinomialUncVars

  number of binomial uncertain variables (from the `binomial_uncertain` *specification in* **VarAUV***)*

- size_t numNegBinomialUncVars

  number of negative binomial uncertain variables (from the `negative_binomial_uncertain` *specification in* **VarAUV***)*

- size_t numGeometricUncVars

  number of geometric uncertain variables (from the `geometric_uncertain` *specification in* **VarAUV**

- size_t numHyperGeomUncVars

  number of hypergeometric uncertain variables (from the `hypergeometric_uncertain` *specification in* **VarAUV***))*

- size_t numHistogramPtUncVars

  number of histogram point uncertain variables (from the `histogram_point_uncertain` *specification in* **VarAUV***)*

- size_t numIntervalUncVars

  number of interval uncertain variables (from the `interval_uncertain` *specification in* **VarEUV***)*

- size_t numContinuousStateVars

  number of continuous state variables (from the `continuous_state` *specification in* **VarSV***)*

- size_t numDiscreteStateRangeVars

  number of discrete state variables defined by an integer range (from the `discrete_state_range` *specification in* **VarDV***)*

- size_t numDiscreteStateSetIntVars

number of discrete state variables defined by a set of integers (from the `discrete_state_set_integer` *specification in* **VarDV***)*

- size_t numDiscreteStateSetRealVars

  number of discrete state variables defined by a set of reals (from the `discrete_state_set_real` *specification in* **VarDV***)*

- RealVector continuousDesignVars

  initial values for the continuous design variables array (from the `continuous_design` `initial_point` *specification in* **VarDV***)*

- RealVector continuousDesignLowerBnds

  lower bounds array for the continuous design variables (from the `continuous_design` `lower_bounds` *specification in* **VarDV***)*

- RealVector continuousDesignUpperBnds

  upper bounds array for the continuous design variables (from the `continuous_design` `upper_bounds` *specification in* **VarDV***)*

- StringArray continuousDesignScaleTypes

  scale types array for the continuous design variables (from the `continuous_design` `scale_types` *specification in* **VarDV***)*

- RealVector continuousDesignScales

  scales array for the continuous design variables (from the `continuous_design` `scales` *specification in* **VarDV***)*

- IntVector discreteDesignRangeVars

  initial values for the discrete design variables defined by an integer range (from the `discrete_design_range` `initial_point` *specification in* **VarDV***)*

- IntVector discreteDesignRangeLowerBnds

  lower bounds array for the discrete design variables defined by an integer range (from the `discrete_design_range` `lower_bounds` *specification in* **VarDV***)*

- IntVector discreteDesignRangeUpperBnds

  upper bounds array for the discrete design variables defined by an integer range(from the `discrete_design_range` `upper_bounds` *specification in* **VarDV***)*

- IntVector discreteDesignSetIntVars

  initial values for the discrete design variables defined by an integer set (from the `discrete_design_set_integer` `initial_point` *specification in* **VarDV***)*

- RealVector discreteDesignSetRealVars

  initial values for the discrete design variables defined by a real set (from the `discrete_design_set_real` `initial_point` *specification in* **VarDV***)*

- IntSetArray discreteDesignSetInt

complete set of admissible values for each of the discrete design variables defined by an integer set (from the `discrete_design_set_integer` `set_values` *specification in* **VarDV***)*

- RealSetArray discreteDesignSetReal

  complete set of admissible values for each of the discrete design variables defined by a real set (from the `discrete_design_set_real` `set_values` *specification in* **VarDV***)*

- StringArray continuousDesignLabels

  labels array for the continuous design variables (from the `continuous_design` `descriptors` *specification in* **VarDV***)*

- StringArray discreteDesignRangeLabels

  labels array for the discrete design variables defined by an integer range (from the `discrete_design_range` `descriptors` *specification in* **VarDV***)*

- StringArray discreteDesignSetIntLabels

  labels array for the discrete design variables defined by an integer set (from the `discrete_design_range` `descriptors` *specification in* **VarDV***)*

- StringArray discreteDesignSetRealLabels

  labels array for the discrete design variables defined by a real set (from the `discrete_design_range` `descriptors` *specification in* **VarDV***)*

- RealVector normalUncMeans

  means of the normal uncertain variables (from the `nuv_means` *specification in* **VarAUV***)*

- RealVector normalUncStdDevs

  standard deviations of the normal uncertain variables (from the `nuv_std_deviations` *specification in* **VarAUV***)*

- RealVector normalUncLowerBnds

  distribution lower bounds for the normal uncertain variables (from the `nuv_lower_bounds` *specification in* **VarAUV***)*

- RealVector normalUncUpperBnds

  distribution upper bounds for the normal uncertain variables (from the `nuv_upper_bounds` *specification in* **VarAUV***)*

- RealVector lognormalUncLambdas

  lambdas (means of the corresponding normals) of the lognormal uncertain variables (from the `lnuv_lambdas` *specification in* **VarAUV***)*

- RealVector lognormalUncZetas

  zetas (standard deviations of the corresponding normals) of the lognormal uncertain variables (from the `lnuv_zetas` *specification in* **VarAUV***)*

- RealVector lognormalUncMeans

  means of the lognormal uncertain variables (from the `lnuv_means` *specification in* **VarAUV***)*

- RealVector lognormalUncStdDevs

  standard deviations of the lognormal uncertain variables (from the `lnuv_std_deviations` *specification in* **VarAUV***)*

- RealVector lognormalUncErrFacts

  error factors for the lognormal uncertain variables (from the `lnuv_error_factors` *specification in* **VarAUV***)*

- RealVector lognormalUncLowerBnds

  distribution lower bounds for the lognormal uncertain variables (from the `lnuv_lower_bounds` *specification in* **VarAUV***)*

- RealVector lognormalUncUpperBnds

  distribution upper bounds for the lognormal uncertain variables (from the `lnuv_upper_bounds` *specification in* **VarAUV***)*

- RealVector uniformUncLowerBnds

  distribution lower bounds for the uniform uncertain variables (from the `uuv_lower_bounds` *specification in* **VarAUV***)*

- RealVector uniformUncUpperBnds

  distribution upper bounds for the uniform uncertain variables (from the `uuv_upper_bounds` *specification in* **VarAUV***)*

- RealVector loguniformUncLowerBnds

  distribution lower bounds for the loguniform uncertain variables (from the `luuv_lower_bounds` *specification in* **VarAUV***)*

- RealVector loguniformUncUpperBnds

  distribution upper bounds for the loguniform uncertain variables (from the `luuv_upper_bounds` *specification in* **VarAUV***)*

- RealVector triangularUncModes

  modes of the triangular uncertain variables (from the `tuv_modes` *specification in* **VarAUV***)*

- RealVector triangularUncLowerBnds

  distribution lower bounds for the triangular uncertain variables (from the `tuv_lower_bounds` *specification in* **VarAUV***)*

- RealVector triangularUncUpperBnds

  distribution upper bounds for the triangular uncertain variables (from the `tuv_upper_bounds` *specification in* **VarAUV***)*

- RealVector exponentialUncBetas

  beta factors for the exponential uncertain variables (from the `euv_betas` *specification in* **VarAUV***)*

- RealVector betaUncAlphas

alpha factors for the beta uncertain variables (from the `buv_means` *specification in* **VarAUV***)*

- RealVector betaUncBetas

  beta factors for the beta uncertain variables (from the `buv_std_deviations` *specification in* **VarAUV***)*

- RealVector betaUncLowerBnds

  distribution lower bounds for the beta uncertain variables (from the `buv_lower_bounds` *specification in* **VarAUV***)*

- RealVector betaUncUpperBnds

  distribution upper bounds for the beta uncertain variables (from the `buv_upper_bounds` *specification in* **VarAUV***)*

- RealVector gammaUncAlphas

  alpha factors for the gamma uncertain variables (from the `gauv_alphas` *specification in* **VarAUV***)*

- RealVector gammaUncBetas

  beta factors for the gamma uncertain variables (from the `gauv_betas` *specification in* **VarAUV***)*

- RealVector gumbelUncAlphas

  alpha factors for the gumbel uncertain variables (from the `guuv_alphas` *specification in* **VarAUV***)*

- RealVector gumbelUncBetas

  beta factors for of the gumbel uncertain variables (from the `guuv_betas` *specification in* **VarAUV***)*

- RealVector frechetUncAlphas

  alpha factors for the frechet uncertain variables (from the `fuv_alphas` *specification in* **VarAUV***)*

- RealVector frechetUncBetas

  beta factors for the frechet uncertain variables (from the `fuv_betas` *specification in* **VarAUV***)*

- RealVector weibullUncAlphas

  alpha factors for the weibull uncertain variables (from the `wuv_alphas` *specification in* **VarAUV***)*

- RealVector weibullUncBetas

  beta factors for the weibull uncertain variables (from the `wuv_betas` *specification in* **VarAUV***)*

- RealVectorArray histogramUncBinPairs

  an array containing a vector of (x,c) pairs for each bin-based histogram uncertain variable (see continuous linear histogram in LHS manual; from the `histogram_bin_uncertain` *specification in* **VarAUV***). (x,y) ordinate specifications are converted to (x,c) counts within NIDR.*

- RealVector poissonUncLambdas

  lambdas (rate parameter) for the poisson uncertain variables (from the `lambdas` *specification in* **VarAUV***)*

- RealVector binomialUncProbPerTrial

probabilities per each trial (p) for the binomial uncertain variables from the `prob_per_trial` *specification in* **VarAUV***)*

- IntVector binomialUncNumTrials

  Number of trials (N) for the binomial uncertain variables from the `num_trials` *specification in* **VarAUV***).*

- RealVector negBinomialUncProbPerTrial

  probabilities per each trial (p) for the negative binomial uncertain variables from the `prob_per_trial` *specification in* **VarAUV***)*

- IntVector negBinomialUncNumTrials

  Number of trials (N) for the negative binomial uncertain variables from the `num_trials` *specification in* **VarAUV***).*

- RealVector geometricUncProbPerTrial

  probabilities per each trial (p) for the geometric uncertain variables from the `prob_per_trial` *specification in* **VarAUV***)*

- IntVector hyperGeomUncTotalPop

  Size of total populations (N) for the hypergeometric uncertain variables from the `total_population` *specification in* **VarAUV***).*

- IntVector hyperGeomUncSelectedPop

  Size of selected populations for the hypergeometric uncertain variables from the `selected_population` *specification in* **VarAUV***).*

- IntVector hyperGeomUncNumDrawn

  Number failed in the selected populations for the hypergeometric variablesfrom the `num_drawn` *specification in* **VarAUV***).*

- RealVectorArray histogramUncPointPairs

  an array containing a vector of (x,c) pairs for each point-based histogram uncertain variable (see discrete histogram in LHS manual; from the `histogram_point_uncertain` *specification in* **VarAUV***)*

- RealVectorArray intervalUncBasicProbs

  Probability values per interval uncertain variable (from the `iuv_interval_probs` *specification in* **VarEUV***).*

- RealVectorArray intervalUncBounds

  Interval Bounds per interval uncertain variable (from the `iuv_interval_bounds` *specification in* **VarEUV***).*

- RealSymMatrix uncertainCorrelations

  correlation matrix for all uncertain variables (from the `uncertain_correlation_matrix` *specification in* **VarAUV***). This matrix specifies rank correlations for sampling methods (i.e., LHS) and correlation coefficients (rho_ij = normalized covariance matrix) for analytic reliability methods.*

- RealVector continuousStateVars

  initial values for the continuous state variables array (from the `continuous_state` `initial_point` *specification in* **VarSV***)*

- RealVector continuousStateLowerBnds

  lower bounds array for the continuous state variables (from the `continuous_state` `lower_bounds` *specification in* **VarSV***)*

- RealVector continuousStateUpperBnds

  upper bounds array for the continuous state variables (from the `continuous_state` `upper_bounds` *specification in* **VarSV***)*

- IntVector discreteStateRangeVars

  initial values for the discrete state variables defined by an integer range (from the `discrete_state_range` `initial_point` *specification in* **VarSV***)*

- IntVector discreteStateRangeLowerBnds

  lower bounds array for the discrete state variables defined by an integer range (from the `discrete_state_range` `lower_bounds` *specification in* **VarSV***)*

- IntVector discreteStateRangeUpperBnds

  upper bounds array for the discrete state variables defined by an integer range(from the `discrete_state_range` `upper_bounds` *specification in* **VarSV***)*

- IntVector discreteStateSetIntVars

  initial values for the discrete state variables defined by an integer set (from the `discrete_state_set_integer` `initial_point` *specification in* **VarSV***)*

- RealVector discreteStateSetRealVars

  initial values for the discrete state variables defined by a real set (from the `discrete_state_set_real` `initial_point` *specification in* **VarSV***)*

- IntSetArray discreteStateSetInt

  complete set of admissible values for each of the discrete state variables defined by an integer set (from the `discrete_state_set_integer` `set_values` *specification in* **VarSV***)*

- RealSetArray discreteStateSetReal

  complete set of admissible values for each of the discrete state variables defined by a real set (from the `discrete_state_set_real` `set_values` *specification in* **VarSV***)*

- StringArray continuousStateLabels

  labels array for the continuous state variables (from the `continuous_state` `descriptors` *specification in* **VarSV***)*

- StringArray discreteStateRangeLabels

  labels array for the discrete state variables defined by an integer range (from the `discrete_state_range` `descriptors` *specification in* **VarSV***)*

- StringArray discreteStateSetIntLabels

  labels array for the discrete state variables defined by an integer set (from the `discrete_state_range` `descriptors` *specification in* **VarSV***)*

- StringArray discreteStateSetRealLabels

  labels array for the discrete state variables defined by a real set (from the `discrete_state_range descriptors` *specification in* **VarSV***)*

- IntVector discreteDesignSetIntLowerBnds

  discrete design integer set lower bounds inferred from set values

- IntVector discreteDesignSetIntUpperBnds

  discrete design integer set upper bounds inferred from set values

- RealVector discreteDesignSetRealLowerBnds

  discrete design real set lower bounds inferred from set values

- RealVector discreteDesignSetRealUpperBnds

  discrete design real set upper bounds inferred from set values

- RealVector continuousAleatoryUncVars

  array of values for all continuous aleatory uncertain variables

- RealVector continuousAleatoryUncLowerBnds

  distribution lower bounds for all continuous aleatory uncertain variables (collected from `nuv_lower_bounds`*,* `lnuv_lower_bounds`*,* `uuv_lower_bounds`*,* `luuv_lower_bounds`*,* `tuv_lower_bounds`*, and* `buv_lower_bounds` *specifications in* **VarAUV***, and derived for gamma, gumbel, frechet, weibull and histogram bin specifications)*

- RealVector continuousAleatoryUncUpperBnds

  distribution upper bounds for all continuous aleatory uncertain variables (collected from `nuv_upper_bounds`*,* `lnuv_upper_bounds`*,* `uuv_upper_bounds`*,* `luuv_upper_bounds`*,* `tuv_lower_bounds`*, and* `buv_upper_bounds` *specifications in* **VarAUV***, and derived for gamma, gumbel, frechet, weibull and histogram bin specifications)*

- StringArray continuousAleatoryUncLabels

  labels for all continuous aleatory uncertain variables (collected from `nuv_descriptors`*,* `lnuv_descriptors`*,* `uuv_descriptors`*,* `luuv_descriptors`*,* `tuv_descriptors`*,* `buv_descriptors`*,* `gauv_descriptors`*,* `guuv_descriptors`*,* `fuv_descriptors`*,* `wuv_descriptors`*, and* `hbuv_descriptors` *specifications in* **VarAUV***)*

- IntVector discreteIntAleatoryUncVars

  array of values for all discrete integer aleatory uncertain variables

- IntVector discreteIntAleatoryUncLowerBnds

  distribution lower bounds for all discrete integer aleatory uncertain variables

- IntVector discreteIntAleatoryUncUpperBnds

  distribution upper bounds for all discrete integer aleatory uncertain variables

- StringArray discreteIntAleatoryUncLabels

  labels for all discrete integer aleatory uncertain variables

- RealVector discreteRealAleatoryUncVars

  array of values for all discrete real aleatory uncertain variables

- RealVector discreteRealAleatoryUncLowerBnds

  distribution lower bounds for all discrete real aleatory uncertain variables

- RealVector discreteRealAleatoryUncUpperBnds

  distribution upper bounds for all discrete real aleatory uncertain variables

- StringArray discreteRealAleatoryUncLabels

  labels for all discrete real aleatory uncertain variables

- RealVector continuousEpistemicUncVars

  array of values for all continuous epistemic uncertain variables

- RealVector continuousEpistemicUncLowerBnds

  distribution lower bounds for all continuous epistemic uncertain variables

- RealVector continuousEpistemicUncUpperBnds

  distribution upper bounds for all continuous epistemic uncertain variables

- StringArray continuousEpistemicUncLabels

  labels for all continuous epistemic uncertain variables

- IntVector discreteStateSetIntLowerBnds

  discrete state integer set lower bounds inferred from set values

- IntVector discreteStateSetIntUpperBnds

  discrete state integer set upper bounds inferred from set values

- RealVector discreteStateSetRealLowerBnds

  discrete state real set lower bounds inferred from set values

- RealVector discreteStateSetRealUpperBnds

  discrete state real set upper bounds inferred from set values

Private Member Functions

- DataVariablesRep ()

  default constructor

- ~DataVariablesRep ()

destructor

- void write (std::ostream &s) const

  write a DataVariablesRep object to an std::ostream

- void read (MPIUnpackBuffer &s)

  read a DataVariablesRep object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a DataVariablesRep object to a packed MPI buffer

Private Attributes

- int referenceCount

  number of handle objects sharing dataVarsRep

Friends

- class DataVariables

  the handle class can access attributes of the body class directly

43.31.1 Detailed Description

Body class for variables specification data. The DataVariablesRep class is used to contain the data from a variables keyword specification. Default values are managed in the DataVariablesRep constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::dataVariablesList is private.

The documentation for this class was generated from the following files:

- DataVariables.H
- DataVariables.C

43.32 DDACEDesignCompExp Class Reference

Wrapper class for the DDACE design of experiments library. Inheritance diagram for DDACEDesignCompExp::

Public Member Functions

- DDACEDesignCompExp (Model &model)

  primary constructor for building a standard DACE iterator

- DDACEDesignCompExp (Model &model, int samples, int symbols, int seed, const String &sampling_method)

  alternate constructor used for building approximations

- ~DDACEDesignCompExp ()

  destructor

- void pre_run ()

  pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori

- void extract_trends ()

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

- void post_input ()

  read tabular data for post-run mode

- void post_run (std::ostream &s)

  post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way

- int num_samples () const

  get the current number of samples

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)

  reset sampling iterator to use at least min_samples

- const String & sampling_scheme () const

  return sampling name

- void vary_pattern (bool pattern_flag)

  sets varyPattern in derived classes that support it

- void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

Private Member Functions

- void compute_main_effects ()

  builds a DDaceMainEffects::OneWayANOVA if mainEffectsFlag is set

- void resolve_samples_symbols ()

  convenience function for resolving number of samples and number of symbols from input.

Static Private Member Functions

- static void copy_data (const std::vector< DDaceSamplePoint > &dspa, Real ∗ptr, const int ptr_len)

  copy DDACE point to RealVector

Private Attributes

- String daceMethod

  oas, lhs, oa_lhs, random, box_behnken, central_composite, or grid

- int samplesSpec

  initial specification of number of samples

- int symbolsSpec

  initial specification of number of symbols

- int numSamples

  current number of samples to be evaluated

- int numSymbols

  current number of symbols to be used in generating the sample set (inversely related to number of replications)

- const int seedSpec

  the user seed specification for the random number generator (allows repeatable results)

- int randomSeed

  current seed for the random number generator

- bool allDataFlag

  flag which triggers the update of allVars/allResponses for use by Iterator::all_variables() and Iterator::all_responses()

- size_t numDACERuns

  counter for number of run() executions for this object

- bool varyPattern

  flag for continuing the random number sequence from a previous run() execution (e.g., for surrogate-based optimization) so that multiple executions are repeatable but not correlated.

- bool mainEffectsFlag

  flag which specifies main effects

- std::vector< std::vector< int > > symbolMapping

  mapping of symbols for main effects calculations

43.32.1 Detailed Description

Wrapper class for the DDACE design of experiments library. The DDACEDesignCompExp class provides a wrapper for DDACE, a C++ design of experiments library from the Computational Sciences and Mathematics Research (CSMR) department at Sandia's Livermore CA site. This class uses design and analysis of computer experiments (DACE) methods to sample the design space spanned by the bounds of a Model. It returns all generated samples and their corresponding responses as well as the best sample found.

43.32.2 Constructor & Destructor Documentation

43.32.2.1 DDACEDesignCompExp (Model & *model*)

primary constructor for building a standard DACE iterator This constructor is called for a standard iterator built with data from probDescDB.

References Dakota::abort_handler(), DDACEDesignCompExp::daceMethod, DDACEDesignCompExp::mainEffectsFlag, Iterator::maxConcurrency, Iterator::numContinuousVars, and DDACEDesignCompExp::numSamples.

43.32.2.2 DDACEDesignCompExp (Model & *model*, int *samples*, int *symbols*, int *seed*, const String & *sampling_method*)

alternate constructor used for building approximations This alternate constructor is used for instantiations on-the-fly, using only the incoming data. No problem description database queries are used.

References Iterator::maxConcurrency, DDACEDesignCompExp::numSamples, and DDACEDesignCompExp::resolve_samples_symbols().

43.32.3 Member Function Documentation

43.32.3.1 void pre_run () [virtual]

pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori pre-run phase, which a derived iterator may optionally reimplement; when not present, pre-run is likely integrated into the derived run function. This is a virtual function; when re-implementing, a derived class must call its nearest parent's pre_run(), if implemented, typically _before_ performing its own implementation steps.

Reimplemented from Iterator.

References DDACEDesignCompExp::get_parameter_sets(), Iterator::iteratedModel, and PStudyDACE::varBasedDecompFlag.

43.32.3.2 void post_run (std::ostream & *s*) [virtual]

post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way Post-run phase, which a derived iterator may optionally reimplement; when not present, post-run is likely integrated into run. This is a virtual function; when re-implementing, a derived class must call its nearest parent's post_run(), typically _after_ performing its own implementation steps.

Reimplemented from Iterator.

References Analyzer::allResponses, Analyzer::allSamples, SensAnalysisGlobal::compute_correlations(), DDACEDesignCompExp::compute_main_effects(), DDACEDesignCompExp::mainEffectsFlag, PStudyDACE::pStudyDACESensGlobal, Iterator::subIteratorFlag, and PStudyDACE::varBasedDecompFlag.

43.32.3.3 int num_samples () const [inline, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References DDACEDesignCompExp::numSamples.

43.32.3.4 void resolve_samples_symbols () [private]

convenience function for resolving number of samples and number of symbols from input. This function must define a combination of samples and symbols that is acceptable for a particular sampling algorithm. Users provide requests for these quantities, but this function must enforce any restrictions imposed by the sampling algorithms.

References Dakota::abort_handler(), DDACEDesignCompExp::daceMethod, Iterator::numContinuousVars, DDACEDesignCompExp::numSamples, and DDACEDesignCompExp::numSymbols.

Referenced by DDACEDesignCompExp::DDACEDesignCompExp(), and DDACEDesignCompExp::get_-parameter_sets().

43.32.3.5 void copy_data (const std::vector< DDaceSamplePoint > & *dspa*, Real * *ptr*, const int *ptr_len*) [static, private]

copy DDACE point to RealVector copy DDACE point array to RealVectorArray copy DDACE point array to Real*

References Dakota::abort_handler().

Referenced by DDACEDesignCompExp::get_parameter_sets().

The documentation for this class was generated from the following files:

- DDACEDesignCompExp.H
- DDACEDesignCompExp.C

43.33 DirectApplicInterface Class Reference

Derived application interface class which spawns simulation codes and testers using direct procedure calls. Inheritance diagram for DirectApplicInterface::

Public Member Functions

- DirectApplicInterface (const ProblemDescDB &problem_db)

  constructor

- ~DirectApplicInterface ()

  destructor

- void derived_map (const Variables &vars, const ActiveSet &set, Response &response, int fn_eval_id)

  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- void derived_map_asynch (const ParamResponsePair &pair)

  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- void derived_synch (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- void derived_synch_nowait (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- int derived_synchronous_local_analysis (const int &analysis_id)
- const StringArray & analysis_drivers () const

  retrieve the analysis drivers specification for application interfaces

Protected Member Functions

- virtual int derived_map_if (const Dakota::String &if_name)

  execute the input filter portion of a direct evaluation invocation

- virtual int derived_map_ac (const Dakota::String &ac_name)

  execute an analysis code portion of a direct evaluation invocation

- virtual int derived_map_of (const Dakota::String &of_name)

  execute the output filter portion of a direct evaluation invocation

- void set_local_data (const Variables &vars, const ActiveSet &set, const Response &response)

  convenience function for local test simulators which sets per-evaluation variable and response attributes

- void overlay_response (Response &response)

  convenience function for local test simulators which overlays response contributions from multiple analyses using MPI_Reduce

Protected Attributes

- String iFilterName

  name of the direct function input filter

- String oFilterName

  name of the direct function output filter

- driver_t iFilterType

  enum type of the direct function input filter

- driver_t oFilterType

  enum type of the direct function output filter

- bool gradFlag

  signals use of fnGrads in direct simulator functions

- bool hessFlag

  signals use of fnHessians in direct simulator functions

- size_t numFns

  number of functions in fnVals

- size_t numVars

  total number of continuous and discrete variables

- size_t numACV

total number of continuous variables

- size_t numADIV

  total number of discete integer variables

- size_t numADRV

  total number of discete real variables

- size_t numDerivVars

  number of active derivative variables

- unsigned short localDataView

  bit-wise record of which local data views are active; see enum local_data_t

- RealVector xC

  continuous variables used within direct simulator fns

- IntVector xDI

  discrete int variables used within direct simulator fns

- RealVector xDR

  discrete real variables used within direct simulator fns

- StringMultiArray xCLabels

  continuous variable labels

- StringMultiArray xDILabels

  discrete integer variable labels

- StringMultiArray xDRLabels

  discrete real variable labels

- std::map< String, var_t > varTypeMap

  map from variable label to enum

- std::map< String, driver_t > driverTypeMap

  map from driver name to enum

- std::map< var_t, Real > xCM

  map from var_t enum to continuous value

- std::map< var_t, int > xDIM

  map from var_t enum to discrete int value

- std::map< var_t, Real > xDRM

  map from var_t enum to discrete real value

- std::vector< var_t > varTypeDVV

  var_t enumerations corresponding to DVV components

- std::vector< var_t > xCMLabels

  var_t enumerations corresponding to continuous variable labels

- std::vector< var_t > xDIMLabels

  var_t enumerations corresponding to discrete integer variable labels

- std::vector< var_t > xDRMLabels

  var_t enumerations corresponding to discrete real variable labels

- ShortArray directFnASV

  class scope active set vector

- SizetArray directFnDVV

  class scope derivative variables vector

- RealVector fnVals

  response fn values within direct simulator fns

- RealMatrix fnGrads

  response fn gradients w/i direct simulator fns

- RealSymMatrixArray fnHessians

  response fn Hessians within direct fns

- StringArray analysisDrivers

  the set of analyses within each function evaluation (from the analysis_drivers interface specification)

- std::vector< driver_t > analysisDriverTypes

  conversion of analysisDrivers to driver_t

- size_t analysisDriverIndex

  the index of the active analysis driver within analysisDrivers

- String2DArray analysisComponents

  the set of optional analysis components used by the analysis drivers (from the analysis_components interface specification)

- engine * matlabEngine

  pointer to the MATLAB engine used for direct evaluations

Private Member Functions

- int cantilever ()

  scaled cantilever test function for optimization

- int mod_cantilever ()

  unscaled cantilever test function for UQ

- int cyl_head ()

  the cylinder head constrained optimization test fn

- int multimodal ()

  multimodal UQ test function

- int log_ratio ()

  the log_ratio UQ test function

- int short_column ()

  the short_column UQ/OUU test function

- int lf_short_column ()

  a low fidelity short_column test function

- int rosenbrock ()

  the Rosenbrock optimization and least squares test fn

- int generalized_rosenbrock ()

  n-dimensional Rosenbrock (Schittkowski)

- int extended_rosenbrock ()

  n-dimensional Rosenbrock (Nocedal/Wright)

- int lf_rosenbrock ()

  a low fidelity version of the Rosenbrock function

- int gerstner ()

  the isotropic/anisotropic Gerstner test function family

- int scalable_gerstner ()

  scalable versions of the Gerstner test family

- int steel_column_cost ()

  the steel_column_cost UQ/OUU test function

- int steel_column_perf ()

  the steel_column_perf UQ/OUU test function

- int sobol_rational ()

  Sobol SA rational test function.

- int sobol_g_function ()

  Sobol SA discontinuous test function.

- int sobol_ishigami ()

  Sobol SA transcendental test function.

- int text_book ()

  the text_book constrained optimization test function

- int text_book1 ()

  portion of text_book() evaluating the objective fn

- int text_book2 ()

  portion of text_book() evaluating constraint 1

- int text_book3 ()

  portion of text_book() evaluating constraint 2

- int text_book_ouu ()

  the text_book_ouu OUU test function

- int scalable_text_book ()

  scalable version of the text_book test function

- void herbie1D (size_t der_mode, Real xc_loc, std::vector< Real > &w_and_ders)

  1D components of herbie function

- void smooth_herbie1D (size_t der_mode, Real xc_loc, std::vector< Real > &w_and_ders)

  1D components of smooth_herbie function

- void shubert1D (size_t der_mode, Real xc_loc, std::vector< Real > &w_and_ders)

  1D components of shubert function

- int herbie ()

  returns the N-D herbie function

- int smooth_herbie ()

  returns the N-D smooth herbie function

- int shubert ()

  returns the N-D shubert function

- void separable_combine (Real mult_scale_factor, std::vector< Real > &w, std::vector< Real > &d1w, std::vector< Real > &d2w)

  utility to combine components of seperable fns

- int salinas ()

  direct interface to the SALINAS structural dynamics code

- int mc_api_run ()

  direct interface to ModelCenter via API, HKIM 4/3/03

- int matlab_engine_run ()

  direct interface to Matlab via API, BMA 11/28/05

- int matlab_field_prep (mxArray *dakota_matlab, const char *field_name)

  check that the dakota_matlab strucutre has the specified field_name and add if necessary; free structure memory in preparation for new alloc

- int python_run ()

  direct interface to Python via API, BMA 07/02/07

- template<class ArrayT , class Size >
  bool python_convert_int (const ArrayT &src, Size size, PyObject **dst)

  convert arrays of integer types to Python list or numpy array

- bool python_convert (const RealVector &src, PyObject **dst)

  convert RealVector to Python list or numpy array

- bool python_convert (const RealVector &c_src, const IntVector &di_src, const RealVector &dr_src, PyObject **dst)

  convert RealVector + IntVector + RealVector to Python mixed list or numpy double array

- bool python_convert (const StringMultiArray &src, PyObject **dst)

  convert labels

- bool python_convert (const StringMultiArray &c_src, const StringMultiArray &di_src, const StringMultiArray &dr_src, PyObject **dst)

  convert all labels to single list

- bool python_convert (PyObject *pyv, RealVector &rv, const int &dim)

  convert python [list of int or float] or [numpy array of double] to RealVector (for fns)

- bool python_convert (PyObject *pyv, double *rv, const int &dim)

  convert python [list of int or float] or [numpy array of double] to double[], for use as helper in converting gradients

- bool python_convert (PyObject *pym, RealMatrix &rm)

  convert python [list of lists of int or float] or [numpy array of dbl] to RealMatrix (for gradients)

- bool python_convert (PyObject *pym, RealSymMatrix &rm)

  convert python [list of lists of int or float] or [numpy array of dbl] to RealMatrix (used as helper in Hessian conversion)

- bool python_convert (PyObject *pyma, RealSymMatrixArray &rma)

  convert python [list of lists of lists of int or float] or [numpy array of double] to RealSymMatrixArray (for Hessians)

Private Attributes

- bool userNumpyFlag

  whether the user requested numpy data structures

43.33.1 Detailed Description

Derived application interface class which spawns simulation codes and testers using direct procedure calls. DirectApplicInterface uses a few linkable simulation codes and several internal member functions to perform parameter to response mappings.

43.33.2 Member Function Documentation

43.33.2.1 int derived_synchronous_local_analysis (const int & *analysis_id*) [inline, virtual]

This code provides the derived function used by ApplicationInterface::serve_analyses_synch().

Reimplemented from ApplicationInterface.

References DirectApplicInterface::analysisDriverIndex, DirectApplicInterface::analysisDrivers, and DirectApplicInterface::derived_map_ac().

43.33.2.2 int derived_map_ac (const Dakota::String & *ac_name*) [protected, virtual]

execute an analysis code portion of a direct evaluation invocation When a direct analysis/filter is a member function, the (vars,set,response) data does not need to be passed through the API. If, however, non-member analysis/filter functions are added, then pass (vars,set,response) through to the non-member fns:

```
// API declaration
int sim(const Variables& vars, const ActiveSet& set, Response& response);
// use of API within derived_map_ac()
if (ac_name == "sim")
  fail_code = sim(directFnVars, directFnActSet, directFnResponse);
```

Reimplemented in ParallelDirectApplicInterface, and SerialDirectApplicInterface.

References Dakota::abort_handler(), ApplicationInterface::analysisServerId, DirectApplicInterface::cantilever(), DirectApplicInterface::cyl_head(), DirectApplicInterface::driverTypeMap,

DirectApplicInterface::extended_rosenbrock(), DirectApplicInterface::generalized_rosenbrock(), DirectApplicInterface::gerstner(), DirectApplicInterface::herbie(), DirectApplicInterface::lf_rosenbrock(), DirectApplicInterface::lf_short_column(), DirectApplicInterface::log_ratio(), DirectApplicInterface::matlab_engine_run(), DirectApplicInterface::mc_api_run(), DirectApplicInterface::mod_cantilever(), DirectApplicInterface::multimodal(), DirectApplicInterface::python_run(), DirectApplicInterface::rosenbrock(), DirectApplicInterface::salinas(), DirectApplicInterface::scalable_gerstner(), DirectApplicInterface::scalable_text_book(), DirectApplicInterface::short_column(), DirectApplicInterface::shubert(), DirectApplicInterface::smooth_herbie(), DirectApplicInterface::sobol_g_function(), DirectApplicInterface::sobol_ishigami(), DirectApplicInterface::sobol_rational(), DirectApplicInterface::steel_column_cost(), DirectApplicInterface::steel_column_perf(), DirectApplicInterface::text_book(), DirectApplicInterface::text_book1(), DirectApplicInterface::text_book2(), DirectApplicInterface::text_book3(), and DirectApplicInterface::text_book_ouu().

Referenced by DirectApplicInterface::derived_map(), and DirectApplicInterface::derived_synchronous_local_analysis().

43.33.2.3 void herbie1D (size_t *der_mode*, Real *xc_loc*, std::vector< Real > & *w_and_ders*) [private]

1D components of herbie function 1D Herbie function and its derivatives (apart from a multiplicative factor)

Referenced by DirectApplicInterface::herbie().

43.33.2.4 void smooth_herbie1D (size_t *der_mode*, Real *xc_loc*, std::vector< Real > & *w_and_ders*) [private]

1D components of smooth_herbie function 1D Smoothed Herbie= 1DHerbie minus the high frequency sine term, and its derivatives (apart from a multiplicative factor)

Referenced by DirectApplicInterface::smooth_herbie().

43.33.2.5 void shubert1D (size_t *der_mode*, Real *xc_loc*, std::vector< Real > & *w_and_ders*) [private]

1D components of shubert function 1D Shubert function and its derivatives (apart from a multiplicative factor)

Referenced by DirectApplicInterface::shubert().

43.33.2.6 int herbie () [private]

returns the N-D herbie function N-D Herbie function and its derivatives.

References DirectApplicInterface::directFnASV, DirectApplicInterface::directFnDVV, DirectApplicInterface::herbie1D(), DirectApplicInterface::numDerivVars, DirectApplicInterface::numVars, DirectApplicInterface::separable_combine(), and DirectApplicInterface::xC.

Referenced by DirectApplicInterface::derived_map_ac().

43.33.2.7 int smooth_herbie () [private]

returns the N-D smooth herbie function N-D Smoothed Herbie function and its derivatives.

References DirectApplicInterface::directFnASV, DirectApplicInterface::directFnDVV, DirectApplicInterface::numDerivVars, DirectApplicInterface::numVars, DirectApplicInterface::separable_combine(), DirectApplicInterface::smooth_herbie1D(), and DirectApplicInterface::xC.

Referenced by DirectApplicInterface::derived_map_ac().

43.33.2.8 void separable_combine (Real *mult_scale_factor*, std::vector< Real > & *w*, std::vector< Real > & *d1w*, std::vector< Real > & *d2w*) [private]

utility to combine components of seperable fns this function combines N 1D functions and their derivatives to compute a N-D separable function and its derivatives, logic is general enough to support different 1D functions in different dimensions (can mix and match)

References DirectApplicInterface::directFnASV, DirectApplicInterface::directFnDVV, DirectApplicInterface::fnGrads, DirectApplicInterface::fnHessians, DirectApplicInterface::fnVals, DirectApplicInterface::numDerivVars, and DirectApplicInterface::numVars.

Referenced by DirectApplicInterface::herbie(), DirectApplicInterface::shubert(), and DirectApplicInterface::smooth_herbie().

43.33.2.9 bool python_convert_int (const ArrayT & *src*, Size *sz*, PyObject ** *dst*) [inline, private]

convert arrays of integer types to Python list or numpy array convert all integer array types including IntVector, ShortArray, and SizetArray to Python list of ints or numpy array of ints

References DirectApplicInterface::userNumpyFlag.

Referenced by DirectApplicInterface::python_run().

The documentation for this class was generated from the following files:

- DirectApplicInterface.H
- DirectApplicInterface.C

43.34 DiscrepancyCorrection Class Reference

Base class for discrepancy corrections.

Public Member Functions

- DiscrepancyCorrection ()

  default constructor

- DiscrepancyCorrection (Model &surr_model, const IntSet &surr_fn_indices, short corr_type, short corr_order)

  standard constructor

- DiscrepancyCorrection (const IntSet &surr_fn_indices, size_t num_fns, size_t num_vars, short corr_type, short corr_order)

  alternate constructor

- ~DiscrepancyCorrection ()

  destructor

- void initialize (Model &surr_model, const IntSet &surr_fn_indices, short corr_type, short corr_order)

  initialize the DiscrepancyCorrection data

- void initialize (const IntSet &surr_fn_indices, size_t num_fns, size_t num_vars, short corr_type, short corr_order)

  initialize the DiscrepancyCorrection data

- void compute (const RealVector &c_vars, const Response &truth_response, const Response &approx_response, bool quiet_flag=false)

  compute the correction required to bring approx_response into agreement with truth_response and store in {add,mult}Corrections

- void compute (const Response &truth_response, const Response &approx_response, Response &discrepancy_response, bool quiet_flag=false)

  compute the correction required to bring approx_response into agreement with truth_response and store in discrepancy_response

- void apply (const RealVector &c_vars, Response &approx_response, bool quiet_flag=false)

  apply the correction computed in compute() to approx_response

- bool active () const

  indicates an active correction via non-empty correctionType

- short correction_type () const

  return correctionType

- short correction_order () const

  return correctionOrder

- short data_order () const

  return dataOrder

- bool computed () const

  return correctionComputed

Protected Attributes

- IntSet surrogateFnIndices

  for mixed response sets, this array specifies the response function subset that is approximated

- short correctionType

  approximation correction approach to be used: NO_CORRECTION, ADDITIVE_CORRECTION, MULTIPLICATIVE_CORRECTION, or COMBINED_CORRECTION.

- short correctionOrder

  approximation correction order to be used: 0, 1, or 2

- short dataOrder

  order of correction data in 3-bit format: overlay of 1 (value), 2 (gradient), and 4 (Hessian)

- bool correctionComputed

  flag indicating whether or not a correction has been computed and is available for application

- size_t numFns

  total number of response functions (of which surrogateFnIndices may define a subset)

- size_t numVars

  number of continuous variables active in the correction

Private Member Functions

- void initialize_corrections ()

  internal convenience function shared by overloaded initialize() variants

- bool check_scaling (const RealVector &truth_fns, const RealVector &approx_fns)

  define badScalingFlag

- void compute_additive (const Response &truth_response, const Response &approx_response, int index, Real &discrep_fn, RealVector &discrep_grad, RealSymMatrix &discrep_hess)

internal convenience function for computing additive corrections between truth and approximate responses

- void compute_multiplicative (const Response &truth_response, const Response &approx_response, int index, Real &discrep_fn, RealVector &discrep_grad, RealSymMatrix &discrep_hess)

  internal convenience function for computing multiplicative corrections between truth and approximate responses

- void apply_additive (const RealVector &c_vars, Response &approx_response)

  internal convenience function for applying additive corrections to an approximate response

- void apply_multiplicative (const RealVector &c_vars, Response &approx_response)

  internal convenience function for applying multiplicative corrections to an approximate response

- void apply_additive (const RealVector &c_vars, RealVector &approx_fns)

  internal convenience function for applying additive corrections to a set of response functions

- void apply_multiplicative (const RealVector &c_vars, RealVector &approx_fns)

  internal convenience function for applying multiplicative corrections to a set of response functions

- const Response & search_db (const RealVector &c_vars, const ShortArray &search_asv)

  search data_pairs for missing approximation data

Private Attributes

- bool badScalingFlag

  flag used to indicate function values near zero for multiplicative corrections; triggers an automatic switch to additive corrections

- bool computeAdditive

  flag indicating the need for additive correction calculations

- bool computeMultiplicative

  flag indicating the need for multiplicative correction calculations

- std::vector< Approximation > addCorrections

  array of additive corrections; surrogate models of a model discrepancy function (formed from model differences)

- std::vector< Approximation > multCorrections

  array of multiplicative corrections; surrogate models of a model discrepancy function (formed from model ratios)

- Model surrModel

  shallow copy of the surrogate model instance as returned by Model::surrogate_model() (the DataFitSurrModel or HierarchSurrModel::lowFidelityModel instance)

- RealVector combineFactors

factors for combining additive and multiplicative corrections. Each factor is the weighting applied to the additive correction and 1.-factor is the weighting applied to the multiplicative correction. The factor value is determined by an additional requirement to match the high fidelity function value at the previous correction point (e.g., previous trust region center). This results in a multipoint correction instead of a strictly local correction.

- RealVector correctionPrevCenterPt

  copy of center point from the previous correction cycle

- RealVector truthFnsCenter

  truth function values at the current correction point

- RealVector approxFnsCenter

  Surrogate function values at the current correction point.

- RealMatrix approxGradsCenter

  Surrogate gradient values at the current correction point.

- RealVector truthFnsPrevCenter

  copy of truth function values at center of previous correction cycle

- RealVector approxFnsPrevCenter

  copy of approximate function values at center of previous correction cycle

43.34.1 Detailed Description

Base class for discrepancy corrections. The DiscrepancyCorrection class provides common functions for computing and applying corrections to approximations.

43.34.2 Member Function Documentation

43.34.2.1 void compute (const RealVector & *c_vars*, const Response & *truth_response*, const Response & *approx_response*, bool *quiet_flag* = `false`)

compute the correction required to bring approx_response into agreement with truth_response and store in {add,mult}Corrections Compute an additive or multiplicative correction that corrects the approx_response to have 0th-order consistency (matches values), 1st-order consistency (matches values and gradients), or 2nd-order consistency (matches values, gradients, and Hessians) with the truth_response at a single point (e.g., the center of a trust region). The 0th-order, 1st-order, and 2nd-order corrections use scalar values, linear scaling functions, and quadratic scaling functions, respectively, for each response function.

References Response::active_set(), DiscrepancyCorrection::addCorrections, DiscrepancyCorrection::apply(), DiscrepancyCorrection::apply_additive(), DiscrepancyCorrection::apply_multiplicative(), DiscrepancyCorrection::approxFnsCenter, DiscrepancyCorrection::approxFnsPrevCenter, DiscrepancyCorrection::approxGradsCenter, DiscrepancyCorrection::badScalingFlag, DiscrepancyCorrection::check_scaling(), DiscrepancyCorrection::combineFactors, DiscrepancyCorrection::compute_additive(),

DiscrepancyCorrection::compute_multiplicative(), DiscrepancyCorrection::computeAdditive, DiscrepancyCorrection::computeMultiplicative, Response::copy(), DiscrepancyCorrection::correctionComputed, DiscrepancyCorrection::correctionOrder, DiscrepancyCorrection::correctionPrevCenterPt, DiscrepancyCorrection::correctionType, DiscrepancyCorrection::dataOrder, Response::function_gradients(), Response::function_values(), Model::is_null(), DiscrepancyCorrection::multCorrections, DiscrepancyCorrection::numFns, DiscrepancyCorrection::numVars, ActiveSet::request_values(), DiscrepancyCorrection::surrModel, DiscrepancyCorrection::surrogateFnIndices, DiscrepancyCorrection::truthFnsCenter, and DiscrepancyCorrection::truthFnsPrevCenter.

Referenced by HierarchSurrModel::derived_compute_response(), DataFitSurrModel::derived_compute_response(), HierarchSurrModel::derived_synchronize(), DataFitSurrModel::derived_synchronize(), HierarchSurrModel::derived_synchronize_nowait(), DataFitSurrModel::derived_synchronize_nowait(), and SurrBasedLocalMinimizer::minimize_surrogates().

The documentation for this class was generated from the following files:

- DiscrepancyCorrection.H
- DiscrepancyCorrection.C

43.35 DOTOptimizer Class Reference

Wrapper class for the DOT optimization library. Inheritance diagram for DOTOptimizer::

Public Member Functions

- DOTOptimizer (Model &model)

  standard constructor

- DOTOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor

- ~DOTOptimizer ()

  destructor

- void find_optimum ()

  Used within the optimizer branch for computing the optimal solution. Redefines the run virtual function for the optimizer branch.

Protected Member Functions

- void initialize_run ()

  performs run-time set up

Private Member Functions

- void initialize ()

  Shared constructor code.

- void allocate_workspace ()

  Allocates workspace for the optimizer.

- void allocate_constraints ()

  Allocates constraint mappings.

Private Attributes

- int dotInfo

  INFO from DOT manual.

- int dotFDSinfo

  internal DOT parameter NGOTOZ

- int dotMethod

  METHOD from DOT manual.

- int printControl

  IPRINT from DOT manual (controls output verbosity).

- int optimizationType

  MINMAX from DOT manual (minimize or maximize).

- RealArray realCntlParmArray

  RPRM from DOT manual.

- IntArray intCntlParmArray

  IPRM from DOT manual.

- RealVector designVars

  array of design variable values passed to DOT

- Real objFnValue

  value of the objective function passed to DOT

- RealVector constraintValues

  array of nonlinear constraint values passed to DOT

- int realWorkSpaceSize

  size of realWorkSpace

- int intWorkSpaceSize

  size of intWorkSpace

- RealArray realWorkSpace

  real work space for DOT

- IntArray intWorkSpace

int work space for DOT

- int numDotNlnConstr

  total number of nonlinear constraints seen by DOT

- int numDotLinConstr

  total number of linear constraints seen by DOT

- int numDotConstr

  total number of linear and nonlinear constraints seen by DOT

- SizetArray constraintMappingIndices

  a container of indices for referencing the corresponding Response constraints used in computing the DOT constraints.

- RealArray constraintMappingMultipliers

  a container of multipliers for mapping the Response constraints to the DOT constraints.

- RealArray constraintMappingOffsets

  a container of offsets for mapping the Response constraints to the DOT constraints.

43.35.1 Detailed Description

Wrapper class for the DOT optimization library. The DOTOptimizer class provides a wrapper for DOT, a commercial Fortran 77 optimization library from Vanderplaats Research and Development. It uses a reverse communication mode, which avoids the static member function issues that arise with function pointer designs (see NPSOLOptimizer and SNLLOptimizer).

The user input mappings are as follows: `max_iterations` is mapped into DOT's `ITMAX` parameter within its `IPRM` array, `max_function_evaluations` is implemented directly in the find_optimum() loop since there is no DOT parameter equivalent, `convergence_tolerance` is mapped into DOT's `DELOBJ` parameter (the relative convergence tolerance) within its `RPRM` array, `output` verbosity is mapped into DOT's `IPRINT` parameter within its function call parameter list (verbose: `IPRINT` = 7; quiet: `IPRINT` = 3), and `optimization_type` is mapped into DOT's `MINMAX` parameter within its function call parameter list. Refer to [Vanderplaats Research and Development, 1995] for information on `IPRM`, `RPRM`, and the DOT function call parameter list.

43.35.2 Member Data Documentation

43.35.2.1 int dotInfo [private]

INFO from DOT manual. Information requested by DOT: 0=optimization complete, 1=get values, 2=get gradients

Referenced by DOTOptimizer::find_optimum(), and DOTOptimizer::initialize_run().

43.35.2.2 int dotFDSinfo [private]

internal DOT parameter NGOTOZ the DOT parameter list has been modified to pass NGOTOZ, which signals whether DOT is finite-differencing (nonzero value) or performing the line search (zero value).

Referenced by DOTOptimizer::find_optimum().

43.35.2.3 int dotMethod [private]

METHOD from DOT manual. For nonlinear constraints: 0/1 = dot_mmfd, 2 = dot_slp, 3 = dot_sqp. For unconstrained: 0/1 = dot_bfgs, 2 = dot_frcg.

Referenced by DOTOptimizer::allocate_constraints(), DOTOptimizer::allocate_workspace(), DOTOptimizer::DOTOptimizer(), and DOTOptimizer::find_optimum().

43.35.2.4 int printControl [private]

IPRINT from DOT manual (controls output verbosity). Values range from 0 (least output) to 7 (most output).

Referenced by DOTOptimizer::DOTOptimizer(), and DOTOptimizer::find_optimum().

43.35.2.5 int optimizationType [private]

MINMAX from DOT manual (minimize or maximize). Values of 0 or -1 (minimize) or 1 (maximize).

Referenced by DOTOptimizer::DOTOptimizer(), and DOTOptimizer::find_optimum().

43.35.2.6 RealArray realCntlParmArray [private]

RPRM from DOT manual. Array of real control parameters.

Referenced by DOTOptimizer::find_optimum(), and DOTOptimizer::initialize().

43.35.2.7 IntArray intCntlParmArray [private]

IPRM from DOT manual. Array of integer control parameters.

Referenced by DOTOptimizer::find_optimum(), and DOTOptimizer::initialize().

43.35.2.8 RealVector constraintValues [private]

array of nonlinear constraint values passed to DOT This array must be of nonzero length and must contain only one-sided inequality constraints which are <= 0 (which requires a transformation from 2-sided inequalities and equalities).

Referenced by DOTOptimizer::allocate_constraints(), and DOTOptimizer::find_optimum().

43.35.2.9 SizetArray constraintMappingIndices [private]

a container of indices for referencing the corresponding Response constraints used in computing the DOT constraints. The length of the container corresponds to the number of DOT constraints, and each entry in the container points to the corresponding DAKOTA constraint.

Referenced by DOTOptimizer::allocate_constraints(), and DOTOptimizer::find_optimum().

43.35.2.10 RealArray constraintMappingMultipliers [private]

a container of multipliers for mapping the Response constraints to the DOT constraints. The length of the container corresponds to the number of DOT constraints, and each entry in the container stores a multiplier for the DAKOTA constraint identified with constraintMappingIndices. These multipliers are currently +1 or -1.

Referenced by DOTOptimizer::allocate_constraints(), and DOTOptimizer::find_optimum().

43.35.2.11 RealArray constraintMappingOffsets [private]

a container of offsets for mapping the Response constraints to the DOT constraints. The length of the container corresponds to the number of DOT constraints, and each entry in the container stores an offset for the DAKOTA constraint identified with constraintMappingIndices. These offsets involve inequality bounds or equality targets, since DOT assumes constraint allowables = 0.

Referenced by DOTOptimizer::allocate_constraints(), and DOTOptimizer::find_optimum().

The documentation for this class was generated from the following files:

- DOTOptimizer.H
- DOTOptimizer.C

43.36 Driver Class Reference

A subclass of the JEGA front end driver that exposes the individual protected methods to execute the algorithm.

Public Member Functions

- GeneticAlgorithm ∗ ExtractAllData (const AlgorithmConfig &algConfig)

  Reads all required data from the problem description database stored in the supplied algorithm config.

- DesignOFSortSet PerformIterations (GeneticAlgorithm ∗theGA)

  Performs the required iterations on the supplied GA.

- void DestroyAlgorithm (GeneticAlgorithm ∗theGA)

  Deletes the supplied GA.

- Driver (const ProblemConfig &probConfig)

  Default constructs a Driver.

43.36.1 Detailed Description

A subclass of the JEGA front end driver that exposes the individual protected methods to execute the algorithm. This is necessary because DAKOTA requires that all problem information be extracted from the problem description DB at the time of Optimizer construction and the front end does it all in the execute algorithm method which must be called in find_optimum.

43.36.2 Constructor & Destructor Documentation

43.36.2.1 Driver (const ProblemConfig & *probConfig*) `[inline]`

Default constructs a Driver.

Parameters:

probConfig The definition of the problem to be solved by this Driver whenever ExecuteAlgorithm is called.

The problem can be solved in multiple ways by multiple algorithms even using multiple different evaluators by issuing multiple calls to ExecuteAlgorithm with different AlgorithmConfigs.

43.36.3 Member Function Documentation

43.36.3.1 GeneticAlgorithm∗ ExtractAllData (const AlgorithmConfig & *algConfig*) `[inline]`

Reads all required data from the problem description database stored in the supplied algorithm config. The returned GA is fully configured and ready to be run. It must also be destroyed at some later time. You MUST call

DestroyAlgorithm for this purpose. Failure to do so could result in a memory leak and an eventual segmentation fault! Be sure to call DestroyAlgorithm prior to destroying the algorithm config that was used to create it!

This is just here to expose the base class method to users.

Parameters:

algConfig The fully loaded configuration object containing the database of parameters for the algorithm to be run on the known problem.

Returns:

The fully configured and loaded GA ready to be run using the PerformIterations method.

Referenced by JEGAOptimizer::find_optimum().

43.36.3.2 DesignOFSortSet PerformIterations (GeneticAlgorithm ∗ *theGA*) `[inline]`

Performs the required iterations on the supplied GA. This includes the calls to AlgorithmInitialize and AlgorithmFinalize and logs some information if appropriate.

This is just here to expose the base class method to users.

Parameters:

theGA The GA on which to perform iterations. This parameter must be non-null.

Returns:

The final solutions reported by the supplied GA after all iterations and call to AlgorithmFinalize.

Referenced by JEGAOptimizer::find_optimum().

43.36.3.3 void DestroyAlgorithm (GeneticAlgorithm ∗ *theGA*) `[inline]`

Deletes the supplied GA. Use this method to destroy a GA after all iterations have been run. This method knows if the log associated with the GA was created here and needs to be destroyed as well or not.

This is just here to expose the base class method to users.

Be sure to use this prior to destoying the algorithm config object which contains the target. The GA destructor needs the target to be in tact.

Parameters:

theGA The algorithm that is no longer needed and thus must be destroyed.

Referenced by JEGAOptimizer::find_optimum().

The documentation for this class was generated from the following file:

- JEGAOptimizer.C

43.37 EffGlobalMinimizer Class Reference

Implementation of Efficient Global Optimization/Least Squares algorithms. Inheritance diagram for EffGlobalMinimizer::

Public Member Functions

- EffGlobalMinimizer (Model &model)

  standard constructor

- ~EffGlobalMinimizer ()

  alternate constructor for instantiations "on the fly"

- void minimize_surrogates ()

  Used for computing the optimal solution using a surrogate-based approach. Redefines the Iterator::run() virtual function.

- const Model & algorithm_space_model () const

  return the result of any recasting or surrogate model recursion layered on top of iteratedModel by the derived Iterator ctor chain

Private Member Functions

- void minimize_surrogates_on_model ()

  called by minimize_surrogates for setUpType == "model"

- void get_best_sample ()

  called by minimize_surrogates for setUpType == "user_functions"

- Real expected_improvement (const RealVector &means, const RealVector &variances)

  expected improvement function for the GP

- RealVector expected_violation (const RealVector &means, const RealVector &variances)

  expected violation function for the constraint functions

- void update_penalty ()

  initialize and update the penaltyParameter

Static Private Member Functions

- static void EIF_objective_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in the Expected Improvement (EIF) problem formulation for PMA

Private Attributes

- String setUpType

  controls iteration mode: "model" (normal usage) or "user_functions" (user-supplied functions mode for "on the fly" instantiations).

- Model fHatModel

  GP model of response, one approximation per response function.

- Model eifModel

  recast model which assimilates mean and variance to solve the max(EIF) sub-problem

- Real meritFnStar

  minimum penalized response from among true function evaluations

- RealVector truthFnStar

  true function values corresponding to the minimum penalized response

- RealVector varStar

  point that corresponds to the optimal value meritFnStar

- short dataOrder

  order of the data used for surrogate construction, in ActiveSet request vector 3-bit format; user may override responses spec

Static Private Attributes

- static EffGlobalMinimizer * effGlobalInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.37.1 Detailed Description

Implementation of Efficient Global Optimization/Least Squares algorithms. The EffGlobalMinimizer class provides an implementation of the Efficient Global Optimization algorithm developed by Jones, Schonlau, & Welch as well as adaptation of the concept to nonlinear least squares.

43.37.2 Constructor & Destructor Documentation

43.37.2.1 ~EffGlobalMinimizer ()

alternate constructor for instantiations "on the fly" destructor

This is an alternate constructor for instantiations on the fly using a Model but no ProblemDescDB.

References SurrBasedMinimizer::approxSubProbMinimizer, EffGlobalMinimizer::eifModel, Model::free_communicators(), and Iterator::maximum_concurrency().

43.37.3 Member Function Documentation

43.37.3.1 void get_best_sample () [private]

called by minimize_surrogates for setUpType == "user_functions" determine best solution from among sample data for expected imporovement function

References Model::approximation_data(), SurrBasedMinimizer::augmented_lagrangian_merit(), Model::compute_response(), Model::continuous_variables(), Dakota::copy_data(), Model::current_response(), EffGlobalMinimizer::fHatModel, Response::function_values(), Iterator::iteratedModel, EffGlobalMinimizer::meritFnStar, Iterator::numFunctions, SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, Model::primary_response_fn_weights(), EffGlobalMinimizer::truthFnStar, and EffGlobalMinimizer::varStar.

Referenced by EffGlobalMinimizer::minimize_surrogates_on_model().

The documentation for this class was generated from the following files:

- EffGlobalMinimizer.H
- EffGlobalMinimizer.C

43.38 EmbeddedHybridStrategy Class Reference

Strategy for closely-coupled hybrid minimization, typically involving the embedding of local search methods within global search methods. Inheritance diagram for EmbeddedHybridStrategy::

Public Member Functions

- EmbeddedHybridStrategy (ProblemDescDB &problem_db)

  constructor

- ~EmbeddedHybridStrategy ()

  destructor

Protected Member Functions

- void run_strategy ()

  Performs the hybrid minimization strategy by executing multiple iterators on different models of varying fidelity.

- const Variables & variables_results () const

  return the final solution from selectedIterators (variables)

- const Response & response_results () const

  return the final solution from selectedIterators (response)

Private Attributes

- Real localSearchProb

  the probability of running a local search refinement within phases of the global minimization for coupled hybrids

43.38.1 Detailed Description

Strategy for closely-coupled hybrid minimization, typically involving the embedding of local search methods within global search methods. This strategy uses multiple methods in close coordination, generally using a local

search minimizer repeatedly within a global minimizer (the local search minimizer refines candidate minima which are fed back to the global minimizer).

The documentation for this class was generated from the following files:

- EmbeddedHybridStrategy.H
- EmbeddedHybridStrategy.C

43.39 Evaluator Class Reference

An evaluator specialization that knows how to interact with Dakota.

Public Member Functions

- virtual bool Evaluate (DesignGroup &group)

  Does evaluation of each design in group.

- virtual bool Evaluate (Design &des)

  This method cannot be used!!

- virtual std::string GetName () const

  Returns the proper name of this operator.

- virtual std::string GetDescription () const

  Returns a full description of what this operator does and how.

- virtual GeneticAlgorithmOperator ∗ Clone (GeneticAlgorithm &algorithm) const

  Creates and returns a pointer to an exact duplicate of this operator.

- Evaluator (GeneticAlgorithm &algorithm, Model &model)

  Constructs a Evaluator for use by algorithm.

- Evaluator (const Evaluator &copy)

  Copy constructs a Evaluator.

- Evaluator (const Evaluator &copy, GeneticAlgorithm &algorithm, Model &model)

  Copy constructs a Evaluator for use by algorithm.

Static Public Member Functions

- static const std::string & Name ()

  Returns the proper name of this operator.

- static const std::string & Description ()

  Returns a full description of what this operator does and how.

Protected Member Functions

- void SeparateVariables (const Design &from, RealVector &intoCont, IntVector &intoDiscInt, RealVector &intoDiscReal) const

  This method fills intoCont, intoDiscInt and intoDiscReal appropriately using the values of from.

- void RecordResponses (const RealVector &from, Design &into) const

  Records the computed objective and constraint function values into into.

- std::size_t GetNumberNonLinearConstraints () const

  Returns the number of non-linear constraints for the problem.

- std::size_t GetNumberLinearConstraints () const

  Returns the number of linear constraints for the problem.

Private Member Functions

- Evaluator (GeneticAlgorithm &algorithm)

  This constructor has no implementation and cannot be used.

Private Attributes

- Model & _model

  The Model known by this evaluator.

43.39.1 Detailed Description

An evaluator specialization that knows how to interact with Dakota. This evaluator knows how to use the model to do evaluations both in synchronous and asynchronous modes.

43.39.2 Constructor & Destructor Documentation

43.39.2.1 Evaluator (GeneticAlgorithm & *algorithm*, Model & *model*) `[inline]`

Constructs a Evaluator for use by *algorithm*. The optimizer is needed for purposes of variable scaling.

Parameters:

algorithm The GA for which the new evaluator is to be used.

model The model through which evaluations will be done.

Referenced by Evaluator::Clone().

43.39.2.2 Evaluator (const Evaluator & *copy*) [inline]

Copy constructs a Evaluator.

Parameters:

copy The evaluator from which properties are to be duplicated into this.

43.39.2.3 Evaluator (const Evaluator & *copy*, GeneticAlgorithm & *algorithm*, Model & *model*) [inline]

Copy constructs a Evaluator for use by *algorithm*. The optimizer is needed for purposes of variable scaling.

Parameters:

copy The existing Evaluator from which to retrieve properties.

algorithm The GA for which the new evaluator is to be used.

model The model through which evaluations will be done.

43.39.2.4 Evaluator (GeneticAlgorithm & *algorithm*) [private]

This constructor has no implementation and cannot be used. This constructor can never be used. It is provided so that this operator can still be registered in an operator registry even though it can never be instantiated from there.

Parameters:

algorithm The GA for which the new evaluator is to be used.

43.39.3 Member Function Documentation

43.39.3.1 static const std::string& Name () [inline, static]

Returns the proper name of this operator.

Returns:

The string "DAKOTA JEGA Evaluator".

Referenced by Evaluator::GetName().

43.39.3.2 static const std::string& Description () [inline, static]

Returns a full description of what this operator does and how. The returned text is:

```
This evaluator uses Sandia's DAKOTA optimization
software to evaluate the passed in Designs.  This
makes it possible to take advantage of the fact that
DAKOTA is designed to run on massively parallel machines.
```

.

Returns:

A description of the operation of this operator.

Referenced by Evaluator::GetDescription().

43.39.3.3 void SeparateVariables (const Design & *from*, RealVector & *intoCont*, IntVector & *intoDiscInt*, RealVector & *intoDiscReal*) const `[protected]`

This method fills *intoCont*, *intoDiscInt* and *intoDiscReal* appropriately using the values of *from*. The discrete integer design variable values are placed in *intoDiscInt*, the discrete real design variable values are placed in *intoDiscReal*, and the continuum are placed into *intoCont*. The values are written into the vectors from the beginning so any previous contents of the vectors will be overwritten.

Parameters:

from The Design class object from which to extract the discrete design variable values.

intoDiscInt The vector into which to place the extracted discrete integer values.

intoDiscReal The vector into which to place the extracted discrete real values.

intoCont The vector into which to place the extracted continuous values.

References Evaluator::_model, Model::cv(), Model::discrete_design_set_int_values(), Model::discrete_design_-set_real_values(), Model::div(), and Model::drv().

Referenced by Evaluator::Evaluate().

43.39.3.4 void RecordResponses (const RealVector & *from*, Design & *into*) const `[protected]`

Records the computed objective and constraint function values into *into*. This method takes the response values stored in *from* and properly transfers them into the *into* design.

The response vector *from* is expected to contain values for each objective function followed by values for each non-linear constraint in the order in which the info objects were loaded into the target by the optimizer class.

Parameters:

from The vector of responses to install into *into*.

into The Design to which the responses belong and into which they must be written.

References Evaluator::GetNumberNonLinearConstraints().

Referenced by Evaluator::Evaluate().

43.39.3.5 std::size_t GetNumberNonLinearConstraints () const `[inline, protected]`

Returns the number of non-linear constraints for the problem. This is computed by adding the number of non-linear equality constraints to the number of non-linear inequality constraints. These values are obtained from the model.

Returns:

The total number of non-linear constraints.

References Evaluator::_model, Model::num_nonlinear_eq_constraints(), and Model::num_nonlinear_ineq_-constraints().

Referenced by Evaluator::Evaluate(), and Evaluator::RecordResponses().

43.39.3.6 std::size_t GetNumberLinearConstraints () const [inline, protected]

Returns the number of linear constraints for the problem. This is computed by adding the number of linear equality constraints to the number of linear inequality constraints. These values are obtained from the model.

Returns:

The total number of linear constraints.

References Evaluator::_model, Model::num_linear_eq_constraints(), and Model::num_linear_ineq_-constraints().

43.39.3.7 bool Evaluate (DesignGroup & *group*) [virtual]

Does evaluation of each design in *group*. This method uses the Model known by this class to get Designs evaluated. It properly formats the Design class information in a way that Dakota will understand and then interprets the Dakota results and puts them back into the Design class object. It respects the asynchronous flag in the Model so evaluations may occur synchronously or asynchronously.

Prior to evaluating a Design, this class checks to see if it is marked as already evaluated. If it is, then the evaluation of that Design is not carried out. This is not strictly necessary because Dakota keeps track of evaluated designs and does not re-evaluate. An exception is the case of a population read in from a file complete with responses where Dakota is unaware of the evaluations.

Parameters:

group The group of Design class objects to be evaluated.

Returns:

true if all evaluations completed and false otherwise.

References Evaluator::_model, Model::asynch_compute_response(), Model::asynch_flag(), Model::compute_-response(), Model::continuous_variables(), Model::current_response(), Model::discrete_int_variables(), Model::discrete_real_variables(), Response::function_values(), Evaluator::GetName(), Evaluator::GetNumberNonLinearConstraints(), Evaluator::RecordResponses(), Evaluator::SeparateVariables(), and Model::synchronize().

43.39.3.8 virtual bool Evaluate (Design & *des*) [inline, virtual]

This method cannot be used!! This method does nothing and cannot be called. This is because in the case of asynchronous evaluation, this method would be unable to conform. It would require that each evaluation be done in a synchronous fashion.

Parameters:

des A Design that would be evaluated if this method worked.

Returns:

Would return true if the Design were evaluated and false otherwise. Never actually returns here. Issues a fatal error. Otherwise, it would always return false.

References Evaluator::GetName().

43.39.3.9 virtual std::string GetName () const `[inline, virtual]`

Returns the proper name of this operator.

Returns:

See Name().

References Evaluator::Name().

Referenced by Evaluator::Evaluate().

43.39.3.10 virtual std::string GetDescription () const `[inline, virtual]`

Returns a full description of what this operator does and how.

Returns:

See Description().

References Evaluator::Description().

43.39.3.11 virtual GeneticAlgorithmOperator* Clone (GeneticAlgorithm & *algorithm*) const `[inline, virtual]`

Creates and returns a pointer to an exact duplicate of this operator.

Parameters:

algorithm The GA for which the clone is being created.

Returns:

A clone of this operator.

References Evaluator::_model, and Evaluator::Evaluator().

43.39.4 Member Data Documentation

43.39.4.1 Model& _model [private]

The Model known by this evaluator. It is through this model that evaluations will take place.

Referenced by Evaluator::Clone(), Evaluator::Evaluate(), Evaluator::GetNumberLinearConstraints(), Evaluator::GetNumberNonLinearConstraints(), and Evaluator::SeparateVariables().

The documentation for this class was generated from the following file:

- JEGAOptimizer.C

43.40 EvaluatorCreator Class Reference

A specialization of the JEGA::FrontEnd::EvaluatorCreator that creates a new instance of a Evaluator.

Public Member Functions

- virtual GeneticAlgorithmEvaluator ∗ CreateEvaluator (GeneticAlgorithm &alg)

  Overriden to return a newly created Evaluator.

- EvaluatorCreator (Model &theModel)

  Constructs an EvaluatorCreator using the supplied model.

Private Attributes

- Model & _theModel

  The user defined model to be passed to the constructor of the Evaluator.

43.40.1 Detailed Description

A specialization of the JEGA::FrontEnd::EvaluatorCreator that creates a new instance of a Evaluator.

43.40.2 Constructor & Destructor Documentation

43.40.2.1 EvaluatorCreator (Model & *theModel*) `[inline]`

Constructs an EvaluatorCreator using the supplied model.

Parameters:

theModel The Dakota::Model this creator will pass to the created evaluator.

43.40.3 Member Function Documentation

43.40.3.1 virtual GeneticAlgorithmEvaluator∗ CreateEvaluator (GeneticAlgorithm & *alg*) `[inline, virtual]`

Overriden to return a newly created Evaluator. The GA will assume ownership of the evaluator so we needn't worry about keeping track of it for destruction. The additional parameters needed by the Evaluator are stored as members of this class at construction time.

Parameters:

alg The GA for which the evaluator is to be created.

Returns:

A pointer to a newly created Evaluator.

References EvaluatorCreator::_theModel.

The documentation for this class was generated from the following file:

- JEGAOptimizer.C

43.41 ForkAnalysisCode Class Reference

Derived class in the AnalysisCode class hierarchy which spawns simulations using forks. Inheritance diagram for ForkAnalysisCode::

Public Member Functions

- ForkAnalysisCode (const ProblemDescDB &problem_db)

  constructor

- ~ForkAnalysisCode ()

  destructor

- pid_t fork_program (const bool block_flag)

  spawn a child process using fork()/vfork()/execvp() and wait for completion using waitpid() if block_flag is true

- void check_status (const int status)

  check the exit status of a forked process and abort if an error code was returned

- void ifilter_argument_list ()

  set argList for execution of the input filter

- void ofilter_argument_list ()

  set argList for execution of the output filter

- void driver_argument_list (const int analysis_id)

  set argList for execution of the specified analysis driver

Private Attributes

- std::vector< std::string > argList

  an array of strings for use with execvp(const char ∗, char ∗ const ∗). These are converted to an array of const char∗'s in fork_program().

43.41.1 Detailed Description

Derived class in the AnalysisCode class hierarchy which spawns simulations using forks. ForkAnalysisCode creates a copy of the parent DAKOTA process using fork()/vfork() and then replaces the copy with a simulation process using execvp(). The parent process can then use waitpid() to wait on completion of the simulation process.

43.41.2 Member Function Documentation

43.41.2.1 void check_status (const int *status*)

check the exit status of a forked process and abort if an error code was returned Check to see if the process terminated abnormally (WIFEXITED(status)==0) or if either execvp or the application returned a status code of -1 (WIFEXITED(status)!=0 && (signed char)WEXITSTATUS(status)==-1). If one of these conditions is detected, output a failure message and abort. Note: the application code should not return a status code of -1 unless an immediate abort of dakota is wanted. If for instance, failure capturing is to be used, the application code should write the word "FAIL" to the appropriate results file and return a status code of 0 through exit().

References Dakota::abort_handler().

Referenced by ForkApplicInterface::asynchronous_local_analyses(), ForkApplicInterface::derived_synch(), ForkApplicInterface::derived_synch_nowait(), ForkAnalysisCode::fork_program(), and ForkApplicInterface::serve_analyses_asynch().

The documentation for this class was generated from the following files:

- ForkAnalysisCode.H
- ForkAnalysisCode.C

43.42 ForkApplicInterface Class Reference

Derived application interface class which spawns simulation codes using forks. Inheritance diagram for ForkApplicInterface::

Public Member Functions

- ForkApplicInterface (const ProblemDescDB &problem_db)

  constructor

- ~ForkApplicInterface ()

  destructor

- void derived_map (const Variables &vars, const ActiveSet &set, Response &response, int fn_eval_id)

  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- void derived_map_asynch (const ParamResponsePair &pair)

  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- void derived_synch (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- void derived_synch_nowait (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- int derived_synchronous_local_analysis (const int &analysis_id)
- const StringArray & analysis_drivers () const

  retrieve the analysis drivers specification for application interfaces

- const AnalysisCode * analysis_code () const

  return AnalysisCode::fileNameMap when defined for derived Interface class

Private Member Functions

- void derived_synch_kernel (PRPQueue &prp_queue, const pid_t pid)

  Convenience function for common code between derived_synch() & derived_synch_nowait().

- pid_t fork_application (const bool block_flag)

  perform the complete function evaluation by managing the input filter, analysis programs, and output filter

- void asynchronous_local_analyses (const int &start, const int &end, const int &step)

  execute analyses asynchronously on the local processor

- void synchronous_local_analyses (const int &start, const int &end, const int &step)

  execute analyses synchronously on the local processor

- void serve_analyses_asynch ()

  serve the analysis scheduler and execute analysis jobs asynchronously

Private Attributes

- ForkAnalysisCode forkSimulator

  ForkAnalysisCode provides convenience functions for forking individual programs and checking fork exit status.

- std::map< pid_t, int > processIdMap

  map of fork process id's to function evaluation id's for asynchronous evaluations

43.42.1 Detailed Description

Derived application interface class which spawns simulation codes using forks. ForkApplicInterface uses a ForkAnalysisCode object for performing simulation invocations.

43.42.2 Member Function Documentation

43.42.2.1 int derived_synchronous_local_analysis (const int & *analysis_id*) `[inline, virtual]`

This code provides the derived function used by ApplicationInterface:: serve_analyses_synch() as well as a convenience function for ForkApplicInterface::synchronous_local_analyses() below.

Reimplemented from ApplicationInterface.

References ForkAnalysisCode::driver_argument_list(), ForkAnalysisCode::fork_program(), and ForkApplicInterface::forkSimulator.

Referenced by ForkApplicInterface::synchronous_local_analyses().

43.42.2.2 pid_t fork_application (const bool *block_flag*) [private]

perform the complete function evaluation by managing the input filter, analysis programs, and output filter Manage the input filter, 1 or more analysis programs, and the output filter in blocking or nonblocking mode as governed by block_flag. In the case of a single analysis and no filters, a single fork is performed, while in other cases, an initial fork is reforked multiple times. Called from derived_map() with block_flag == BLOCK and from derived_map_-asynch() with block_flag == FALL_THROUGH. Uses ForkAnalysisCode::fork_program() to spawn individual program components within the function evaluation.

References Dakota::abort_handler(), ApplicationInterface::analysisServerId, Application-Interface::asynchLocalAnalysisConcurrency, ApplicationInterface::asynchLocalAnalysisFlag, ForkApplicInterface::asynchronous_local_analyses(), ParallelLibrary::barrier_e(), AnalysisCode::command_-line_arguments(), ForkAnalysisCode::driver_argument_list(), ApplicationInterface::eaDedMasterFlag, Applicationinterface::evalCommRank, ApplicationInterface::evalCommSize, ForkAnalysisCode::fork_program(), ForkApplicInterface::forkSimulator, ForkAnalysisCode::ifilter_argument_list(), AnalysisCode::input_filter_-name(), AnalysisCode::multiple_parameters_filenames(), ApplicationInterface::numAnalysisDrivers, Applicationinterface::numAnalysisServers, ForkAnalysisCode::ofilter_argument_list(), AnalysisCode::output_-filter_name(), Interface::outputLevel, ApplicationInterface::parallelLib, AnalysisCode::parameters_-filename(), AnalysisCode::program_names(), AnalysisCode::results_filename(), ApplicationInterface::self_-schedule_analyses(), ForkApplicInterface::serve_analyses_asynch(), ApplicationInterface::serve_-analyses_synch(), AnalysisCode::suppress_output_flag(), ApplicationInterface::suppressOutput, and ForkApplicInterface::synchronous_local_analyses().

Referenced by ForkApplicInterface::derived_map(), and ForkApplicInterface::derived_map_asynch().

43.42.2.3 void asynchronous_local_analyses (const int & *start*, const int & *end*, const int & *step*) [private]

execute analyses asynchronously on the local processor Schedule analyses asynchronously on the local processor using a self-scheduling approach (start to end in step increments). Concurrency is limited by asynchLocalAnalysisConcurrency. Modeled after ApplicationInterface::asynchronous_local_evaluations(). NOTE: This function should be elevated to ApplicationInterface if and when another derived interface class supports asynchronous local analyses.

References Dakota::abort_handler(), ApplicationInterface::asynchLocalAnalysisConcurrency, ForkAnalysisCode::check_status(), ForkAnalysisCode::driver_argument_list(), Dakota::find_-index(), ForkAnalysisCode::fork_program(), ForkApplicInterface::forkSimulator, ApplicationInterface::numAnalysisDrivers, and ForkApplicInterface::processIdMap.

Referenced by ForkApplicInterface::fork_application().

43.42.2.4 void synchronous_local_analyses (const int & *start*, const int & *end*, const int & *step*) [inline, private]

execute analyses synchronously on the local processor Execute analyses synchronously in succession on the local processor (start to end in step increments). Modeled after ApplicationInterface::synchronous_local_evaluations().

References ForkApplicInterface::derived_synchronous_local_analysis().

Referenced by ForkApplicInterface::fork_application().

43.42.2.5 void serve_analyses_asynch () [private]

serve the analysis scheduler and execute analysis jobs asynchronously This code runs multiple asynch analyses on each server. It is modeled after ApplicationInterface::serve_evaluations_asynch(). NOTE: This fn should be elevated to ApplicationInterface if and when another derived interface class supports hybrid analysis parallelism.

References Dakota::abort_handler(), ApplicationInterface::asynchLocalAnalysisConcurrency, ForkAnalysisCode::check_status(), ForkAnalysisCode::driver_argument_list(), Dakota::find_index(), ForkAnalysisCode::fork_program(), ForkApplicInterface::forkSimulator, ParallelLibrary::irecv_ea(), ApplicationInterface::numAnalysisDrivers, ApplicationInterface::parallelLib, ForkApplicInterface::processIdMap, ParallelLibrary::recv_ea(), ParallelLibrary::send_ea(), and ParallelLibrary::test().

Referenced by ForkApplicInterface::fork_application().

The documentation for this class was generated from the following files:

- ForkApplicInterface.H
- ForkApplicInterface.C

43.43 FSUDesignCompExp Class Reference

Wrapper class for the FSUDace QMC/CVT library. Inheritance diagram for FSUDesignCompExp::

Public Member Functions

- FSUDesignCompExp (Model &model)

  primary constructor for building a standard DACE iterator

- FSUDesignCompExp (Model &model, int samples, int seed, const String &sampling_method)

  alternate constructor for building a DACE iterator on-the-fly

- ~FSUDesignCompExp ()

  destructor

- void pre_run ()

  pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori

- void extract_trends ()

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

- void post_input ()

  read tabular data for post-run mode

- void post_run (std::ostream &s)

  post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way

- int num_samples () const

  get the current number of samples

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)

  reset sampling iterator to use at least min_samples

- const String & sampling_scheme () const

  return sampling name

- void vary_pattern (bool pattern_flag)

  sets varyPattern in derived classes that support it

- void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

Private Member Functions

- void enforce_input_rules ()

  enforce sanity checks/modifications for the user input specification

Private Attributes

- int samplesSpec

  initial specification of number of samples

- int numSamples

  current number of samples to be evaluated

- bool allDataFlag

  flag which triggers the update of allVars/allResponses for use by Iterator::all_variables() and Iterator::all_responses()

- size_t numDACERuns

  counter for number of run() executions for this object

- bool latinizeFlag

  flag which specifies latinization of QMC or CVT sample sets

- IntVector sequenceStart

  Integer vector defining a starting index into the sequence for random variable sampled. Default is 0 0 0 (e.g. for three random variables).

- IntVector sequenceLeap

  Integer vector defining the leap number for each sequence being generated. Default is 1 1 1 (e.g. for three random vars.).

- IntVector primeBase

  Integer vector defining the prime base for each sequence being generated. Default is 2 3 5 (e.g., for three random vars.).

- int seedSpec

  the user seed specification for the random number generator (allows repeatable results)

- int randomSeed

  current seed for the random number generator

- bool varyPattern

  flag for continuing the random number or QMC sequence from a previous run() execution (e.g., for surrogate-based optimization) so that multiple executions are repeatable but not identical.

- int numCVTTrials

  specifies the number of sample points taken at internal CVT iteration

- int trialType

  Trial type in CVT. Specifies where the points are placed for consideration relative to the centroids. Choices are grid (2), halton (1), uniform (0), or random (-1). Default is random.

43.43.1 Detailed Description

Wrapper class for the FSUDace QMC/CVT library. The FSUDesignCompExp class provides a wrapper for FSUDace, a C++ design of experiments library from Florida State University. This class uses quasi Monte Carlo (QMC) and Centroidal Voronoi Tesselation (CVT) methods to uniformly sample the parameter space spanned by the active bounds of the current Model. It returns all generated samples and their corresponding responses as well as the best sample found.

43.43.2 Constructor & Destructor Documentation

43.43.2.1 FSUDesignCompExp (Model & *model*)

primary constructor for building a standard DACE iterator This constructor is called for a standard iterator built with data from probDescDB.

References Dakota::abort_handler(), ProblemDescDB::get_bool(), ProblemDescDB::get_idv(), ProblemDescDB::get_int(), ProblemDescDB::get_string(), Iterator::maxConcurrency, Iterator::methodName, Iterator::numContinuousVars, FSUDesignCompExp::numCVTTrials, FSUDesignCompExp::numSamples, FSUDesignCompExp::primeBase, Iterator::probDescDB, FSUDesignCompExp::randomSeed, FSUDesignCompExp::seedSpec, FSUDesignCompExp::sequenceLeap, FSUDesignCompExp::sequenceStart, FSUDesignCompExp::trialType, and FSUDesignCompExp::varyPattern.

43.43.2.2 FSUDesignCompExp (Model & *model*, int *samples*, int *seed*, const String & *sampling_method*)

alternate constructor for building a DACE iterator on-the-fly This alternate constructor is used for instantiations on-the-fly, using only the incoming data. No problem description database queries are used.

References Dakota::abort_handler(), Iterator::maxConcurrency, Iterator::methodName, Iterator::numContinuousVars, FSUDesignCompExp::numCVTTrials, FSUDesignCompExp::numSamples, FSUDesignCompExp::primeBase, FSUDesignCompExp::randomSeed, FSUDesignCompExp::seedSpec, FSUDesignCompExp::sequenceLeap, FSUDesignCompExp::sequenceStart, and FSUDesignCompExp::trialType.

43.43.3 Member Function Documentation

43.43.3.1 void pre_run () [virtual]

pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori pre-run phase, which a derived iterator may optionally reimplement; when not present, pre-run is likely integrated into the derived run function. This is a virtual function; when re-implementing, a derived class must call its nearest parent's pre_run(), if implemented, typically _before_ performing its own implementation steps.

Reimplemented from Iterator.

References FSUDesignCompExp::get_parameter_sets(), Iterator::iteratedModel, and PStudyDACE::varBasedDecompFlag.

43.43.3.2 void post_run (std::ostream & *s*) [virtual]

post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way Post-run phase, which a derived iterator may optionally reimplement; when not present, post-run is likely integrated into run. This is a virtual function; when re-implementing, a derived class must call its nearest parent's post_run(), typically _after_ performing its own implementation steps.

Reimplemented from Iterator.

References Analyzer::allResponses, Analyzer::allSamples, SensAnalysisGlobal::compute_correlations(), PStudyDACE::pStudyDACESensGlobal, Iterator::subIteratorFlag, and PStudyDACE::varBasedDecompFlag.

43.43.3.3 int num_samples () const [inline, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References FSUDesignCompExp::numSamples.

43.43.3.4 void enforce_input_rules () [private]

enforce sanity checks/modifications for the user input specification Users may input a variety of quantities, but this function must enforce any restrictions imposed by the sampling algorithms.

References Dakota::abort_handler(), Iterator::methodName, Iterator::numContinuousVars, FSUDesignCompExp::numSamples, and FSUDesignCompExp::primeBase.

Referenced by FSUDesignCompExp::get_parameter_sets().

The documentation for this class was generated from the following files:

- FSUDesignCompExp.H
- FSUDesignCompExp.C

43.44 GaussProcApproximation Class Reference

Derived approximation class for Gaussian Process implementation. Inheritance diagram for GaussProcApproximation::

Public Member Functions

- GaussProcApproximation ()
  default constructor
- GaussProcApproximation (size_t num_vars, short data_order)
  alternate constructor
- GaussProcApproximation (const ProblemDescDB &problem_db, const size_t &num_acv)
  standard constructor
- ~GaussProcApproximation ()
  destructor

Protected Member Functions

- int min_coefficients () const
  return the minimum number of samples (unknowns) required to build the derived class approximation type in numVars dimensions
- int num_constraints () const
  return the number of constraints to be enforced via an anchor point
- void build ()
  find the covariance parameters governing the Gaussian process response
- Real get_value (const RealVector &x)
  retrieve the function value for a given parameter set x
- Real get_prediction_variance (const RealVector &x)
  retrieve the variance of the predicted value for a given parameter set x
- const RealVector & get_gradient (const RealVector &x)
  retrieve the function gradient at the predicted value for a given parameter set x

Private Member Functions

- void GPmodel_build ()

  Function to compute hyperparameters governing the GP.

- void GPmodel_apply (const RealVector &new_x, bool variance_flag, bool gradients_flag)

  Function returns a response value using the GP surface.

- void normalize_training_data ()

  Normalizes the initial inputs upon which the GP surface is based.

- void get_trend ()

  Gets the trend (basis) functions for the calculation of the mean of the GP If the order = 0, the trend is a constant, if the order = 1, trend is linear, if order = 2, trend is quadratic.

- void get_beta_coefficients ()

  Gets the beta coefficients for the calculation of the mean of the GP.

- int get_cholesky_factor ()

  Gets the Cholesky factorization of the covariance matrix, with error checking.

- void get_process_variance ()

  Gets the estimate of the process variance given the values of beta and the correlation lengthscales.

- void get_cov_matrix ()

  calculates the covariance matrix for a given set of input points

- void get_cov_vector ()

  calculates the covariance vector between a new point x and the set of inputs upon which the GP is based

- void optimize_theta_global ()

  sets up and performs the optimization of the negative log likelihood to determine the optimal values of the covariance parameters using NCSUDirect

- void optimize_theta_multipoint ()

  sets up and performs the optimization of the negative log likelihood to determine the optimal values of the covariance parameters using a gradient-based solver and multiple starting points

- void predict (bool variance_flag, bool gradients_flag)

  Calculates the predicted new response value for x in normalized space.

- Real calc_nll ()

  calculates the negative log likelihood function (based on covariance matrix)

- void calc_grad_nll ()

  Gets the gradient of the negative log likelihood function with respect to the correlation lengthscales, theta.

- void get_grad_cov_vector ()

  Calculate the derivatives of the covariance vector, with respect to each componeent of x.

- void run_point_selection ()

  Runs the point selection algorithm, which will choose a subset of the training set with which to construct the GP model, and estimate the necessary parameters.

- void initialize_point_selection ()

  Initializes the point selection routine by choosing a small initial subset of the training points.

- void pointsel_get_errors (RealArray &delta)

  Uses the current GP model to compute predictions at all of the training points and find the errors.

- int addpoint (int, IntArray &added_index)

  Adds a point to the effective training set. Returns 1 on success.

- int pointsel_add_sel (const RealArray &delta)

  Accepts a vector of unsorted prediction errors, determines which points should be added to the effective training set, and adds them.

- Real maxval (const RealArray &) const

  Return the maximum value of the elements in a vector.

- void pointsel_write_points ()

  Writes out the training set before and after point selection.

- void lhood_2d_grid_eval ()

  For problems with 2D input, evaluates the negative log likelihood on a grid.

- void writex (const char[ ])

  Writes out the current training set (in original units) to a specified file.

- void writeCovMat (char[ ])

  Writes out the covariance matrix to a specified file.

Static Private Member Functions

- static void negloglik (int mode, int n, const Teuchos::SerialDenseVector< int, double > &X, Real &fx, Teuchos::SerialDenseVector< int, double > &grad_x, int &result_mode)

  static function used by OPT++ as the objective function to optimize the hyperparameters in the covariance of the GP by minimizing the negative log likelihood

- static void constraint_eval (int mode, int n, const Teuchos::SerialDenseVector< int, double > &X, Teuchos::SerialDenseVector< int, double > &g, Teuchos::SerialDenseMatrix< int, double > &gradC, int &result_mode)

static function used by OPT++ as the constraint function in the optimization of the negative log likelihood. Currently this function is empty: it is an unconstrained optimization.

- static double negloglikNCSU (const RealVector &x)

  function used by NCSUOptimizer to optimize negloglik objective

Private Attributes

- Real approxValue

  value of the approximation returned by get_value()

- Real approxVariance

  value of the approximation returned by get_prediction_variance()

- RealMatrix trainPoints

  A 2-D array (num sample sites = rows, num vars = columns) used to create the Gaussian process.

- RealMatrix trainValues

  An array of response values; one response value per sample site.

- RealVector trainMeans

  The mean of the input columns of trainPoints.

- RealVector trainStdvs

  The standard deviation of the input columns of trainPoints.

- RealMatrix normTrainPoints

  Current working set of normalized points upon which the GP is based.

- RealMatrix trendFunction

  matrix to hold the trend function

- RealMatrix betaCoeffs

  matrix to hold the beta coefficients for the trend function

- RealSymMatrix covMatrix

  The covariance matrix where each element (i,j) is the covariance between points Xi and Xj in the initial set of samples.

- RealMatrix covVector

  The covariance vector where each element (j,0) is the covariance between a new point X and point Xj from the initial set of samples.

- RealMatrix approxPoint

  Point at which a prediction is requested. This is currently a single point, but it could be generalized to be a vector of points.

- RealMatrix gradNegLogLikTheta

  matrix to hold the gradient of the negative log likelihood with respect to the theta correlation terms

- Teuchos::SerialSpdDenseSolver< int, Real > covSlvr

  The global solver for all computations involving the inverse of the covariance matrix.

- RealMatrix gradCovVector

  A matrix, where each column is the derivative of the covVector with respect to a particular componenet of X.

- RealMatrix normTrainPointsAll

  Set of all original samples available.

- RealMatrix trainValuesAll

  All original samples available.

- RealMatrix trendFunctionAll

  Trend function values corresponding to all original samples.

- RealMatrix Rinv_YFb

  Matrix for storing inverse of correlation matrix Rinv∗(Y-FB).

- size_t numObs

  The number of observations on which the GP surface is built.

- size_t numObsAll

  The original number of observations.

- short trendOrder

  The number of variables in each X variable (number of dimensions of the problem).

- RealVector thetaParams

  Theta is the vector of covariance parameters for the GP. We determine the values of theta by optimization Currently, the covariance function is theta[0]∗exp(-0.5∗sume)+delta∗pow(sige,2). sume is the sum squared of weighted distances; it involves a sum of theta[1](Xi(1)-Xj(1))^2 + theta[2](Xi(2)-Xj(2))^2 + ... where Xi(1) is the first dimension value of multi-dimensional variable Xi. delta∗pow(sige,2) is a jitter term used to improve matrix computations. delta is zero for the covariance between different points and 1 for the covariance between the same point. sige is the underlying process error.

- Real procVar

  The process variance, the multiplier of the correlation matrix.

- IntArray pointsAddedIndex

  Used by the point selection algorithm, this vector keeps track all points which have been added.

- int cholFlag

  A global indicator for success of the Cholesky factorization.

- bool usePointSelection

  a flag to indicate the use of point selection

Static Private Attributes

- static GaussProcApproximation * GPinstance

  pointer to the active object instance used within the static evaluator

43.44.1 Detailed Description

Derived approximation class for Gaussian Process implementation. The GaussProcApproximation class provides a global approximation (surrogate) based on a Gaussian process. The Gaussian process is built after normalizing the function values, with zero mean. Opt++ is used to determine the optimal values of the covariance parameters, those which minimize the negative log likelihood function.

43.44.2 Constructor & Destructor Documentation

43.44.2.1 GaussProcApproximation () [inline]

default constructor alternate constructor used by EffGlobalOptimization and NonDGlobalReliability that does not use a problem database defaults here are no point selectinn and quadratic trend function.

43.44.3 Member Function Documentation

43.44.3.1 void GPmodel_apply (const RealVector & *new_x*, bool *variance_flag*, bool *gradients_flag*) [private]

Function returns a response value using the GP surface. The response value is computed at the design point specified by the RealVector function argument.

References Dakota::abort_handler(), GaussProcApproximation::approxPoint, GaussProcApproximation::get_cov_vector(), Approximation::numVars, GaussProcApproximation::predict(), GaussProcApproximation::trainMeans, and GaussProcApproximation::trainStdvs.

Referenced by GaussProcApproximation::get_gradient(), GaussProcApproximation::get_prediction_variance(), GaussProcApproximation::get_value(), and GaussProcApproximation::pointsel_get_errors().

43.44.4 Member Data Documentation

43.44.4.1 short trendOrder [private]

The number of variables in each X variable (number of dimensions of the problem). The order of the basis function for the mean of the GP If the order = 0, the trend is a constant, if the order = 1, trend is linear, if order =

2, trend is quadratic.

Referenced by GaussProcApproximation::GaussProcApproximation(), GaussProcApproximation::get_beta_coefficients(), GaussProcApproximation::get_trend(), GaussProcApproximation::GPmodel_build(), and GaussProcApproximation::predict().

The documentation for this class was generated from the following files:

- GaussProcApproximation.H
- GaussProcApproximation.C

43.45 GetLongOpt Class Reference

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France). Inheritance diagram for GetLongOpt::

Public Types

- enum OptType { Valueless, OptionalValue, MandatoryValue }

  enum for different types of values associated with command line options.

Public Member Functions

- GetLongOpt (const char optmark= '-')

  Constructor.

- ~GetLongOpt ()

  Destructor.

- int parse (int argc, char *const *argv)

  parse the command line args (argc, argv).

- int parse (char *const str, char *const p)

  parse a string of options (typically given from the environment).

- int enroll (const char *const opt, const OptType t, const char *const desc, const char *const val)

  Add an option to the list of valid command options.

- const char * retrieve (const char *const opt) const

  Retrieve value of option.

- void usage (std::ostream &outfile=Cout) const

  Print usage information to outfile.

- void usage (const char *str)

  Change header of usage output to str.

- void store (const char *name, const char *value)

  Store a specified option value.

Private Member Functions

- char ∗ basename (char ∗const p) const

  extract the base name from a string as delimited by '/'

- int setcell (Cell ∗c, char ∗valtoken, char ∗nexttoken, const char ∗p)

  internal convenience function for setting Cell::value

Private Attributes

- Cell ∗ table

  option table

- const char ∗ ustring

  usage message

- char ∗ pname

  program basename

- char optmarker

  option marker

- int enroll_done

  finished enrolling

- Cell ∗ last

  last entry in option table

43.45.1 Detailed Description

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France). GetLongOpt manages the definition and parsing of "long options." Command line options can be abbreviated as long as there is no ambiguity. If an option requires a value, the value should be separated from the option either by whitespace or an "=".

43.45.2 Member Enumeration Documentation

43.45.2.1 enum OptType

enum for different types of values associated with command line options.

Enumerator:

Valueless option that may never have a value

OptionalValue option with optional value

MandatoryValue option with required value

43.45.3 Constructor & Destructor Documentation

43.45.3.1 GetLongOpt (const char *optmark* = '-')

Constructor. Constructor for GetLongOpt takes an optional argument: the option marker. If unspecified, this defaults to '-', the standard (?) Unix option marker.

References GetLongOpt::enroll_done, GetLongOpt::last, GetLongOpt::optmarker, GetLongOpt::table, and GetLongOpt::ustring.

43.45.4 Member Function Documentation

43.45.4.1 int parse (int *argc*, char *const * *argv*)

parse the command line args (argc, argv). A return value < 1 represents a parse error. Appropriate error messages are printed when errors are seen. parse returns the the optind (see getopt(3)) if parsing is successful.

References GetLongOpt::basename(), GetLongOpt::enroll_done, GetLongOpt::optmarker, GetLongOpt::pname, GetLongOpt::setcell(), and GetLongOpt::table.

Referenced by CommandLineHandler::check_usage().

43.45.4.2 int parse (char *const *str*, char *const *p*)

parse a string of options (typically given from the environment). A return value < 1 represents a parse error. Appropriate error messages are printed when errors are seen. parse takes two strings: the first one is the string to be parsed and the second one is a string to be prefixed to the parse errors.

References GetLongOpt::enroll_done, GetLongOpt::optmarker, GetLongOpt::setcell(), and GetLongOpt::table.

43.45.4.3 int enroll (const char *const *opt*, const OptType *t*, const char *const *desc*, const char *const *val*)

Add an option to the list of valid command options. enroll adds option specifications to its internal database. The first argument is the option sting. The second is an enum saying if the option is a flag (Valueless), if it requires a mandatory value (MandatoryValue) or if it takes an optional value (OptionalValue). The third argument is a string giving a brief description of the option. This description will be used by GetLongOpt::usage. GetLongOpt, for usage-printing, uses {$val} to represent values needed by the options. {< $val>="">} is a mandatory value and {[$val]} is an optional value. The final argument to enroll is the default string to be returned if the option is not specified. For flags (options with Valueless), use "" (empty string, or in fact any arbitrary string) for specifying TRUE and 0 (null pointer) to specify FALSE.

References GetLongOpt::enroll_done, GetLongOpt::last, and GetLongOpt::table.

Referenced by CommandLineHandler::initialize_options().

43.45.4.4 const char ∗ retrieve (const char ∗const *opt*) const

Retrieve value of option. The values of the options that are enrolled in the database can be retrieved using retrieve. This returns a string and this string should be converted to whatever type you want. See atoi, atof, atol, etc. If a "parse" is not done before retrieving all you will get are the default values you gave while enrolling! Ambiguities while retrieving (may happen when options are abbreviated) are resolved by taking the matching option that was enrolled last. For example, -{v} will expand to {-verify}. If you try to retrieve something you didn't enroll, you will get a warning message.

References GetLongOpt::optmarker, and GetLongOpt::table.

Referenced by CommandLineHandler::check_usage(), CommandLineHandler::instantiate_flag(), main(), ProblemDescDB::manage_inputs(), ParallelLibrary::manage_run_modes(), CommandLineHandler::read_restart_evals(), and ParallelLibrary::specify_outputs_restart().

43.45.4.5 void usage (const char ∗ *str*) `[inline]`

Change header of usage output to str. GetLongOpt::usage is overloaded. If passed a string "str", it sets the internal usage string to "str". Otherwise it simply prints the command usage.

References GetLongOpt::ustring.

The documentation for this class was generated from the following files:

- CommandLineHandler.H
- CommandLineHandler.C

43.46 Graphics Class Reference

The Graphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.

Public Member Functions

- Graphics ()

  constructor

- ~Graphics ()

  destructor

- void create_plots_2d (const Variables &vars, const Response &response)

  creates the 2d graphics window and initializes the plots

- void create_tabular_datastream (const Variables &vars, const Response &response, const std::string &tabular_data_file)

  opens the tabular data file stream and prints the headings

- void add_datapoint (const Variables &vars, const Response &response)

  adds data to each window in the 2d graphics and adds a row to the tabular data file based on the results of a model evaluation

- void add_datapoint (int i, double x, double y)

  adds data to a single window in the 2d graphics

- void new_dataset (int i)

  creates a separate line graphic for subsequent data points for a single window in the 2d graphics

- void close ()

  close graphics windows and tabular datastream

- void set_x_labels2d (const char *x_label)

  set x label for each plot equal to x_label

- void set_y_labels2d (const char *y_label)

  set y label for each plot equal to y_label

- void set_x_label2d (int i, const char *x_label)

  set x label for ith plot equal to x_label

- void set_y_label2d (int i, const char *y_label)

  set y label for ith plot equal to y_label

- void graphics_counter (int cntr)

set graphicsCntr equal to cntr

- int graphics_counter () const

  return graphicsCntr

- void tabular_counter_label (const std::string &label)

  set tabularCntrLabel equal to label

Private Attributes

- Graphics2D * graphics2D

  pointer to the 2D graphics object

- bool win2dOn

  flag to indicate if 2D graphics window is active

- bool tabularDataFlag

  flag to indicate if tabular data stream is active

- int graphicsCntr

  used for x axis values in 2D graphics and for 1st column in tabular data

- std::string tabularCntrLabel

  label for counter used in first line comment w/i the tabular data file

- std::ofstream tabularDataFStream

  file stream for tabulation of graphics data within compute_response

43.46.1 Detailed Description

The Graphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc. There is only one Graphics object (dakotaGraphics) and it is global (for convenient access from strategies, models, and approximations).

43.46.2 Member Function Documentation

43.46.2.1 void create_plots_2d (const Variables & *vars*, const Response & *response*)

creates the 2d graphics window and initializes the plots Sets up a single event loop for duration of the dakotaGraphics object, continuously adding data to a single window. There is no reset. To start over with a new data set, you need a new object (delete old and instantiate new).

References Variables::continuous_variable_labels(), Variables::cv(), Variables::discrete_int_variable_labels(), Variables::discrete_real_variable_labels(), Variables::div(), Variables::drv(), Response::function_labels(), Graphics::graphics2D, Response::num_functions(), Dakota::re_match(), and Graphics::win2dOn.

Referenced by SurrBasedMinimizer::initialize_graphics(), NonDReliability::initialize_graphics(), and Iterator::initialize_graphics().

43.46.2.2 void create_tabular_datastream (const Variables & *vars*, const Response & *response*, const std::string & *tabular_data_file*)

opens the tabular data file stream and prints the headings Opens the tabular data file stream and prints headings, one for each continuous and discrete variable and one for each response function, using the variable and response function labels. This tabular data is used for post-processing of DAKOTA results in Matlab, Tecplot, etc.

References Graphics::tabularCntrLabel, Graphics::tabularDataFlag, and Graphics::tabularDataFStream.

Referenced by SurrBasedMinimizer::initialize_graphics(), and Iterator::initialize_graphics().

43.46.2.3 void add_datapoint (const Variables & *vars*, const Response & *response*)

adds data to each window in the 2d graphics and adds a row to the tabular data file based on the results of a model evaluation Adds data to each 2d plot and each tabular data column (one for each active variable and for each response function). graphicsCntr is used for the x axis in the graphics and the first column in the tabular data.

References Response::active_set_request_vector(), Variables::continuous_variables(), Variables::discrete_int_variables(), Variables::discrete_real_variables(), Response::function_values(), Graphics::graphics2D, Graphics::graphicsCntr, Graphics::tabularDataFlag, Graphics::tabularDataFStream, Graphics::win2dOn, and Dakota::write_data_tabular().

Referenced by Model::compute_response(), NonDLocalReliability::mean_value(), SurrBasedLocalMinimizer::minimize_surrogates(), Model::synchronize(), Model::synchronize_nowait(), and NonDLocalReliability::update_level_data().

43.46.2.4 void add_datapoint (int *i*, double *x*, double *y*)

adds data to a single window in the 2d graphics Adds data to a single 2d plot. Allows complete flexibility in defining other kinds of x-y plotting in the 2D graphics.

References Graphics::graphics2D, and Graphics::win2dOn.

43.46.2.5 void new_dataset (int *i*)

creates a separate line graphic for subsequent data points for a single window in the 2d graphics Used for displaying multiple data sets within the same plot.

References Graphics::graphics2D, and Graphics::win2dOn.

Referenced by NonDLocalReliability::update_level_data().

The documentation for this class was generated from the following files:

- DakotaGraphics.H

- DakotaGraphics.C

43.47 GridApplicInterface Class Reference

Derived application interface class which spawns simulation codes using grid services such as Condor or Globus. Inheritance diagram for GridApplicInterface::

Public Member Functions

- GridApplicInterface (const ProblemDescDB &problem_db)

  constructor

- ~GridApplicInterface ()

  destructor

- void derived_map (const Variables &vars, const ActiveSet &set, Response &response, int fn_eval_id)

  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- void derived_map_asynch (const ParamResponsePair &pair)

  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- void derived_synch (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- void derived_synch_nowait (PRPQueue &prp_queue)

  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- int derived_synchronous_local_analysis (const int &analysis_id)
- const StringArray & analysis_drivers () const

  retrieve the analysis drivers specification for application interfaces

- const AnalysisCode * analysis_code () const

  return AnalysisCode::fileNameMap when defined for derived Interface class

Public Attributes

- SysCallAnalysisCode code

  Used to read/write parameter files and responses.

Protected Member Functions

- void derived_synch_kernel (PRPQueue &prp_queue)

  Convenience function for common code between derived_synch() & derived_synch_nowait().

- bool grid_file_test (const String &root_file)

  test file(s) for existence based on root_file name

Protected Attributes

- IntSet idSet

  Set of function evaluation id's for active asynchronous system call evaluations.

- IntShortMap failCountMap

  map linking function evaluation id's to number of response read failures

- start_grid_computing_t start_grid_computing

  handle to dynamically linked start_grid_computing function

- perform_analysis_t perform_analysis

  handle to dynamically linked perform_analysis grid function

- get_jobs_completed_t get_jobs_completed

  handle to dynamically linked get_jobs_completed grid function

- stop_grid_computing_t stop_grid_computing

  handle to dynamically linked stop_grid_computing function

43.47.1 Detailed Description

Derived application interface class which spawns simulation codes using grid services such as Condor or Globus. This class is currently a modified copy of SysCallApplicInterface adapted for use with an external grid dervices library which was dynamically linked using dlopen() services.

43.47.2 Member Function Documentation

43.47.2.1 int derived_synchronous_local_analysis (const int & *analysis_id*) `[inline, virtual]`

This code provides the derived function used by ApplicationInterface::serve_analyses_synch().

TODO - allow local analyses?????

Reimplemented from ApplicationInterface.

References GridApplicInterface::code, and SysCallAnalysisCode::spawn_analysis().

43.47.2.2 void derived_synch_kernel (PRPQueue & *prp_queue*) `[protected]`

Convenience function for common code between derived_synch() & derived_synch_nowait(). Convenience function for common code between wait and nowait case.

References Dakota::abort_handler(), Response::active_set(), GridApplicInterface::code, ApplicationInterface::completionSet, GridApplicInterface::failCountMap, GridApplicInterface::grid_file_test(), GridApplicInterface::idSet, Dakota::lookup_by_eval_id(), ApplicationInterface::manage_failure(), ParamResponsePair::prp_parameters(), ParamResponsePair::prp_response(), AnalysisCode::read_results_files(), and AnalysisCode::results_filename().

Referenced by GridApplicInterface::derived_synch(), and GridApplicInterface::derived_synch_nowait().

The documentation for this class was generated from the following files:

- GridApplicInterface.H
- GridApplicInterface.C

43.48 HierarchSurrModel Class Reference

Derived model class within the surrogate model branch for managing hierarchical surrogates (models of varying fidelity). Inheritance diagram for HierarchSurrModel::

Public Member Functions

- HierarchSurrModel (ProblemDescDB &problem_db)

  constructor

- ~HierarchSurrModel ()

  destructor

Protected Member Functions

- void derived_compute_response (const ActiveSet &set)

  portion of compute_response() specific to HierarchSurrModel

- void derived_asynch_compute_response (const ActiveSet &set)

  portion of asynch_compute_response() specific to HierarchSurrModel

- const IntResponseMap & derived_synchronize ()

  portion of synchronize() specific to HierarchSurrModel

- const IntResponseMap & derived_synchronize_nowait ()

  portion of synchronize_nowait() specific to HierarchSurrModel

- Model & surrogate_model ()

  return lowFidelityModel

- Model & truth_model ()

  return highFidelityModel

- void derived_subordinate_models (ModelList &ml, bool recurse_flag)

  return lowFidelityModel and highFidelityModel

- void primary_response_fn_weights (const RealVector &wts, bool recurse_flag=true)

  set the relative weightings for multiple objective functions or least squares terms and optionally recurses into LF/HF models

- void surrogate_response_mode (short mode)

  set responseMode and pass any bypass request on to highFidelityModel for any lower-level surrogate recursions.

- void surrogate_function_indices (const IntSet &surr_fn_indices)

  (re)set the surrogate index set in SurrogateModel::surrogateFnIndices

- void build_approximation ()

  use highFidelityModel to compute the truth values needed for correction of lowFidelityModel results

- void component_parallel_mode (short mode)

  update component parallel mode for supporting parallelism in lowFidelityModel and highFidelityModel

- void derived_init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set up lowFidelityModel and highFidelityModel for parallel operations

- void derived_init_serial ()

  set up lowFidelityModel and highFidelityModel for serial operations.

- void derived_set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set active parallel configuration within lowFidelityModel and highFidelityModel

- void derived_free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  deallocate communicator partitions for the HierarchSurrModel (request forwarded to lowFidelityModel and highFidelityModel)

- void serve ()

  Service lowFidelityModel and highFidelityModel job requests received from the master. Completes when a termination message is received from stop_servers().

- void stop_servers ()

  Executed by the master to terminate lowFidelityModel and highFidelityModel server operations when iteration on the HierarchSurrModel is complete.

- void inactive_view (short view, bool recurse_flag=true)

  update the Model's inactive view based on higher level (nested) context and optionally recurse into

- int evaluation_id () const

  Return the current evaluation id for the HierarchSurrModel.

- void set_evaluation_reference ()

  set the evaluation counter reference points for the HierarchSurrModel (request forwarded to lowFidelityModel and highFidelityModel)

- void fine_grained_evaluation_counters ()

  request fine-grained evaluation reporting within lowFidelityModel and highFidelityModel

- void print_evaluation_summary (std::ostream &s, bool minimal_header=false, bool relative_count=true) const

  print the evaluation summary for the HierarchSurrModel (request forwarded to lowFidelityModel and highFidelityModel)

Private Member Functions

- void update_model (Model &model)

  update the incoming model (lowFidelityModel or highFidelityModel) with current variable values/bounds/labels

Private Attributes

- int hierModelEvalCntr

  number of calls to derived_compute_response()/ derived_asynch_compute_response()

- IntResponseMap cachedTruthRespMap

  map of high-fidelity responses retrieved in derived_synchronize_nowait() that could not be returned since corresponding low-fidelity response portions were still pending.

- Model lowFidelityModel

  provides approximate low fidelity function evaluations. Model is of arbitrary type and supports recursions (e.g., lowFidelityModel can be a data fit surrogate on a low fidelity model).

- Model highFidelityModel

  provides truth evaluations for computing corrections to the low fidelity results. Model is of arbitrary type and supports recursions.

- Response highFidRefResponse

  the reference high fidelity response computed in build_approximation() and used for calculating corrections.

43.48.1 Detailed Description

Derived model class within the surrogate model branch for managing hierarchical surrogates (models of varying fidelity). The HierarchSurrModel class manages hierarchical models of varying fidelity. In particular, it uses a low fidelity model as a surrogate for a high fidelity model. The class contains a lowFidelityModel which performs the approximate low fidelity function evaluations and a highFidelityModel which provides truth evaluations for computing corrections to the low fidelity results.

43.48.2 Member Function Documentation

43.48.2.1 void derived_compute_response (const ActiveSet & *set*) [protected, virtual]

portion of compute_response() specific to HierarchSurrModel Compute the response synchronously using lowFidelityModel, highFidelityModel, or both (mixed case). For the lowFidelityModel portion, compute the high fidelity response if needed with build_approximation(), and, if correction is active, correct the low fidelity results.

Reimplemented from Model.

References Response::active_set(), DiscrepancyCorrection::apply(), SurrogateModel::approxBuilds, SurrogateModel::asv_mapping(), HierarchSurrModel::build_approximation(), HierarchSurrModel::component_parallel_mode(), DiscrepancyCorrection::compute(), Model::compute_response(), DiscrepancyCorrection::computed(), Variables::continuous_variables(), Response::copy(), Model::current_response(), Model::currentResponse, Model::currentVariables, SurrogateModel::deltaCorr, SurrogateModel::force_rebuild(), HierarchSurrModel::hierModelEvalCntr, HierarchSurrModel::highFidelityModel, HierarchSurrModel::highFidRefResponse, HierarchSurrModel::lowFidelityModel, Model::outputLevel, ActiveSet::request_vector(), SurrogateModel::response_mapping(), SurrogateModel::responseMode, Response::update(), and HierarchSurrModel::update_model().

43.48.2.2 void derived_asynch_compute_response (const ActiveSet & *set*) [protected, virtual]

portion of asynch_compute_response() specific to HierarchSurrModel Compute the response asynchronously using lowFidelityModel, highFidelityModel, or both (mixed case). For the lowFidelityModel portion, compute the high fidelity response with build_approximation() (for correcting the low fidelity results in derived_synchronize() and derived_synchronize_nowait()) if not performed previously.

Reimplemented from Model.

References SurrogateModel::approxBuilds, SurrogateModel::asv_mapping(), Model::asynch_compute_response(), HierarchSurrModel::build_approximation(), Variables::continuous_variables(), Dakota::copy_data(), Model::currentVariables, Model::evaluation_id(), SurrogateModel::force_rebuild(), HierarchSurrModel::hierModelEvalCntr, HierarchSurrModel::highFidelityModel, HierarchSurrModel::lowFidelityModel, SurrogateModel::rawCVarsMap, ActiveSet::request_vector(), SurrogateModel::responseMode, SurrogateModel::surrIdMap, SurrogateModel::truthIdMap, and HierarchSurrModel::update_model().

43.48.2.3 const IntResponseMap & derived_synchronize () [protected, virtual]

portion of synchronize() specific to HierarchSurrModel Blocking retrieval of asynchronous evaluations from lowFidelityModel, highFidelityModel, or both (mixed case). For the lowFidelityModel portion, apply correction (if active) to each response in the array. derived_synchronize() is designed for the general case where derived_asynch_compute_response() may be inconsistent in its use of low fidelity evaluations, high fidelity evaluations, or both.

Reimplemented from Model.

References DiscrepancyCorrection::apply(), SurrogateModel::cachedApproxRespMap, HierarchSurrModel::cachedTruthRespMap, HierarchSurrModel::component_parallel_mode(), DiscrepancyCorrection::compute(), DiscrepancyCorrection::computed(), SurrogateModel::deltaCorr, HierarchSurrModel::highFidelityModel, HierarchSurrModel::highFidRefResponse, HierarchSurrModel::lowFidelityModel, Model::outputLevel, SurrogateModel::rawCVarsMap, SurrogateModel::response_mapping(), Surrogate-

Model::responseMode, SurrogateModel::surrIdMap, SurrogateModel::surrResponseMap, Model::synchronize(), and SurrogateModel::truthIdMap.

43.48.2.4 const IntResponseMap & derived_synchronize_nowait () [protected, virtual]

portion of synchronize_nowait() specific to HierarchSurrModel Nonblocking retrieval of asynchronous evaluations from lowFidelityModel, highFidelityModel, or both (mixed case). For the lowFidelityModel portion, apply correction (if active) to each response in the map. derived_synchronize_nowait() is designed for the general case where derived_asynch_compute_response() may be inconsistent in its use of actual evals, approx evals, or both.

Reimplemented from Model.

References DiscrepancyCorrection::apply(), SurrogateModel::cachedApproxRespMap, HierarchSurrModel::cachedTruthRespMap, HierarchSurrModel::component_parallel_mode(), DiscrepancyCorrection::compute(), DiscrepancyCorrection::computed(), SurrogateModel::deltaCorr, HierarchSurrModel::highFidelityModel, HierarchSurrModel::highFidRefResponse, HierarchSurrModel::lowFidelityModel, Model::outputLevel, SurrogateModel::rawCVarsMap, SurrogateModel::response_mapping(), SurrogateModel::responseMode, SurrogateModel::surrIdMap, SurrogateModel::surrResponseMap, Model::synchronize_nowait(), and SurrogateModel::truthIdMap.

43.48.2.5 int evaluation_id () const [inline, protected, virtual]

Return the current evaluation id for the HierarchSurrModel. return the hierarchical model evaluation count. Due to possibly intermittent use of surrogate bypass, this is not the same as either the loFi or hiFi model evaluation counts. It also does not distinguish duplicate evals.

Reimplemented from Model.

References HierarchSurrModel::hierModelEvalCntr.

The documentation for this class was generated from the following files:

- HierarchSurrModel.H
- HierarchSurrModel.C

43.49 HybridStrategy Class Reference

Base class for hybrid minimization strategies. Inheritance diagram for HybridStrategy::

Protected Member Functions

- HybridStrategy (ProblemDescDB &problem_db)

  constructor

- ~HybridStrategy ()

  destructor

- void allocate_methods ()

  initialize selectedIterators and userDefinedModels

- void deallocate_methods ()

  free communicators for selectedIterators and userDefinedModels

Protected Attributes

- StringArray methodList

  the list of method identifiers

- int numIterators

  number of methods in methodList

- IteratorArray selectedIterators

  the set of iterators, one for each entry in methodList

- ModelArray userDefinedModels

  the set of models, one for each iterator

43.49.1 Detailed Description

Base class for hybrid minimization strategies. This base class shares code for three approaches to hybrid minimization: (1) the sequential hybrid; (2) the embedded hybrid; and (3) the collaborative hybrid.

The documentation for this class was generated from the following files:

- HybridStrategy.H
- HybridStrategy.C

43.50 Interface Class Reference

Base class for the interface class hierarchy. Inheritance diagram for Interface::

Public Member Functions

- Interface ()

  default constructor

- Interface (ProblemDescDB &problem_db)

  standard constructor for envelope

- Interface (const Interface &interface)

  copy constructor

- virtual ~Interface ()

  destructor

- Interface operator= (const Interface &interface)

  assignment operator

- virtual void map (const Variables &vars, const ActiveSet &set, Response &response, const bool asynch_-flag=false)

  the function evaluator: provides a "mapping" from the variables to the responses.

- virtual const IntResponseMap & synch ()

  recovers data from a series of asynchronous evaluations (blocking)

- virtual const IntResponseMap & synch_nowait ()

  recovers data from a series of asynchronous evaluations (nonblocking)

- virtual void serve_evaluations ()

  evaluation server function for multiprocessor executions

- virtual void stop_evaluation_servers ()

  send messages from iterator rank 0 to terminate evaluation servers

- virtual void init_communicators (const IntArray &message_lengths, const int &max_iterator_concurrency)

allocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

- virtual void set_communicators (const IntArray &message_lengths)

  set the local parallel partition data for an interface (the partitions are already allocated in ParallelLibrary).

- virtual void free_communicators ()

  deallocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

- virtual void init_serial ()

  reset certain defaults for serial interface objects.

- virtual int asynch_local_evaluation_concurrency () const

  return the user-specified concurrency for asynch local evaluations

- virtual String interface_synchronization () const

  return the user-specified interface synchronization

- virtual int minimum_points (bool constraint_flag) const

  returns the minimum number of points required to build a particular ApproximationInterface (used by DataFitSurrModels).

- virtual int recommended_points (bool constraint_flag) const

  returns the recommended number of points required to build a particular ApproximationInterface (used by DataFitSurrModels).

- virtual void approximation_function_indices (const IntSet &approx_fn_indices)

  set the (currently active) approximation function index set

- virtual void update_approximation (const Variables &vars, const IntResponsePair &response_pr)

  updates the anchor point for an approximation

- virtual void update_approximation (const RealMatrix &samples, const IntResponseMap &resp_map)

  updates the current data points for an approximation

- virtual void update_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map)

  updates the current data points for an approximation

- virtual void append_approximation (const Variables &vars, const IntResponsePair &response_pr)

  appends a single point to an existing approximation

- virtual void append_approximation (const RealMatrix &samples, const IntResponseMap &resp_map)

  appends multiple points to an existing approximation

- virtual void append_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map)

  appends multiple points to an existing approximation

- virtual void build_approximation (const RealVector &lower_bnds, const RealVector &upper_bnds)

  builds the approximation

- virtual void rebuild_approximation (const BoolDeque &rebuild_deque)

  rebuilds the approximation after a data update

- virtual void pop_approximation (bool save_surr_data)

  removes data from last append from the approximation

- virtual void restore_approximation ()

  restores the approximation to a selected previous state

- virtual bool restore_available ()

  queries the approximation for the ability to restore a previous increment

- virtual void finalize_approximation ()

  finalizes the approximation by applying all trial increments

- virtual void store_approximation ()

  move the current approximation into storage for later combination

- virtual void combine_approximation (short corr_type)

  combine the current approximation with one previously stored

- virtual void clear_current ()

  clears current data from an approximation interface

- virtual void clear_all ()

  clears all data from an approximation interface

- virtual std::vector< Approximation > & approximations ()

  retrieve the Approximations within an ApproximationInterface

- virtual const Pecos::SurrogateData & approximation_data (size_t index)

  retrieve the approximation data from a particular Approximation within an ApproximationInterface

- virtual const RealVectorArray & approximation_coefficients ()

  retrieve the approximation coefficients from each Approximation within an ApproximationInterface

- virtual void approximation_coefficients (const RealVectorArray &approx_coeffs)

  set the approximation coefficients within each Approximation within an ApproximationInterface

- virtual const RealVector & approximation_variances (const RealVector &c_variables)

  retrieve the approximation variances from each Approximation within an ApproximationInterface

- virtual const StringArray & analysis_drivers () const

  retrieve the analysis drivers specification for application interfaces

- virtual const AnalysisCode ∗ analysis_code () const

  return AnalysisCode::fileNameMap when defined for derived Interface class

- virtual bool evaluation_cache () const

  return flag indicating usage of the global evaluation cache

- void assign_rep (Interface ∗interface_rep, bool ref_count_incr=true)

  replaces existing letter with a new one

- const String & interface_type () const

  returns the interface type

- const String & interface_id () const

  returns the interface identifier

- int evaluation_id () const

  returns the value of the (total) evaluation id counter for the interface

- void fine_grained_evaluation_counters (size_t num_fns)

  set fineGrainEvalCounters to true and initialize counters if needed

- void init_evaluation_counters (size_t num_fns)

  initialize fine grained evaluation counters

- void set_evaluation_reference ()

  set evaluation count reference points for the interface

- void print_evaluation_summary (std::ostream &s, bool minimal_header, bool relative_count) const

  print an evaluation summary for the interface

- bool multi_proc_eval_flag () const

  returns a flag signaling the use of multiprocessor evaluation partitions

- bool iterator_eval_dedicated_master_flag () const

  returns a flag signaling the use of a dedicated master processor at the iterator-evaluation scheduling level

- bool is_null () const

  function to check interfaceRep (does this envelope contain a letter?)

Protected Member Functions

- Interface (BaseConstructor, const ProblemDescDB &problem_db)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- Interface (NoDBBaseConstructor, size_t num_fns, short output_level)

  constructor initializes the base class part of letter classes (NoDBBaseConstructor used for on the fly instantiations without a DB)

- void init_algebraic_mappings (const Variables &vars, const Response &response)

  Define algebraicACVIndices, algebraicACVIds, and algebraicFnIndices.

- void asv_mapping (const ActiveSet &total_set, ActiveSet &algebraic_set, ActiveSet &core_set)

  define the evaluation requirements for algebraic_mappings() (algebraic_set) and the core Application/Approximation mapping (core_set) from the total Interface evaluation requirements (total_set)

- void algebraic_mappings (const Variables &vars, const ActiveSet &algebraic_set, Response &algebraic_response)

  evaluate the algebraic_response using the AMPL solver library and the data extracted from the algebraic_mappings file

- void response_mapping (const Response &algebraic_response, const Response &core_response, Response &total_response)

  combine the response from algebraic_mappings() with the response from derived_map() to create the total response

Protected Attributes

- String interfaceType

  the interface type: system, fork, direct, grid, or approximation

- String interfaceId

  the interface specification identifier string from the DAKOTA input file

- bool algebraicMappings

  flag for the presence of algebraic_mappings that define the subset of an Interface's parameter to response mapping that is explicit and algebraic.

- bool coreMappings

  flag for the presence of non-algebraic mappings that define the core of an Interface's parameter to response mapping (using analysis_drivers for ApplicationInterface or functionSurfaces for ApproximationInterface).

- int currEvalId

  identifier for the current evaluation, which may differ from the evaluation counters in the case of evaluation scheduling; used on iterator master as well as server processors. Currently, this is set prior to all invocations of derived_map() for all processors.

- bool fineGrainEvalCounters

  controls use of fn val/grad/hess counters

- int evalIdCntr

  total interface evaluation counter

- int newEvalIdCntr

  new (non-duplicate) interface evaluation counter

- int evalIdRefPt

  iteration reference point for evalIdCntr

- int newEvalIdRefPt

  iteration reference point for newEvalIdCntr

- IntArray fnValCounter

  number of value evaluations by resp fn

- IntArray fnGradCounter

  number of gradient evaluations by resp fn

- IntArray fnHessCounter

  number of Hessian evaluations by resp fn

- IntArray newFnValCounter

  number of new value evaluations by resp fn

- IntArray newFnGradCounter

  number of new gradient evaluations by resp fn

- IntArray newFnHessCounter

  number of new Hessian evaluations by resp fn

- IntArray fnValRefPt

  iteration reference point for fnValCounter

- IntArray fnGradRefPt

  iteration reference point for fnGradCounter

- IntArray fnHessRefPt

  iteration reference point for fnHessCounter

- IntArray newFnValRefPt

  iteration reference point for newFnValCounter

- IntArray newFnGradRefPt

  iteration reference point for newFnGradCounter

- IntArray newFnHessRefPt

  iteration reference point for newFnHessCounter

- IntResponseMap rawResponseMap

  Set of responses returned after either a blocking or nonblocking schedule of asynchronous evaluations.

- StringArray fnLabels

  response function descriptors from the DAKOTA input file (used in print_evaluation_summary() and derived direct interface classes)

- bool multiProcEvalFlag

  flag for multiprocessor evaluation partitions (evalComm)

- bool ieDedMasterFlag

  flag for dedicated master partitioning at the iterator level

- short outputLevel

  output verbosity level: {SILENT,QUIET,NORMAL,VERBOSE,DEBUG}_OUTPUT

Private Member Functions

- Interface * get_interface (ProblemDescDB &problem_db)

  Used by the envelope to instantiate the correct letter class.

- int algebraic_function_type (String)

  Used by algebraic mappings to determine the correct AMPL function evaluation call to make.

Private Attributes

- StringArray algebraicVarTags

  set of variable tags from AMPL stub.col

- SizetArray algebraicACVIndices

  set of indices mapping AMPL algebraic variables to DAKOTA all continuous variables

- SizetArray algebraicACVIds

  set of ids mapping AMPL algebraic variables to DAKOTA all continuous variables

- StringArray algebraicFnTags

  set of function tags from AMPL stub.row

- IntArray algebraicFnTypes

  function type: > 0 = objective, < 0 = constraint |value|-1 is the objective (constraint) index when making AMPL objval (conival) calls

- SizetArray algebraicFnIndices

  set of indices mapping AMPL algebraic objective functions to DAKOTA response functions

- RealArray algebraicConstraintWeights

  set of weights for computing Hessian matrices for algebraic constraints;

- int numAlgebraicResponses

  number of algebraic responses (objectives+constraints)

- Interface ∗ interfaceRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing interfaceRep

- ASL ∗ asl

  pointer to an AMPL solver library (ASL) object

43.50.1 Detailed Description

Base class for the interface class hierarchy. The Interface class hierarchy provides the part of a Model that is responsible for mapping a set of Variables into a set of Responses. The mapping is performed using either a simulation-based application interface or a surrogate-based approximation interface. For memory efficiency and enhanced polymorphism, the interface hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Interface) serves as the envelope and one of the derived classes (selected in Interface::get_interface()) serves as the letter.

43.50.2 Constructor & Destructor Documentation

43.50.2.1 Interface ()

default constructor used in Model envelope class instantiations

43.50.2.2 Interface (ProblemDescDB & *problem_db*)

standard constructor for envelope Used in Model instantiation to build the envelope. This constructor only needs to extract enough data to properly execute get_interface, since Interface::Interface(BaseConstructor, problem_db) builds the actual base class data inherited by the derived interfaces.

References Dakota::abort_handler(), Interface::get_interface(), and Interface::interfaceRep.

43.50.2.3 Interface (const Interface & *interface*)

copy constructor Copy constructor manages sharing of interfaceRep and incrementing of referenceCount.

References Interface::interfaceRep, and Interface::referenceCount.

43.50.2.4 ~Interface () `[virtual]`

destructor Destructor decrements referenceCount and only deletes interfaceRep if referenceCount is zero.

References Interface::interfaceRep, and Interface::referenceCount.

43.50.2.5 Interface (BaseConstructor, const ProblemDescDB & *problem_db*) `[protected]`

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all inherited interfaces. get_interface() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_-interface() again). Since this is the letter and the letter IS the representation, interfaceRep is set to NULL (an uninitialized pointer causes problems in ~Interface).

References Dakota::abort_handler(), Interface::algebraic_function_type(), Interface::algebraicConstraintWeights, Interface::algebraicFnTags, Interface::algebraicFnTypes, Interface::algebraicMappings, Interface::algebraicVarTags, Interface::asl, Interface::coreMappings, String::data(), String::ends(), Interface::fineGrainEvalCounters, Interface::fnLabels, ProblemDescDB::get_dsa(), ProblemDescDB::get_string(), Interface::init_evaluation_counters(), and Interface::outputLevel.

43.50.3 Member Function Documentation

43.50.3.1 Interface operator= (const Interface & *interface*)

assignment operator Assignment operator decrements referenceCount for old interfaceRep, assigns new interfaceRep, and increments referenceCount for new interfaceRep.

References Interface::interfaceRep, and Interface::referenceCount.

43.50.3.2 void assign_rep (Interface * *interface_rep*, bool *ref_count_incr* = `true`)

replaces existing letter with a new one Similar to the assignment operator, the assign_rep() function decrements referenceCount for the old interfaceRep and assigns the new interfaceRep. It is different in that it is used for publishing derived class letters to existing envelopes, as opposed to sharing representations among multiple envelopes (in particular, assign_rep is passed a letter object and operator= is passed an envelope object). Letter assignment supports two models as governed by ref_count_incr:

- ref_count_incr = true (default): the incoming letter belongs to another envelope. In this case, increment the reference count in the normal manner so that deallocation of the letter is handled properly.
- ref_count_incr = false: the incoming letter is instantiated on the fly and has no envelope. This case is modeled after get_interface(): a letter is dynamically allocated using new and passed into assign_rep, the

letter's reference count is not incremented, and the letter is not remotely deleted (its memory management is passed over to the envelope).

References Dakota::abort_handler(), Interface::interfaceRep, and Interface::referenceCount.

Referenced by DataFitSurrModel::DataFitSurrModel(), model_interface_plugins(), and run_dakota().

43.50.3.3 Interface ∗ get_interface (ProblemDescDB & *problem_db*) [private]

Used by the envelope to instantiate the correct letter class. used only by the envelope constructor to initialize interfaceRep to the appropriate derived type.

References ProblemDescDB::get_string().

Referenced by Interface::Interface().

43.50.4 Member Data Documentation

43.50.4.1 IntResponseMap rawResponseMap [protected]

Set of responses returned after either a blocking or nonblocking schedule of asynchronous evaluations. The map is a full/partial set of completions which are identified through their evalIdCntr key. The raw set is postprocessed (i.e., finite diff grads merged) in Model::synchronize() where it becomes responseMap.

Referenced by ApplicationInterface::asynchronous_local_evaluations(), ApplicationInterface::asynchronous_local_evaluations_nowait(), ApplicationInterface::asynchronous_local_evaluations_static(), ApplicationInterface::self_schedule_evaluations(), ApplicationInterface::static_schedule_evaluations(), ApproximationInterface::synch(), ApplicationInterface::synch(), ApproximationInterface::synch_nowait(), ApplicationInterface::synch_nowait(), and ApplicationInterface::synchronous_local_evaluations().

The documentation for this class was generated from the following files:

- DakotaInterface.H
- DakotaInterface.C

43.51 Iterator Class Reference

Base class for the iterator class hierarchy. Inheritance diagram for Iterator::

Public Member Functions

- Iterator ()

  default constructor

- Iterator (Model &model)

  standard envelope constructor

- Iterator (const String &method_name, Model &model)

  alternate envelope constructor for instantiations by name

- Iterator (const Iterator &iterator)

  copy constructor

- virtual ~Iterator ()

  destructor

- Iterator operator= (const Iterator &iterator)

  assignment operator

- virtual void initialize_run ()

  utility function to perform common operations prior to pre_run(); typically memory initialization; setting of instance pointers

- virtual void pre_run ()

  pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori

- virtual void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- virtual void post_run (std::ostream &s)

  post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way

- virtual void finalize_run ()

  utility function to perform common operations following post_run(); deallocation and resetting of instance pointers

- virtual void reset ()

  restore initial state for repeated sub-iterator executions

- virtual const Variables & variables_results () const

  return a single final iterator solution (variables)

- virtual const Response & response_results () const

  return a single final iterator solution (response)

- virtual const VariablesArray & variables_array_results ()

  return multiple final iterator solutions (variables). This should only be used if returns_multiple_points() returns true.

- virtual const ResponseArray & response_array_results ()

  return multiple final iterator solutions (response). This should only be used if returns_multiple_points() returns true.

- virtual bool accepts_multiple_points () const

  indicates if this iterator accepts multiple initial points. Default return is false. Override to return true if appropriate.

- virtual bool returns_multiple_points () const

  indicates if this iterator returns multiple final points. Default return is false. Override to return true if appropriate.

- virtual void initial_points (const VariablesArray &pts)

  sets the multiple initial points for this iterator. This should only be used if accepts_multiple_points() returns true.

- virtual void response_results_active_set (const ActiveSet &set)

  set the requested data for the final iterator response results

- virtual void initialize_graphics (bool graph_2d, bool tabular_data, const String &tabular_file)

  initialize the 2D graphics window and the tabular graphics data

- virtual void print_results (std::ostream &s)

  print the final iterator results

- virtual int num_samples () const

  get the current number of samples

- virtual void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)

reset sampling iterator to use at least min_samples

- virtual const String & sampling_scheme () const

  return sampling name

- virtual const Model & algorithm_space_model () const

  return the result of any recasting or surrogate model recursion layered on top of iteratedModel by the derived Iterator ctor chain

- virtual String uses_method () const

  return name of any enabling iterator used by this iterator

- virtual void method_recourse ()

  perform a method switch, if possible, due to a detected conflict

- virtual const VariablesArray & all_variables ()

  return the complete set of evaluated variables

- virtual const RealMatrix & all_samples ()

  return the complete set of evaluated samples

- virtual const IntResponseMap & all_responses () const

  return the complete set of computed responses

- virtual bool compact_mode () const

  returns Analyzer::compactMode

- void run_iterator (std::ostream &s)

  orchestrate initialize/pre/run/post/finalize phases

- void assign_rep (Iterator *iterator_rep, bool ref_count_incr=true)

  replaces existing letter with a new one

- ProblemDescDB & problem_description_db () const

  return the problem description database (probDescDB)

- const String & method_name () const

  return the method name

- const String & method_id () const

  return the method identifier (methodId)

- void output_level (short out_lev)

  set the method output level (outputLevel)

- short output_level () const

return the method output level (outputLevel)

- void summary_output (bool summary_output_flag)

  Set summary output control; true enables evaluation/results summary.

- int maximum_concurrency () const

  return the maximum concurrency supported by the iterator

- void maximum_concurrency (int max_conc)

  set the maximum concurrency supported by the iterator

- size_t num_final_solutions () const

  return the number of solutions to retain in best variables/response arrays

- void num_final_solutions (size_t num_final)

  set the number of solutions to retain in best variables/response arrays

- void active_set (const ActiveSet &set)

  set the default active set vector (for use with iterators that employ evaluate_parameter_sets())

- const ActiveSet & active_set () const

  return the default active set vector (used by iterators that employ evaluate_parameter_sets())

- void sub_iterator_flag (bool si_flag)

  set subIteratorFlag (and update summaryOutputFlag if needed)

- void active_variable_mappings (const SizetArray &c_index1, const SizetArray &di_index1, const SizetArray &dr_index1, const ShortArray &c_target2, const ShortArray &di_target2, const ShortArray &dr_target2)

  set primaryA{CV,DIV,DRV}MapIndices, secondaryA{CV,DIV,DRV}MapTargets

- bool is_null () const

  function to check iteratorRep (does this envelope contain a letter?)

- Iterator * iterator_rep () const

  returns iteratorRep for access to derived class member functions that are not mapped to the top Iterator level

Protected Member Functions

- Iterator (BaseConstructor, Model &model)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- Iterator (NoDBBaseConstructor, Model &model)

  alternate constructor for base iterator classes constructed on the fly

- Iterator (NoDBBaseConstructor)

  alternate constructor for base iterator classes constructed on the fly

- virtual const VariablesArray & initial_points () const

  gets the multiple initial points for this iterator. This will only be meaningful after a call to initial_points mutator.

Protected Attributes

- Model iteratedModel

  shallow copy of the model passed into the constructor or a thin RecastModel wrapped around it

- ProblemDescDB & probDescDB

  class member reference to the problem description database

- String methodName

  name of the iterator (the user's method spec)

- Real convergenceTol

  iteration convergence tolerance

- int maxIterations

  maximum number of iterations for the iterator

- int maxFunctionEvals

  maximum number of fn evaluations for the iterator

- int maxConcurrency

  maximum coarse-grained concurrency

- size_t numFunctions

  number of response functions

- size_t numContinuousVars

  number of active continuous vars

- size_t numDiscreteIntVars

  number of active discrete integer vars

- size_t numDiscreteRealVars

  number of active discrete real vars

- size_t numFinalSolutions

  number of solutions to retain in best variables/response arrays

- ActiveSet activeSet

  tracks the response data requirements on each function evaluation

- VariablesArray bestVariablesArray

  collection of N best solution variables found during the study

- ResponseArray bestResponseArray

  collection of N best solution responses found during the study

- bool subIteratorFlag

  flag indicating if this Iterator is a sub-iterator (NestedModel::subIterator or DataFitSurrModel::daceIterator)

- SizetArray primaryACVarMapIndices

  "primary" all continuous variable mapping indices flowed down from higher level iteration

- SizetArray primaryADIVarMapIndices

  "primary" all discrete int variable mapping indices flowed down from higher level iteration

- SizetArray primaryADRVarMapIndices

  "primary" all discrete real variable mapping indices flowed down from higher level iteration

- ShortArray secondaryACVarMapTargets

  "secondary" all continuous variable mapping targets flowed down from higher level iteration

- ShortArray secondaryADIVarMapTargets

  "secondary" all discrete int variable mapping targets flowed down from higher level iteration

- ShortArray secondaryADRVarMapTargets

  "secondary" all discrete real variable mapping targets flowed down from higher level iteration

- String gradientType

  type of gradient data: analytic, numerical, mixed, or none

- String methodSource

  source of numerical gradient routine: dakota or vendor

- String intervalType

  type of numerical gradient interval: central or forward

- String hessianType

  type of Hessian data: analytic, numerical, quasi, mixed, or none

- Real fdGradStepSize

  relative finite difference step size for numerical gradients

- Real fdHessByGradStepSize

relative finite difference step size for numerical Hessians estimated using first-order differences of gradients

- Real fdHessByFnStepSize

  relative finite difference step size for numerical Hessians estimated using second-order differences of function values

- short outputLevel

  output verbosity level: {SILENT,QUIET,NORMAL,VERBOSE,DEBUG}_OUTPUT

- bool summaryOutputFlag

  flag for summary output (evaluation stats, final results); default true, but false for on-the-fly (helper) iterators and sub-iterator use cases

- bool asynchFlag

  copy of the model's asynchronous evaluation flag

- int writePrecision

  write precision as specified by the user

Private Member Functions

- Iterator * get_iterator (Model &model)

  Used by the envelope to instantiate the correct letter class.

- Iterator * get_iterator (const String &method_name, Model &model)

  Used by the envelope to instantiate the correct letter class.

- virtual void pre_output ()

  convenience function to write variables to file, following pre-run

- virtual void post_input ()

  read tabular data for post-run mode

Private Attributes

- String methodId

  method identifier string from the input file

- Iterator * iteratorRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing iteratorRep

43.51.1 Detailed Description

Base class for the iterator class hierarchy. The Iterator class is the base class for one of the primary class hierarchies in DAKOTA. The iterator hierarchy contains all of the iterative algorithms which use repeated execution of simulations as function evaluations. For memory efficiency and enhanced polymorphism, the iterator hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Iterator) serves as the envelope and one of the derived classes (selected in Iterator::get_iterator()) serves as the letter.

43.51.2 Constructor & Destructor Documentation

43.51.2.1 Iterator ()

default constructor The default constructor is used in Vector<Iterator> instantiations and for initialization of Iterator objects contained in Strategy derived classes (see derived class header files). iteratorRep is NULL in this case (a populated problem_db is needed to build a meaningful Iterator object). This makes it necessary to check for NULL pointers in the copy constructor, assignment operator, and destructor.

Referenced by SurrBasedGlobalMinimizer::SurrBasedGlobalMinimizer(), and SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.51.2.2 Iterator (Model & *model*)

standard envelope constructor Used in iterator instantiations within strategy constructors. Envelope constructor only needs to extract enough data to properly execute get_iterator(), since letter holds the actual base class data.

References Dakota::abort_handler(), Iterator::get_iterator(), and Iterator::iteratorRep.

43.51.2.3 Iterator (const String & *method_name*, Model & *model*)

alternate envelope constructor for instantiations by name Used in sub-iterator instantiations within iterator constructors. Envelope constructor only needs to extract enough data to properly execute get_iterator(), since letter holds the actual base class data.

References Dakota::abort_handler(), Iterator::get_iterator(), and Iterator::iteratorRep.

43.51.2.4 Iterator (const Iterator & *iterator*)

copy constructor Copy constructor manages sharing of iteratorRep and incrementing of referenceCount.

References Iterator::iteratorRep, and Iterator::referenceCount.

43.51.2.5 ~Iterator () `[virtual]`

destructor Destructor decrements referenceCount and only deletes iteratorRep when referenceCount reaches zero.

References Iterator::iteratorRep, and Iterator::referenceCount.

43.51.2.6 Iterator (BaseConstructor, Model & *model*) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor builds the base class data for all inherited iterators. get_iterator() instantiates a derived class and the derived class selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_iterator() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~Iterator).

References Dakota::abort_handler(), Iterator::fdGradStepSize, Iterator::fdHessByFnStepSize, Iterator::fdHessByGradStepSize, ProblemDescDB::get_dil(), ProblemDescDB::get_rdv(), Iterator::gradientType, Iterator::hessianType, Iterator::intervalType, Iterator::methodName, Iterator::methodSource, Iterator::numContinuousVars, Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, Iterator::numFunctions, and Iterator::probDescDB.

43.51.2.7 Iterator (NoDBBaseConstructor, Model & *model*) [protected]

alternate constructor for base iterator classes constructed on the fly This alternate constructor builds base class data for inherited iterators. It is used for on-the-fly instantiations for which DB queries cannot be used. Therefore it only sets attributes taken from the incoming model. Since there are no iterator-specific redefinitions of maxIterations or numFinalSolutions in NoDBBaseConstructor mode, go ahead and assign default value for all iterators.

43.51.2.8 Iterator (NoDBBaseConstructor) [protected]

alternate constructor for base iterator classes constructed on the fly This alternate constructor builds base class data for inherited iterators. It is used for on-the-fly instantiations for which DB queries cannot be used. It has no incoming model, so only sets up a minimal set of defaults. However, its use is preferable to the default constructor, which should remain as minimal as possible. Since there are no iterator-specific redefinitions of maxIterations or numFinalSolutions in NoDBBaseConstructor mode, go ahead and assign default value for all iterators.

43.51.3 Member Function Documentation

43.51.3.1 Iterator operator= (const Iterator & *iterator*)

assignment operator Assignment operator decrements referenceCount for old iteratorRep, assigns new iteratorRep, and increments referenceCount for new iteratorRep.

References Iterator::iteratorRep, and Iterator::referenceCount.

43.51.3.2 void initialize_run () [virtual]

utility function to perform common operations prior to pre_run(); typically memory initialization; setting of instance pointers Perform initialization phases of run sequence, like allocating memory and setting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's initialize_run(), typically _before_ performing its own implementation steps.

Reimplemented in CONMINOptimizer, LeastSq, Minimizer, NonD, Optimizer, DOTOptimizer, NLPQLPOptimizer, SNLLLeastSq, and SNLLOptimizer.

References Model::asynch_flag(), Iterator::asynchFlag, Iterator::initialize_run(), Model::is_null(), Iterator::iteratedModel, Iterator::iteratorRep, Iterator::maxConcurrency, Model::set_communicators(), Model::set_evaluation_reference(), and Iterator::summaryOutputFlag.

Referenced by Iterator::initialize_run(), and Iterator::run_iterator().

43.51.3.3 void pre_run () [virtual]

pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori pre-run phase, which a derived iterator may optionally reimplement; when not present, pre-run is likely integrated into the derived run function. This is a virtual function; when re-implementing, a derived class must call its nearest parent's pre_run(), if implemented, typically _before_ performing its own implementation steps.

Reimplemented in DDACEDesignCompExp, FSUDesignCompExp, NonDLHSSampling, ParamStudy, and PSUADEDesignCompExp.

References Iterator::iteratorRep, and Iterator::pre_run().

Referenced by Iterator::pre_run(), and Iterator::run_iterator().

43.51.3.4 void run () [virtual]

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented in LeastSq, NonD, Optimizer, PStudyDACE, Verification, and SurrBasedMinimizer.

References Dakota::abort_handler(), Iterator::iteratorRep, and Iterator::run().

Referenced by Iterator::run(), and Iterator::run_iterator().

43.51.3.5 void post_run (std::ostream & *s*) [virtual]

post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way Post-run phase, which a derived iterator may optionally reimplement; when not present, post-run is likely integrated into run. This is a virtual function; when re-implementing, a derived class must call its nearest parent's post_run(), typically _after_ performing its own implementation steps.

Reimplemented in COLINOptimizer, LeastSq, Optimizer, DDACEDesignCompExp, FSUDesignCompExp, NonDLHSSampling, ParamStudy, PSUADEDesignCompExp, SNLLLeastSq, and SNLLOptimizer.

References Model::is_null(), Iterator::iteratedModel, Iterator::iteratorRep, Iterator::post_run(), Model::print_evaluation_summary(), Iterator::print_results(), and Iterator::summaryOutputFlag.

Referenced by Iterator::post_run(), and Iterator::run_iterator().

43.51.3.6 void finalize_run () [virtual]

utility function to perform common operations following post_run(); deallocation and resetting of instance pointers Optional: perform finalization phases of run sequence, like deallocating memory and resetting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's finalize_run(), typically _after_ performing its own implementation steps.

Reimplemented in LeastSq, Minimizer, NonD, Optimizer, SNLLLeastSq, and SNLLOptimizer.

References Iterator::finalize_run(), and Iterator::iteratorRep.

Referenced by Iterator::finalize_run(), and Iterator::run_iterator().

43.51.3.7 void initialize_graphics (bool *graph_2d*, bool *tabular_data*, const String & *tabular_file*) [virtual]

initialize the 2D graphics window and the tabular graphics data This is a convenience function for encapsulating graphics initialization operations. It does not require a strategyRep forward since it is only used by letter objects.

Reimplemented in NonDReliability, and SurrBasedMinimizer.

References Model::auto_graphics(), Graphics::create_plots_2d(), Graphics::create_tabular_datastream(), Model::current_response(), Model::current_variables(), Dakota::dakota_graphics, Iterator::initialize_graphics(), Iterator::iteratedModel, and Iterator::iteratorRep.

Referenced by Iterator::initialize_graphics(), SequentialHybridStrategy::run_sequential(), SingleMethodStrategy::run_strategy(), EmbeddedHybridStrategy::run_strategy(), ConcurrentStrategy::run_strategy(), and CollaborativeHybridStrategy::run_strategy().

43.51.3.8 void print_results (std::ostream & *s*) [virtual]

print the final iterator results This virtual function provides additional iterator-specific final results outputs beyond the function evaluation summary printed in finalize_run().

Reimplemented in Analyzer, LeastSq, Optimizer, PStudyDACE, Verification, NonDExpansion, NonDGlobalReliability, NonDGPMSABayesCalibration, NonDIncremLHSSampling, NonDInterval, NonDLHSSampling, NonDLocalReliability, RichExtrapVerification, and SurrBasedMinimizer.

References Iterator::iteratorRep, and Iterator::print_results().

Referenced by Iterator::post_run(), and Iterator::print_results().

43.51.3.9 int num_samples () const [virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented in DDACEDesignCompExp, FSUDesignCompExp, NonDCubature, NonDQuadrature, NonDSampling, NonDSparseGrid, and PSUADEDesignCompExp.

References Model::derivative_concurrency(), Iterator::iteratedModel, Iterator::iteratorRep, Iterator::maxConcurrency, and Iterator::num_samples().

Referenced by DataFitSurrModel::build_global(), NonDGlobalReliability::get_best_sample(), Iterator::num_samples(), Analyzer::samples_to_variables_array(), and Analyzer::variables_array_to_samples().

43.51.3.10 void run_iterator (std::ostream & *s*)

orchestrate initialize/pre/run/post/finalize phases Iterator supports a construct/initialize-run/pre-run/run/post-run/finalize-run/destruct progression. This member (non-virtual) function sequences these run phases; it accepts an ostream, but controls verbosity with outputLevel

References ParallelLibrary::command_line_post_run(), ParallelLibrary::command_line_pre_run(), ParallelLibrary::command_line_run(), Iterator::finalize_run(), Iterator::initialize_run(), Iterator::iteratedModel, Iterator::iteratorRep, Iterator::methodName, Iterator::outputLevel, Model::parallel_library(), Iterator::post_input(), Iterator::post_run(), Iterator::pre_output(), Iterator::pre_run(), Iterator::run(), Iterator::run_iterator(), and Iterator::summaryOutputFlag.

Referenced by DataFitSurrModel::build_global(), NonDExpansion::compute_statistics(), NestedModel::derived_compute_response(), NonDGlobalReliability::importance_sampling(), SurrBasedLocalMinimizer::minimize_surrogates(), SurrBasedGlobalMinimizer::minimize_surrogates(), EffGlobalMinimizer::minimize_surrogates_on_model(), NonDLocalReliability::mpp_search(), NonDGlobalReliability::optimize_gaussian_process(), GaussProcApproximation::optimize_theta_global(), GaussProcApproximation::optimize_theta_multipoint(), NonDLocalReliability::probability(), NonDLocalInterval::quantify_uncertainty(), NonDLHSInterval::quantify_uncertainty(), NonDGPMSABayesCalibration::quantify_uncertainty(), NonDGlobalInterval::quantify_uncertainty(), NonDBayesCalibration::quantify_uncertainty(), SurrBasedLocalMinimizer::relax_constraints(), Strategy::run_iterator(), and Iterator::run_iterator().

43.51.3.11 void assign_rep (Iterator * *iterator_rep*, bool *ref_count_incr* = `true`)

replaces existing letter with a new one Similar to the assignment operator, the assign_rep() function decrements referenceCount for the old iteratorRep and assigns the new iteratorRep. It is different in that it is used for publishing derived class letters to existing envelopes, as opposed to sharing representations among multiple envelopes (in particular, assign_rep is passed a letter object and operator= is passed an envelope object). Letter assignment supports two models as governed by ref_count_incr:

- ref_count_incr = true (default): the incoming letter belongs to another envelope. In this case, increment the reference count in the normal manner so that deallocation of the letter is handled properly.
- ref_count_incr = false: the incoming letter is instantiated on the fly and has no envelope. This case is modeled after get_iterator(): a letter is dynamically allocated using new and passed into assign_rep, the letter's reference count is not incremented, and the letter is not remotely deleted (its memory management is passed over to the envelope).

References Dakota::abort_handler(), Iterator::iteratorRep, and Iterator::referenceCount.

Referenced by NonDExpansion::construct_cubature(), NonDExpansion::construct_expansion_sampler(), NonD::construct_lhs(), NonDExpansion::construct_quadrature(), NonDExpansion::construct_sparse_grid(), EffGlobalMinimizer::EffGlobalMinimizer(), NonDLocalReliability::method_recourse(), NonDLocalInterval::method_recourse(), NonDBayesCalibration::NonDBayesCalibration(), NonDGlobalInterval::NonDGlobalInterval(), NonDGlobalReliability::NonDGlobalReliability(),

NonDGPMSABayesCalibration::NonDGPMSABayesCalibration(), NonDLHSInterval::NonDLHSInterval(), NonDLocalInterval::NonDLocalInterval(), NonDLocalReliability::NonDLocalReliability(), GaussProcApproximation::optimize_theta_global(), GaussProcApproximation::optimize_theta_multipoint(), and SurrBasedLocalMinimizer::relax_constraints().

43.51.3.12 Iterator ∗ get_iterator (Model & *model*) [private]

Used by the envelope to instantiate the correct letter class. Used only by the envelope constructor to initialize iteratorRep to the appropriate derived type, as given by the methodName attribute.

References String::begins(), String::ends(), ProblemDescDB::get_string(), Iterator::method_name(), Iterator::methodName, and Iterator::probDescDB.

Referenced by Iterator::Iterator().

43.51.3.13 Iterator ∗ get_iterator (const String & *method_name*, Model & *model*) [private]

Used by the envelope to instantiate the correct letter class. Used only by the envelope constructor to initialize iteratorRep to the appropriate derived type, as given by the passed method_name.

References String::begins().

43.51.4 Member Data Documentation

43.51.4.1 Real fdGradStepSize [protected]

relative finite difference step size for numerical gradients A scalar value (instead of the vector fd_gradient_step_size spec) is used within the iterator hierarchy since this attribute is only used to publish a step size to vendor numerical gradient algorithms.

Referenced by DOTOptimizer::initialize(), CONMINOptimizer::initialize(), Iterator::Iterator(), NLSSOLLeastSq::NLSSOLLeastSq(), NPSOLOptimizer::NPSOLOptimizer(), SNLLLeastSq::SNLLLeastSq(), and SNLLOptimizer::SNLLOptimizer().

43.51.4.2 Real fdHessByGradStepSize [protected]

relative finite difference step size for numerical Hessians estimated using first-order differences of gradients A scalar value (instead of the vector fd_hessian_step_size spec) is used within the iterator hierarchy since this attribute is only used to publish a step size to vendor numerical Hessian algorithms.

Referenced by Iterator::Iterator().

43.51.4.3 Real fdHessByFnStepSize [protected]

relative finite difference step size for numerical Hessians estimated using second-order differences of function values A scalar value (instead of the vector fd_hessian_step_size spec) is used within the iterator hierarchy since this attribute is only used to publish a step size to vendor numerical Hessian algorithms.

Referenced by Iterator::Iterator().

The documentation for this class was generated from the following files:

- DakotaIterator.H
- DakotaIterator.C

43.52 JEGAOptimizer Class Reference

A version of Dakota::Optimizer for instantiation of John Eddy's Genetic Algorithms (JEGA). Inheritance diagram for JEGAOptimizer::

Classes

- class Driver

  A subclass of the JEGA front end driver that exposes the individual protected methods to execute the algorithm.

- class Evaluator

  An evaluator specialization that knows how to interact with Dakota.

- class EvaluatorCreator

  A specialization of the JEGA::FrontEnd::EvaluatorCreator that creates a new instance of a Evaluator.

Public Member Functions

- virtual void find_optimum ()

  Performs the iterations to determine the optimal set of solutions.

- virtual bool accepts_multiple_points () const

  Overridden to return true since JEGA algorithms can accept multiple initial points.

- virtual bool returns_multiple_points () const

  Overridden to return true since JEGA algorithms can return multiple final points.

- virtual void initial_points (const VariablesArray &pts)

  Overridden to assign the _initPts member variable to the passed in collection of Dakota::Variables.

- virtual const VariablesArray & initial_points () const

  Overridden to return the collection of initial points for the JEGA algorithm created and run by this JEGAOptimizer.

- JEGAOptimizer (Model &model)

  Constructs a JEGAOptimizer class object.

- ~JEGAOptimizer ()

  Destructs a JEGAOptimizer.

Protected Member Functions

- void LoadDakotaResponses (const JEGA::Utilities::Design &from, Variables &vars, Response &resp) const

  Loads the JEGA-style Design class into equivolent Dakota-style Variables and Response objects.

- void ReCreateTheParameterDatabase ()

  Destroys the current parameter database and creates a new empty one.

- void LoadTheParameterDatabase ()

  Reads information out of the known Dakota::ProblemDescDB and puts it into the current parameter database.

- void LoadAlgorithmConfig (JEGA::FrontEnd::AlgorithmConfig &aConfig)

  Completely initializes the supplied algorithm configuration.

- void LoadProblemConfig (JEGA::FrontEnd::ProblemConfig &pConfig)

  Completely initializes the supplied problem configuration.

- void LoadTheDesignVariables (JEGA::FrontEnd::ProblemConfig &pConfig)

  Adds DesignVariableInfo objects into the problem configuration object.

- void LoadTheObjectiveFunctions (JEGA::FrontEnd::ProblemConfig &pConfig)

  Adds ObjectiveFunctionInfo objects into the problem configuration object.

- void LoadTheConstraints (JEGA::FrontEnd::ProblemConfig &pConfig)

  Adds ConstraintInfo objects into the problem configuration object.

- void GetBestSolutions (const JEGA::Utilities::DesignOFSortSet &from, std::multimap< RealRealPair, JEGA::Utilities::Design * > &designSortMap)

  Returns up to _numBest designs sorted by DAKOTA's fitness (L2 constraint violation, then utopia or objective), taking into account the algorithm type. The front of the returned map can be viewed as a single "best".

- void GetBestMOSolutions (const JEGA::Utilities::DesignOFSortSet &from, std::multimap< RealRealPair, JEGA::Utilities::Design * > &designSortMap)

  Retreive the best Designs from a set of solutions assuming that they are generated by a multi objective algorithm.

- void GetBestSOSolutions (const JEGA::Utilities::DesignOFSortSet &from, std::multimap< RealRealPair, JEGA::Utilities::Design * > &designSortMap)

  Retreive the best Designs from a set of solutions assuming that they are generated by a single objective algorithm.

- JEGA::DoubleMatrix ToDoubleMatrix (const VariablesArray &variables) const

  Converts the items in a VariablesArray into a DoubleMatrix whereby the items in the matrix are the design variables.

- void resize_variables_results_array (std::size_t newsize)

  Safely resizes the best variables array taking into account the requirements put forth by the envelope-letter design pattern.

- void resize_response_results_array (std::size_t newsize)

  Safely resizes the best response array taking into account the requirements put forth by the envelope-letter design pattern.

Private Attributes

- EvaluatorCreator * _theEvalCreator

  A pointer to an EvaluatorCreator used to create the evaluator used by JEGA in Dakota (a JEGAEvaluator).

- JEGA::Utilities::ParameterDatabase * _theParamDB

  A pointer to the ParameterDatabase from which all parameters are retrieved by the created algorithms.

- VariablesArray _initPts

  An array of initial points to use as an initial population.

Static Private Attributes

- static const std::string SOGA_METHOD_TXT

  The text that indicates the SOGA method.

- static const std::string MOGA_METHOD_TXT

  The text that indicates the MOGA method.

43.52.1 Detailed Description

A version of Dakota::Optimizer for instantiation of John Eddy's Genetic Algorithms (JEGA). This class encapsulates the necessary functionality for creating and properly initializing the JEGA algorithms (MOGA and SOGA).

43.52.2 Constructor & Destructor Documentation

43.52.2.1 JEGAOptimizer (Model & *model*)

Constructs a JEGAOptimizer class object. This method does some of the initialization work for the algorithm. In particular, it initialized the JEGA core.

Parameters:

model The Dakota::Model that will be used by this optimizer for problem information, etc.

References JEGAOptimizer::_theEvalCreator, ProblemDescDB::get_int(), ProblemDescDB::get_short(), Model::init_communicators(), Iterator::iteratedModel, JEGAOptimizer::LoadTheParameterDatabase(), Optimizer::localObjectiveRecast, Iterator::maxConcurrency, Iterator::methodName, JEGAOptimizer::MOGA_METHOD_TXT, Iterator::numFinalSolutions, Iterator::probDescDB, and Minimizer::scaleFlag.

43.52.3 Member Function Documentation

43.52.3.1 void LoadDakotaResponses (const JEGA::Utilities::Design & *from*, Dakota::Variables & *vars*, Dakota::Response & *resp*) const [protected]

Loads the JEGA-style Design class into equivolent Dakota-style Variables and Response objects. This version is meant for the case where a Variables and a Response object exist and just need to be loaded.

Parameters:

from The JEGA Design class object from which to extract the variable and response information for Dakota.

vars The Dakota::Variables object into which to load the design variable values of *from*.

resp The Dakota::Response object into which to load the objective function and constraint values of *from*.

References Variables::continuous_variables(), Variables::discrete_int_variables(), Variables::discrete_real_variables(), Response::function_values(), Iterator::numContinuousVars, Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, Iterator::numFunctions, Minimizer::numNonlinearConstraints, and Optimizer::numObjectiveFns.

Referenced by JEGAOptimizer::find_optimum().

43.52.3.2 void LoadTheParameterDatabase () [protected]

Reads information out of the known Dakota::ProblemDescDB and puts it into the current parameter database. This should be called from the JEGAOptimizer constructor since it is the only time when the problem description database is certain to be configured to supply data for this optimizer.

References JEGAOptimizer::_theParamDB, ProblemDescDB::get_bool(), ProblemDescDB::get_int(), ProblemDescDB::get_rdv(), ProblemDescDB::get_real(), ProblemDescDB::get_short(), ProblemDescDB::get_sizet(), ProblemDescDB::get_string(), Iterator::probDescDB, and JEGAOptimizer::ReCreateTheParameterDatabase().

Referenced by JEGAOptimizer::JEGAOptimizer().

43.52.3.3 void LoadAlgorithmConfig (JEGA::FrontEnd::AlgorithmConfig & *aConfig*) [protected]

Completely initializes the supplied algorithm configuration. This loads the supplied configuration object with appropriate data retrieved from the parameter database.

Parameters:

aConfig The algorithm configuration object to load.

References Iterator::method_id(), Iterator::methodName, JEGAOptimizer::MOGA_METHOD_TXT, and JEGAOptimizer::SOGA_METHOD_TXT.

Referenced by JEGAOptimizer::find_optimum().

43.52.3.4 void LoadProblemConfig (JEGA::FrontEnd::ProblemConfig & *pConfig*) `[protected]`

Completely initializes the supplied problem configuration. This loads the fresh configuration object using the LoadTheDesignVariables, LoadTheObjectiveFunctions, and LoadTheConstraints methods.

Parameters:

pConfig The problem configuration object to load.

References JEGAOptimizer::LoadTheConstraints(), JEGAOptimizer::LoadTheDesignVariables(), and JEGAOptimizer::LoadTheObjectiveFunctions().

Referenced by JEGAOptimizer::find_optimum().

43.52.3.5 void LoadTheDesignVariables (JEGA::FrontEnd::ProblemConfig & *pConfig*) `[protected]`

Adds DesignVariableInfo objects into the problem configuration object. This retrieves design variable information from the ParameterDatabase and creates DesignVariableInfo's from it.

Parameters:

pConfig The problem configuration object to load.

References Iterator::iteratedModel, Iterator::numContinuousVars, and Iterator::numDiscreteIntVars.

Referenced by JEGAOptimizer::LoadProblemConfig().

43.52.3.6 void LoadTheObjectiveFunctions (JEGA::FrontEnd::ProblemConfig & *pConfig*) `[protected]`

Adds ObjectiveFunctionInfo objects into the problem configuration object. This retrieves objective function information from the ParameterDatabase and creates ObjectiveFunctionInfo's from it.

Parameters:

pConfig The problem configuration object to load.

References Dakota::asstring(), and Optimizer::numObjectiveFns.

Referenced by JEGAOptimizer::LoadProblemConfig().

43.52.3.7 void LoadTheConstraints (JEGA::FrontEnd::ProblemConfig & *pConfig*) [protected]

Adds ConstraintInfo objects into the problem configuration object. This retrieves constraint function information from the ParameterDatabase and creates ConstraintInfo's from it.

Parameters:

pConfig The problem configuration object to load.

References Dakota::asstring(), Dakota::copy_row_vector(), Iterator::iteratedModel, Minimizer::numLinearEqConstraints, Minimizer::numLinearIneqConstraints, Minimizer::numNonlinearEqConstraints, and Minimizer::numNonlinearIneqConstraints.

Referenced by JEGAOptimizer::LoadProblemConfig().

43.52.3.8 void GetBestSolutions (const JEGA::Utilities::DesignOFSortSet & *from*, std::multimap< RealRealPair, JEGA::Utilities::Design * > & *designSortMap*) [protected]

Returns up to _numBest designs sorted by DAKOTA's fitness (L2 constraint violation, then utopia or objective), taking into account the algorithm type. The front of the returned map can be viewed as a single "best".

Parameters:

from The full set of designs returned by the solver.

designSortMap Map of best solutions with key pair<constraintViolation, fitness>

eventually this functionality must be moved into a separate post-processing application for MO datasets.

References JEGAOptimizer::GetBestMOSolutions(), JEGAOptimizer::GetBestSOSolutions(), Iterator::methodName, JEGAOptimizer::MOGA_METHOD_TXT, and JEGAOptimizer::SOGA_METHOD_TXT.

Referenced by JEGAOptimizer::find_optimum().

43.52.3.9 void GetBestMOSolutions (const JEGA::Utilities::DesignOFSortSet & *from*, std::multimap< RealRealPair, JEGA::Utilities::Design * > & *designSortMap*) [protected]

Retreive the best Designs from a set of solutions assuming that they are generated by a multi objective algorithm. eventually this functionality must be moved into a separate post-processing application for MO datasets.

References Iterator::numFinalSolutions.

Referenced by JEGAOptimizer::GetBestSolutions().

43.52.3.10 void GetBestSOSolutions (const JEGA::Utilities::DesignOFSortSet & *from*, std::multimap< RealRealPair, JEGA::Utilities::Design * > & *designSortMap*) [protected]

Retreive the best Designs from a set of solutions assuming that they are generated by a single objective algorithm. eventually this functionality must be moved into a separate post-processing application for MO datasets.

References JEGAOptimizer::_theParamDB, and Iterator::numFinalSolutions.

Referenced by JEGAOptimizer::GetBestSolutions().

43.52.3.11 JEGA::DoubleMatrix ToDoubleMatrix (const VariablesArray & *variables*) const [protected]

Converts the items in a VariablesArray into a DoubleMatrix whereby the items in the matrix are the design variables. The matrix will not contain responses but when being used by Dakota, this doesn't matter. JEGA will attempt to re-evaluate these points but Dakota will recognize that they do not require re-evaluation and thus it will be a cheap operation.

Parameters:

variables The array of DakotaVariables objects to use as the contents of the returned matrix.

Returns:

The matrix created using the supplied VariablesArray.

Referenced by JEGAOptimizer::find_optimum().

43.52.3.12 void resize_variables_results_array (std::size_t *newsize*) [protected]

Safely resizes the best variables array taking into account the requirements put forth by the envelope-letter design pattern. Do not directly call resize on the bestVariablesArray object unless you intend to share the internal content (letter) with other objects after assignment.

Parameters:

newsize The new size for the variables array.

References Iterator::bestVariablesArray.

Referenced by JEGAOptimizer::find_optimum().

43.52.3.13 void resize_response_results_array (std::size_t *newsize*) [protected]

Safely resizes the best response array taking into account the requirements put forth by the envelope-letter design pattern. Do not directly call resize on the bestResponseArray object unless you intend to share the internal content (letter) with other objects after assignment.

Parameters:

newsize The new size for the responses array.

References Iterator::bestResponseArray.

Referenced by JEGAOptimizer::find_optimum().

43.52.3.14 void find_optimum () [virtual]

Performs the iterations to determine the optimal set of solutions. Override of pure virtual method in Optimizer base class.

The extraction of parameter values actually occurs in this method when the JEGA::FrontEnd::Driver::ExecuteAlgorithm is called. Also the loading of the problem and algorithm configurations occurs in this method. That way, if it is called more than once and the algorithm or problem has changed, it will be accounted for.

Implements Optimizer.

References JEGAOptimizer::_initPts, JEGAOptimizer::_theEvalCreator, JEGAOptimizer::_theParamDB, Driver::DestroyAlgorithm(), Driver::ExtractAllData(), JEGAOptimizer::GetBestSolutions(), JEGAOptimizer::initial_points(), JEGAOptimizer::LoadAlgorithmConfig(), JEGAOptimizer::LoadDakotaResponses(), JEGAOptimizer::LoadProblemConfig(), Driver::PerformIterations(), JEGAOptimizer::resize_response_results_array(), JEGAOptimizer::resize_variables_results_array(), and JEGAOptimizer::ToDoubleMatrix().

43.52.3.15 bool accepts_multiple_points () const [virtual]

Overridden to return true since JEGA algorithms can accept multiple initial points.

Returns:

true, always.

Reimplemented from Iterator.

43.52.3.16 bool returns_multiple_points () const [virtual]

Overridden to return true since JEGA algorithms can return multiple final points.

Returns:

true, always.

Reimplemented from Iterator.

43.52.3.17 void initial_points (const VariablesArray & *pts*) [virtual]

Overridden to assign the _initPts member variable to the passed in collection of Dakota::Variables.

Parameters:

pts The array of initial points for the JEGA algorithm created and run by this JEGAOptimizer.

Reimplemented from Iterator.

References JEGAOptimizer::_initPts.

43.52.3.18 const VariablesArray & initial_points () const [virtual]

Overridden to return the collection of initial points for the JEGA algorithm created and run by this JEGAOptimizer.

Returns:

The collection of initial points for the JEGA algorithm created and run by this JEGAOptimizer.

Reimplemented from Iterator.

References JEGAOptimizer::_initPts.

Referenced by JEGAOptimizer::find_optimum().

43.52.4 Member Data Documentation

43.52.4.1 VariablesArray _initPts [private]

An array of initial points to use as an initial population. This member is here to help support the use of JEGA algorithms in Dakota strategies. If this array is populated, then whatever initializer is specified will be ignored and the DoubleMatrix initializer will be used instead on a matrix created from the data in this array.

Referenced by JEGAOptimizer::find_optimum(), and JEGAOptimizer::initial_points().

The documentation for this class was generated from the following files:

- JEGAOptimizer.H
- JEGAOptimizer.C

43.53 LeastSq Class Reference

Base class for the nonlinear least squares branch of the iterator hierarchy. Inheritance diagram for LeastSq::

Protected Member Functions

- LeastSq ()

  default constructor

- LeastSq (Model &model)

  standard constructor

- LeastSq (NoDBBaseConstructor, Model &model)

  alternate constructor

- ~LeastSq ()

  destructor

- void initialize_run ()
- void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- void post_run (std::ostream &s)
- void finalize_run ()

  utility function to perform common operations following post_run(); deallocation and resetting of instance pointers

- void print_results (std::ostream &s)
- virtual void minimize_residuals ()=0

  Used within the least squares branch for minimizing the sum of squares residuals. Redefines the run virtual function for the least squares branch.

- void get_confidence_intervals ()

  Calculate confidence intervals on estimated parameters.

Static Protected Member Functions

- static void primary_resp_recast (const Variables &native_vars, const Variables &scaled_vars, const Response &native_response, Response &scaled_response)

  primary response conversion map for RecastModel used in scaling: transform least squares terms (fns, grads, Hessians) from native (user) to iterator space

Protected Attributes

- int numLeastSqTerms

  number of least squares terms

- LeastSq * prevLSqInstance

  pointer containing previous value of leastSqInstance

- bool weightFlag

  flag indicating whether weighted least squares is active

- String obsDataFilename

  filename from which to read observed data

- bool obsDataFlag

  flag indicating whether user-supplied data is active

- RealVector obsData

  storage for user-supplied data for computing residuals

- RealVector confBoundsLower

  lower bounds for confidence intervals on calibration parameters

- RealVector confBoundsUpper

  upper bounds for confidence intervals on calibration parameters

Static Protected Attributes

- static LeastSq * leastSqInstance

  pointer to LeastSq instance used in static member functions

43.53.1 Detailed Description

Base class for the nonlinear least squares branch of the iterator hierarchy. The LeastSq class provides common data and functionality for least squares solvers (including NL2OL, NLSSOLLeastSq, and SNLLLeastSq.

43.53.2 Constructor & Destructor Documentation

43.53.2.1 LeastSq (Model & *model*) [protected]

standard constructor This constructor extracts the inherited data for the least squares branch and performs sanity checking on gradient and constraint settings.

References Dakota::abort_handler(), Model::assign_rep(), Iterator::bestVariablesArray, Variables::copy(), Model::current_variables(), Minimizer::cvScaleTypes, ProblemDescDB::get_bool(), ProblemDescDB::get_sizet(), Model::init_communicators(), RecastModel::initialize(), Minimizer::initialize_scaling(), Iterator::iteratedModel, Iterator::maxConcurrency, Model::model_rep(), Iterator::numContinuousVars, Iterator::numFunctions, Minimizer::numIterPrimaryFns, LeastSq::numLeastSqTerms, Minimizer::numNonlinearConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, LeastSq::obsData, LeastSq::obsDataFilename, LeastSq::obsDataFlag, Minimizer::optimizationFlag, Iterator::outputLevel, LeastSq::primary_resp_recast(), Model::primary_response_fn_weights(), Minimizer::primaryRespScaleFlag, Iterator::probDescDB, Dakota::read_data_tabular(), Minimizer::responseScaleTypes, Minimizer::scaleFlag, Minimizer::secondary_resp_recast(), Minimizer::secondaryRespScaleFlag, Minimizer::variables_recast(), Minimizer::varsScaleFlag, and LeastSq::weightFlag.

43.53.3 Member Function Documentation

43.53.3.1 void initialize_run () [protected, virtual]

This function should be invoked (or reimplemented) by any derived implementations of initialize_run() (which would otherwise hide it).

Reimplemented from Minimizer.

Reimplemented in SNLLLeastSq.

References Iterator::iteratedModel, LeastSq::leastSqInstance, LeastSq::obsDataFlag, LeastSq::prevLSqInstance, Minimizer::scaleFlag, and Model::update_from_subordinate_model().

43.53.3.2 void run () [inline, protected, virtual]

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Iterator.

References LeastSq::minimize_residuals().

43.53.3.3 void post_run (std::ostream & *s*) [protected, virtual]

Implements portions of post_run specific to LeastSq for scaling back to native variables and functions. This function should be invoked (or reimplemented) by any derived implementations of post_run() (which would otherwise hide it).

Reimplemented from Iterator.

Reimplemented in SNLLLeastSq.

References Dakota::abort_handler(), Response::active_set_request_vector(), Iterator::bestResponseArray, Iterator::bestVariablesArray, Variables::continuous_variables(), Response::copy(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Response::function_value(), Response::function_values(), Iterator::iteratedModel, Minimizer::modify_s2n(), Minimizer::need_resp_trans_byvars(), LeastSq::numLeastSqTerms, Minimizer::numNonlinearConstraints, Model::primary_response_fn_weights(), Minimizer::primaryRespScaleFlag, Minimizer::response_modify_s2n(), Minimizer::secondaryRespScaleFlag, Model::subordinate_model(), Response::update_partial(), Minimizer::varsScaleFlag, and LeastSq::weightFlag.

43.53.3.4 void finalize_run () [inline, protected, virtual]

utility function to perform common operations following post_run(); deallocation and resetting of instance pointers Optional: perform finalization phases of run sequence, like deallocating memory and resetting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's finalize_run(), typically _after_ performing its own implementation steps.

Reimplemented from Minimizer.

Reimplemented in SNLLLeastSq.

References LeastSq::leastSqInstance, and LeastSq::prevLSqInstance.

43.53.3.5 void print_results (std::ostream & *s*) [protected, virtual]

Redefines default iterator results printing to include nonlinear least squares results (residual terms and constraints).

Reimplemented from Iterator.

References Iterator::activeSet, Iterator::bestResponseArray, Iterator::bestVariablesArray, LeastSq::confBoundsLower, LeastSq::confBoundsUpper, Model::continuous_variable_labels(), Dakota::data_pairs, Model::interface_id(), Iterator::iteratedModel, Dakota::lookup_by_val(), Iterator::numContinuousVars, Iterator::numFunctions, LeastSq::numLeastSqTerms, ActiveSet::request_values(), Dakota::write_data_partial(), and Dakota::write_precision.

43.53.3.6 void primary_resp_recast (const Variables & *native_vars*, const Variables & *scaled_vars*, const Response & *native_response*, Response & *iterator_response*) [static, protected]

primary response conversion map for RecastModel used in scaling: transform least squares terms (fns, grads, Hessians) from native (user) to iterator space Least squares function map from user/native space to iterator/scaled space using a RecastModel. If no scaling also copies constraints.

References Dakota::_NPOS, Response::active_set_derivative_vector(), Response::active_set_request_vector(), Variables::acv(), Variables::all_continuous_variable_ids(), Variables::continuous_variable_ids(), Response::copy(), Variables::cv(), Dakota::find_index(), Response::function_gradient_view(), Response::function_hessian_view(), Response::function_labels(), Response::function_value(), Response::function_values(), Response::function_values_view(), Variables::icv(), Variables::inactive_continuous_variable_ids(), Iterator::iteratedModel, LeastSq::leastSqInstance, Minimizer::need_resp_trans_byvars(), LeastSq::numLeastSqTerms, LeastSq::obsData, LeastSq::obsDataFlag, Iterator::outputLevel, Model::primary_response_fn_weights(), Minimizer::primaryRespScaleFlag, Minimizer::response_modify_n2s(), Model::subordinate_model(), Response::update(), Response::update_partial(), LeastSq::weightFlag, and

Dakota::write_data().

Referenced by LeastSq::LeastSq().

43.53.3.7 void get_confidence_intervals () [protected]

Calculate confidence intervals on estimated parameters. Calculate individual confidence intervals for each parameter. These bounds are based on a linear approximation of the nonlinear model.

References Iterator::activeSet, Iterator::bestResponseArray, Iterator::bestVariablesArray, Model::compute_response(), LeastSq::confBoundsLower, LeastSq::confBoundsUpper, Model::continuous_variables(), Model::current_response(), Response::function_gradients(), Iterator::iteratedModel, Iterator::numContinuousVars, LeastSq::numLeastSqTerms, ActiveSet::request_values(), Minimizer::scaleFlag, and Minimizer::vendorNumericalGradFlag.

Referenced by NLSSOLLeastSq::minimize_residuals(), NL2SOLLeastSq::minimize_residuals(), and SNLLLeastSq::post_run().

The documentation for this class was generated from the following files:

- DakotaLeastSq.H
- DakotaLeastSq.C

43.54 MergedConstraints Class Reference

Derived class within the Constraints hierarchy which employs the merged data view. Inheritance diagram for MergedConstraints::

Public Member Functions

- MergedConstraints (const SharedVariablesData &svd)

  lightweight constructor

- MergedConstraints (const ProblemDescDB &problem_db, const SharedVariablesData &svd)

  standard constructor

- ~MergedConstraints ()

  destructor

- void write (std::ostream &s) const

  write a variable constraints object to an std::ostream

- void read (std::istream &s)

  read a variable constraints object from an std::istream

Protected Member Functions

- void reshape (const SizetArray &vc_totals)

  reshape the lower/upper bound arrays within the Constraints hierarchy

- void build_active_views ()

  construct active views of all variables bounds arrays

- void build_inactive_views ()

  construct inactive views of all variables bounds arrays

43.54.1 Detailed Description

Derived class within the Constraints hierarchy which employs the merged data view. Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MergedConstraints derived class combines continuous and discrete domain types but separates design, uncertain, and state variable types. The result is merged design bounds arrays (mergedDesignLowerBnds, mergedDesignUpperBnds), uncertain distribution bounds arrays (uncertainLowerBnds, uncertainUpperBnds), and merged state bounds arrays (mergedStateLowerBnds, mergedStateUpperBnds). The branch and bound strategy uses this approach (see Variables::get_variables(problem_db) for variables type selection; variables type is passed to the Constraints constructor in Model).

43.54.2 Constructor & Destructor Documentation

43.54.2.1 MergedConstraints (const ProblemDescDB & *problem_db*, const SharedVariablesData & *svd*)

standard constructor In this class, a merged data approach is used in which continuous and discrete arrays are combined into a single continuous array (integrality is relaxed; the converse of truncating reals is not currently supported but could be in the future if needed). Iterators/strategies which use this class include: BranchBndOptimizer. Extract fundamental lower and upper bounds and merge continuous and discrete domains to create mergedDesignLowerBnds, mergedDesignUpperBnds, mergedStateLowerBnds, and mergedStateUpperBnds.

References Constraints::allContinuousLowerBnds, Constraints::allContinuousUpperBnds, Constraints::build_views(), SharedVariablesData::components_totals(), Dakota::copy_data_partial(), ProblemDescDB::get_idv(), ProblemDescDB::get_rdv(), Constraints::manage_linear_constraints(), Dakota::merge_data_partial(), Constraints::sharedVarsData, and SharedVariablesData::vc_lookup().

43.54.3 Member Function Documentation

43.54.3.1 void reshape (const SizetArray & *vc_totals*) `[protected, virtual]`

reshape the lower/upper bound arrays within the Constraints hierarchy Resizes the derived bounds arrays.

Reimplemented from Constraints.

References Constraints::allContinuousLowerBnds, Constraints::allContinuousUpperBnds, and Constraints::build_views().

Referenced by MergedConstraints::MergedConstraints().

The documentation for this class was generated from the following files:

- MergedConstraints.H
- MergedConstraints.C

43.55 MergedVariables Class Reference

Derived class within the Variables hierarchy which employs the merged data view. Inheritance diagram for MergedVariables::

Public Member Functions

- MergedVariables (const ProblemDescDB &problem_db, const std::pair< short, short > &view)

  standard constructor

- MergedVariables (const SharedVariablesData &svd)

  lightweight constructor

- ~MergedVariables ()

  destructor

Protected Member Functions

- void read (std::istream &s)

  read a variables object from an std::istream

- void write (std::ostream &s) const

  write a variables object to an std::ostream

- void write_aprepro (std::ostream &s) const

  write a variables object to an std::ostream in aprepro format

- void read_tabular (std::istream &s)
- void write_tabular (std::ostream &s) const

  write a variables object in tabular format to an std::ostream

- void reshape (const SizetArray &vc_totals)

  reshapes an existing Variables object based on the incoming variablesComponents

- void build_active_views ()

  construct active views of all variables arrays

- void build_inactive_views ()

construct inactive views of all variables arrays

43.55.1 Detailed Description

Derived class within the Variables hierarchy which employs the merged data view. Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MergedVariables derived class combines continuous and discrete domain types but separates design, uncertain, and state variable types. The result is a single continuous array of design variables (mergedDesignVars), a single continuous array of uncertain variables (uncertainVars), and a single continuous array of state variables (mergedStateVars). The branch and bound strategy uses this approach (see Variables::get_variables(problem_db)).

43.55.2 Constructor & Destructor Documentation

43.55.2.1 MergedVariables (const ProblemDescDB & *problem_db*, const std::pair< short, short > & *view*)

standard constructor In this class, a merged data approach is used in which continuous and discrete arrays are combined into a single continuous array (integrality is relaxed; the converse of truncating reals is not currently supported but could be in the future if needed). Iterators/strategies which use this class include: BranchBndOptimizer. Extract fundamental variable types and labels and merge continuous and discrete domains to create aggregate arrays and views.

References Variables::allContinuousVars, Variables::build_views(), SharedVariablesData::components_totals(), Dakota::copy_data_partial(), ProblemDescDB::get_idv(), ProblemDescDB::get_rdv(), Dakota::merge_data_partial(), Variables::sharedVarsData, SharedVariablesData::vc_lookup(), and SharedVariablesData::view().

43.55.3 Member Function Documentation

43.55.3.1 void read_tabular (std::istream & *s*) `[protected, virtual]`

Presumes variables object is appropriately sized to receive data

Reimplemented from Variables.

References Variables::allContinuousVars, SharedVariablesData::components_totals(), Dakota::read_data_partial_tabular(), and Variables::sharedVarsData.

The documentation for this class was generated from the following files:

- MergedVariables.H
- MergedVariables.C

43.56 Minimizer Class Reference

Base class for the optimizer and least squares branches of the iterator hierarchy. Inheritance diagram for Minimizer::

Protected Member Functions

- Minimizer ()

  default constructor

- Minimizer (Model &model)

  standard constructor

- Minimizer (NoDBBaseConstructor, Model &model)

  alternate constructor for "on the fly" instantiations

- Minimizer (NoDBBaseConstructor, size_t num_lin_ineq, size_t num_lin_eq, size_t num_nln_ineq, size_t num_nln_eq)

  alternate constructor for "on the fly" instantiations

- ~Minimizer ()

  destructor

- void initialize_run ()

  utility function to perform common operations prior to pre_run(); typically memory initialization; setting of instance pointers

- void finalize_run ()

  utility function to perform common operations following post_run(); deallocation and resetting of instance pointers

- void initialize_scaling ()

  initialize scaling types, multipliers, and offsets; perform error checking

- void compute_scaling (int object_type, int auto_type, int num_vars, RealVector &lbs, RealVector &ubs, RealVector &targets, const StringArray &scale_strings, const RealVector &scales, IntArray &scale_types, RealVector &scale_mults, RealVector &scale_offsets)

  general helper function for initializing scaling types and factors on a vector of variables, functions, constraints, etc.

- bool compute_scale_factor (const Real lower_bound, const Real upper_bound, Real ∗multiplier, Real ∗offset)

  automatically compute a single scaling factor -- bounds case

- bool compute_scale_factor (const Real target, Real ∗multiplier)

  automatically compute a single scaling factor -- target case

- bool need_resp_trans_byvars (const ShortArray &asv, int start_index, int num_resp)

  determine if response transformation is needed due to variable transformations

- RealVector modify_n2s (const RealVector &native_vars, const IntArray &scale_types, const RealVector &multipliers, const RealVector &offsets) const

  general RealVector mapping from native to scaled variables vectors:

- RealVector modify_s2n (const RealVector &scaled_vars, const IntArray &scale_types, const RealVector &multipliers, const RealVector &offsets) const

  general RealVector mapping from scaled to native variables (and values)

- void response_modify_n2s (const Variables &scaled_vars, const Response &native_response, Response &scaled_response, int native_offset, int recast_offset, int num_responses) const

  map reponses from native to scaled variable space

- void response_modify_s2n (const Variables &native_vars, const Response &scaled_response, Response &native_response, int scaled_offset, int native_offset, int num_responses) const

  map responses from scaled to native variable space

- RealMatrix lin_coeffs_modify_n2s (const RealMatrix &native_coeffs, const RealVector &cv_multipliers, const RealVector &lin_multipliers) const

  general linear coefficients mapping from native to scaled space

- void print_scaling (const String &info, const IntArray &scale_types, const RealVector &scale_mults, const RealVector &scale_offsets, const StringArray &labels)

  print scaling information for a particular response type in tabular form

- Real objective (const RealVector &fn_vals, const RealVector &primary_wts) const

  compute a composite objective value from one or more primary functions

- void objective_gradient (const RealVector &fn_vals, const RealMatrix &fn_grads, const RealVector &primary_wts, RealVector &obj_grad) const

  compute the gradient of the composite objective function

- void objective_hessian (const RealVector &fn_vals, const RealMatrix &fn_grads, const RealSymMatrixArray &fn_hessians, const RealVector &primary_wts, RealSymMatrix &obj_hess) const

  compute the Hessian of the composite objective function

Static Protected Member Functions

- static void variables_recast (const Variables &scaled_vars, Variables &native_vars)

  variables conversion map for RecastModel used in scaling: transform variables from scaled to native (user) space

- static void gnewton_set_recast (const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set)

  conversion of request vector values for the Gauss-Newton Hessian approximation

- static void secondary_resp_recast (const Variables &native_vars, const Variables &scaled_vars, const Response &native_response, Response &scaled_response)

  secondary response conversion map for RecastModel used in scaling: transform constraints (fns, grads, Hessians) from native (user) to

Protected Attributes

- Real constraintTol

  optimizer/least squares constraint tolerance

- Real bigRealBoundSize

  cutoff value for inequality constraint and continuous variable bounds

- int bigIntBoundSize

  cutoff value for discrete variable bounds

- size_t numNonlinearIneqConstraints

  number of nonlinear inequality constraints

- size_t numNonlinearEqConstraints

  number of nonlinear equality constraints

- size_t numLinearIneqConstraints

  number of linear inequality constraints

- size_t numLinearEqConstraints

  number of linear equality constraints

- int numNonlinearConstraints

  total number of nonlinear constraints

- int numLinearConstraints

  total number of linear constraints

- int numConstraints

  total number of linear and nonlinear constraints

- bool optimizationFlag

  flag for use where optimization and NLS must be distinguished

- size_t numUserPrimaryFns

  number of objective functions or least squares terms in the user's model

- size_t numIterPrimaryFns

  number of objective functions or least squares terms in iterator's view

- bool boundConstraintFlag

  convenience flag for denoting the presence of user-specified bound constraints. Used for method selection and error checking.

- bool speculativeFlag

  flag for speculative gradient evaluations

- bool scaleFlag

  flag for overall scaling status

- bool varsScaleFlag

  flag for variables scaling

- bool primaryRespScaleFlag

  flag for primary response scaling

- bool secondaryRespScaleFlag

  flag for secondary response scaling

- IntArray cvScaleTypes

  scale flags for continuous vars.

- RealVector cvScaleMultipliers

  scales for continuous variables

- RealVector cvScaleOffsets

  offsets for continuous variables

- IntArray responseScaleTypes

  scale flags for all responses

- RealVector responseScaleMultipliers

  scales for all responses

- RealVector responseScaleOffsets

  offsets for all responses (zero < for functions, not for nonlin con)

- IntArray linearIneqScaleTypes

  scale flags for linear ineq

- RealVector linearIneqScaleMultipliers

  scales for linear ineq constrs.

- RealVector linearIneqScaleOffsets

  offsets for linear ineq constrs.

- IntArray linearEqScaleTypes

  scale flags for linear eq.

- RealVector linearEqScaleMultipliers

  scales for linear constraints

- RealVector linearEqScaleOffsets

  offsets for linear constraints

- Minimizer * prevMinInstance

  pointer containing previous value of minimizerInstance

- bool vendorNumericalGradFlag

  convenience flag for gradType == numerical && methodSource == vendor

Static Protected Attributes

- static Minimizer * minimizerInstance

  pointer to Minimizer used in static member functions

Friends

- class SOLBase

the SOLBase class is not derived the iterator hierarchy but still needs access to iterator hierarchy data (to avoid attribute replication)

- class SNLLBase

  the SNLLBase class is not derived the iterator hierarchy but still needs access to iterator hierarchy data (to avoid attribute replication)

43.56.1 Detailed Description

Base class for the optimizer and least squares branches of the iterator hierarchy. The Minimizer class provides common data and functionality for Optimizer and LeastSq.

43.56.2 Constructor & Destructor Documentation

43.56.2.1 Minimizer (Model & *model*) `[protected]`

standard constructor This constructor extracts inherited data for the optimizer and least squares branches and performs sanity checking on constraint settings.

References Dakota::abort_handler(), Response::active_set_request_vector(), String::begins(), Iterator::bestResponseArray, Minimizer::bigIntBoundSize, Minimizer::bigRealBoundSize, Minimizer::boundConstraintFlag, Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Response::copy(), Model::current_response(), Model::discrete_int_lower_bounds(), Model::discrete_int_upper_bounds(), Model::discrete_real_lower_bounds(), Model::discrete_real_upper_bounds(), String::ends(), ProblemDescDB::get_sizet(), Iterator::gradientType, Iterator::hessianType, Iterator::maxIterations, Iterator::methodName, Iterator::methodSource, Iterator::numContinuousVars, Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, Iterator::numFinalSolutions, Iterator::numFunctions, Minimizer::numLinearEqConstraints, Minimizer::numLinearIneqConstraints, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Iterator::probDescDB, and Minimizer::vendorNumericalGradFlag.

43.56.3 Member Function Documentation

43.56.3.1 void initialize_run () `[protected, virtual]`

utility function to perform common operations prior to pre_run(); typically memory initialization; setting of instance pointers Perform initialization phases of run sequence, like allocating memory and setting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's initialize_run(), typically _before_ performing its own implementation steps.

Reimplemented from Iterator.

Reimplemented in CONMINOptimizer, LeastSq, Optimizer, DOTOptimizer, NLPQLPOptimizer, SNLLLeastSq, and SNLLOptimizer.

References Model::all_continuous_variables(), Model::all_discrete_int_variables(), Model::all_discrete_real_variables(), Iterator::bestVariablesArray, Iterator::iteratedModel, Minimizer::minimizerInstance, Minimizer::prevMinInstance, and Iterator::subIteratorFlag.

43.56.3.2 void finalize_run () [inline, protected, virtual]

utility function to perform common operations following post_run(); deallocation and resetting of instance pointers Optional: perform finalization phases of run sequence, like deallocating memory and resetting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's finalize_run(), typically _after_ performing its own implementation steps.

Reimplemented from Iterator.

Reimplemented in LeastSq, Optimizer, SNLLLeastSq, and SNLLOptimizer.

References Minimizer::minimizerInstance, and Minimizer::prevMinInstance.

43.56.3.3 void initialize_scaling () [protected]

initialize scaling types, multipliers, and offsets; perform error checking helper function used in constructors of derived classes to set up scaling types, multipliers and offsets when input scaling flag is enabled

References Dakota::abort_handler(), Minimizer::compute_scaling(), Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::continuous_variable_labels(), Model::continuous_variables(), Dakota::copy_data(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, ProblemDescDB::get_dsa(), ProblemDescDB::get_rdv(), Iterator::iteratedModel, Minimizer::lin_coeffs_modify_n2s(), Model::linear_eq_constraint_coeffs(), Model::linear_eq_constraint_targets(), Model::linear_ineq_constraint_coeffs(), Model::linear_ineq_constraint_lower_bounds(), Model::linear_ineq_constraint_upper_bounds(), Minimizer::linearEqScaleMultipliers, Minimizer::linearEqScaleOffsets, Minimizer::linearEqScaleTypes, Minimizer::linearIneqScaleMultipliers, Minimizer::linearIneqScaleOffsets, Minimizer::linearIneqScaleTypes, Model::model_rep(), Minimizer::modify_n2s(), Model::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Iterator::numContinuousVars, Iterator::numFunctions, Minimizer::numLinearEqConstraints, Minimizer::numLinearIneqConstraints, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, Iterator::outputLevel, Minimizer::primaryRespScaleFlag, Minimizer::print_scaling(), Iterator::probDescDB, Model::response_labels(), Minimizer::responseScaleMultipliers, Minimizer::responseScaleOffsets, Minimizer::responseScaleTypes, Minimizer::secondaryRespScaleFlag, RecastModel::submodel_supports_derivative_estimation(), Model::subordinate_model(), Model::supports_derivative_estimation(), and Minimizer::varsScaleFlag.

Referenced by LeastSq::LeastSq(), and Optimizer::Optimizer().

43.56.3.4 void variables_recast (const Variables & *scaled_vars*, Variables & *native_vars*) [static, protected]

variables conversion map for RecastModel used in scaling: transform variables from scaled to native (user) space Variables map from iterator/scaled space to user/native space using a RecastModel.

References Variables::continuous_variable_labels(), Variables::continuous_variables(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Minimizer::minimizerInstance, Minimizer::modify_s2n(), Iterator::outputLevel, and Dakota::write_data().

Referenced by LeastSq::LeastSq(), and Optimizer::Optimizer().

43.56.3.5 void gnewton_set_recast (const Variables & *recast_vars*, const ActiveSet & *recast_set*, ActiveSet & *sub_model_set*) [static, protected]

conversion of request vector values for the Gauss-Newton Hessian approximation For Gauss-Newton Hessian requests, activate the 2 bit and mask the 4 bit.

References ActiveSet::request_value(), and ActiveSet::request_vector().

Referenced by Optimizer::Optimizer(), and SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.56.3.6 void secondary_resp_recast (const Variables & *native_vars*, const Variables & *scaled_vars*, const Response & *native_response*, Response & *iterator_response*) [static, protected]

secondary response conversion map for RecastModel used in scaling: transform constraints (fns, grads, Hessians) from native (user) to Constraint function map from user/native space to iterator/scaled/combined space using a RecastModel.

References Response::active_set_request_vector(), Minimizer::minimizerInstance, Minimizer::need_resp_trans_byvars(), Minimizer::numIterPrimaryFns, Minimizer::numNonlinearConstraints, Minimizer::numUserPrimaryFns, Minimizer::response_modify_n2s(), Minimizer::secondaryRespScaleFlag, and Response::update_partial().

Referenced by LeastSq::LeastSq(), and Optimizer::Optimizer().

43.56.3.7 bool need_resp_trans_byvars (const ShortArray & *asv*, int *start_index*, int *num_resp*) [protected]

determine if response transformation is needed due to variable transformations Determine if variable transformations present and derivatives requested, which implies a response transformation is necessay

References Minimizer::varsScaleFlag.

Referenced by Optimizer::post_run(), LeastSq::post_run(), Optimizer::primary_resp_recast(), LeastSq::primary_resp_recast(), and Minimizer::secondary_resp_recast().

43.56.3.8 RealVector modify_n2s (const RealVector & *native_vars*, const IntArray & *scale_types*, const RealVector & *multipliers*, const RealVector & *offsets*) const [protected]

general RealVector mapping from native to scaled variables vectors: general RealVector mapping from native to scaled variables; loosely, in greatest generality: scaled_var = log((native_var - offset) / multiplier)

Referenced by Minimizer::initialize_scaling().

43.56.3.9 RealVector modify_s2n (const RealVector & *scaled_vars*, const IntArray & *scale_types*, const RealVector & *multipliers*, const RealVector & *offsets*) const [protected]

general RealVector mapping from scaled to native variables (and values) general RealVector mapping from scaled to native variables and/or vals; loosely, in greatest generality: scaled_var = (LOG_BASE^scaled_var) $*$ multiplier + offset

Referenced by SNLLLeastSq::post_run(), Optimizer::post_run(), LeastSq::post_run(), COLINOptimizer::post_run(), and Minimizer::variables_recast().

43.56.3.10 void response_modify_n2s (const Variables & *native_vars*, const Response & *native_response*, Response & *recast_response*, int *native_offset*, int *recast_offset*, int *num_responses*) const [protected]

map reponses from native to scaled variable space scaling response mapping: modifies response from a model (user/native) for use in iterators (scaled) -- not including MOO/NLS objective reductions

References Response::active_set(), Variables::acv(), Variables::all_continuous_variable_ids(), Variables::all_continuous_variables(), Variables::continuous_variable_ids(), Variables::continuous_variables(), Dakota::copy_data(), Variables::cv(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Dakota::find_index(), Response::function_gradient_view(), Response::function_gradients(), Response::function_hessian_view(), Response::function_hessians(), Response::function_labels(), Response::function_value(), Response::function_values(), Variables::icv(), Variables::inactive_continuous_variable_ids(), Variables::inactive_continuous_variables(), Minimizer::numIterPrimaryFns, Iterator::outputLevel, Minimizer::responseScaleMultipliers, Minimizer::responseScaleOffsets, Minimizer::responseScaleTypes, Dakota::write_col_vector_trans(), Dakota::write_data(), and Dakota::write_precision.

Referenced by Optimizer::primary_resp_recast(), LeastSq::primary_resp_recast(), and Minimizer::secondary_resp_recast().

43.56.3.11 void response_modify_s2n (const Variables & *native_vars*, const Response & *scaled_response*, Response & *native_response*, int *scaled_offset*, int *native_offset*, int *num_responses*) const [protected]

map responses from scaled to native variable space scaling response mapping: modifies response from scaled (iterator) to native (user) space -- not including MOO/NLS objective reductions

References Response::active_set(), Variables::acv(), Variables::all_continuous_variable_ids(), Variables::all_continuous_variables(), Variables::continuous_variable_ids(), Variables::continuous_variables(), Dakota::copy_data(), Variables::cv(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Dakota::find_index(), Response::function_gradient_view(), Response::function_gradients(), Response::function_hessian_view(), Response::function_hessians(), Response::function_labels(), Response::function_value(), Response::function_values(), Variables::icv(), Variables::inactive_continuous_variable_ids(), Variables::inactive_continuous_variables(), Minimizer::numIterPrimaryFns, Iterator::outputLevel, Minimizer::responseScaleMultipliers, Minimizer::responseScaleOffsets, Minimizer::responseScaleTypes, Dakota::write_col_vector_trans(), Dakota::write_data(), and Dakota::write_precision.

Referenced by Optimizer::post_run(), and LeastSq::post_run().

43.56.3.12 RealMatrix lin_coeffs_modify_n2s (const RealMatrix & *src_coeffs*, const RealVector & *cv_multipliers*, const RealVector & *lin_multipliers*) const [protected]

general linear coefficients mapping from native to scaled space compute scaled linear constraint matrix given design variable multipliers and linear scaling multipliers. Only scales components corresponding to continuous variables so for src_coeffs of size MxN, lin_multipliers.size() <= M, cv_multipliers.size() <= N

Referenced by Minimizer::initialize_scaling().

43.56.3.13 Real objective (const RealVector & *fn_vals*, const RealVector & *primary_wts*) const [protected]

compute a composite objective value from one or more primary functions The composite objective computation sums up the contributions from one of more primary functions using the primary response fn weights.

References Minimizer::numUserPrimaryFns, and Minimizer::optimizationFlag.

Referenced by SurrBasedLocalMinimizer::approx_subprob_objective_eval(), SurrBasedMinimizer::augmented_lagrangian_merit(), EffGlobalMinimizer::expected_improvement(), SurrBasedMinimizer::lagrangian_merit(), Optimizer::objective_reduction(), SurrBasedMinimizer::penalty_merit(), COLINOptimizer::post_run(), SurrBasedMinimizer::update_filter(), and SurrBasedLocalMinimizer::update_penalty().

43.56.3.14 void objective_gradient (const RealVector & *fn_vals*, const RealMatrix & *fn_grads*, const RealVector & *primary_wts*, RealVector & *obj_grad*) const [protected]

compute the gradient of the composite objective function The composite objective gradient computation combines the contributions from one of more primary function gradients, including the effect of any primary function weights. In the case of a linear mapping (MOO), only the primary function gradients are required, but in the case of a nonlinear mapping (NLS), primary function values are also needed. Within RecastModel::set_mapping(), the active set requests are automatically augmented to make values available when needed, based on nonlinearRespMapping settings.

References Iterator::numContinuousVars, Minimizer::numUserPrimaryFns, and Minimizer::optimizationFlag.

Referenced by SurrBasedLocalMinimizer::approx_subprob_objective_eval(), SurrBasedMinimizer::augmented_lagrangian_gradient(), SurrBasedMinimizer::lagrangian_gradient(), Optimizer::objective_reduction(), SurrBasedMinimizer::penalty_gradient(), and SurrBasedMinimizer::update_lagrange_multipliers().

43.56.3.15 void objective_hessian (const RealVector & *fn_vals*, const RealMatrix & *fn_grads*, const RealSymMatrixArray & *fn_hessians*, const RealVector & *primary_wts*, RealSymMatrix & *obj_hess*) const [protected]

compute the Hessian of the composite objective function The composite objective Hessian computation combines the contributions from one of more primary function Hessians, including the effect of any primary function weights. In the case of a linear mapping (MOO), only the primary function Hessians are required, but in the case of a nonlinear mapping (NLS), primary function values and gradients are also needed in general (gradients only in the case of a Gauss-Newton approximation). Within the default RecastModel::set_mapping(), the active set requests are automatically augmented to make values and gradients available when needed, based on nonlinearRespMapping settings.

References Dakota::abort_handler(), Iterator::numContinuousVars, Minimizer::numUserPrimaryFns, and Minimizer::optimizationFlag.

Referenced by Optimizer::objective_reduction().

The documentation for this class was generated from the following files:

- DakotaMinimizer.H
- DakotaMinimizer.C

43.57 MixedConstraints Class Reference

Derived class within the Constraints hierarchy which employs the default data view (no variable or domain type array merging). Inheritance diagram for MixedConstraints::

Public Member Functions

- MixedConstraints (const SharedVariablesData &svd)

  lightweight constructor

- MixedConstraints (const ProblemDescDB &problem_db, const SharedVariablesData &svd)

  standard constructor

- ~MixedConstraints ()

  destructor

- void write (std::ostream &s) const

  write a variable constraints object to an std::ostream

- void read (std::istream &s)

  read a variable constraints object from an std::istream

Protected Member Functions

- void reshape (const SizetArray &vc_totals)

  reshape the lower/upper bound arrays within the Constraints hierarchy

- void build_active_views ()

  construct active views of all variables bounds arrays

- void build_inactive_views ()

  construct inactive views of all variables bounds arrays

43.57.1 Detailed Description

Derived class within the Constraints hierarchy which employs the default data view (no variable or domain type array merging). Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MixedConstraints derived class separates the design, uncertain, and state variable types as well as the continuous and discrete domain types. The result is separate lower and upper bounds arrays for continuous design, discrete design, uncertain, continuous state, and discrete state variables. This is the default approach, so all iterators and strategies not specifically utilizing the All or Merged views use this approach (see Variables::get_variables(problem_db) for variables type selection; variables type is passed to the Constraints constructor in Model).

43.57.2 Constructor & Destructor Documentation

43.57.2.1 MixedConstraints (const ProblemDescDB & *problem_db*, const SharedVariablesData & *svd*)

standard constructor In this class, mixed continuous/discrete variables are used. Most iterators/strategies use this approach, which is the default in Constraints::get_constraints().

References Constraints::allContinuousLowerBnds, Constraints::allContinuousUpperBnds, Constraints::allDiscreteIntLowerBnds, Constraints::allDiscreteIntUpperBnds, Constraints::allDiscreteRealLowerBnds, Constraints::allDiscreteRealUpperBnds, Constraints::build_views(), SharedVariablesData::components_totals(), Dakota::copy_data_partial(), ProblemDescDB::get_idv(), ProblemDescDB::get_rdv(), Constraints::manage_linear_constraints(), Constraints::numLinearEqCons, Constraints::numLinearIneqCons, Constraints::sharedVarsData, SharedVariablesData::vc_lookup(), and SharedVariablesData::view().

43.57.3 Member Function Documentation

43.57.3.1 void reshape (const SizetArray & *vc_totals*) `[protected, virtual]`

reshape the lower/upper bound arrays within the Constraints hierarchy Resizes the derived bounds arrays.

Reimplemented from Constraints.

References Constraints::allContinuousLowerBnds, Constraints::allContinuousUpperBnds, Constraints::allDiscreteIntLowerBnds, Constraints::allDiscreteIntUpperBnds, Constraints::allDiscreteRealLowerBnds, Constraints::allDiscreteRealUpperBnds, and Constraints::build_views().

Referenced by MixedConstraints::MixedConstraints().

The documentation for this class was generated from the following files:

- MixedConstraints.H
- MixedConstraints.C

43.58 MixedVariables Class Reference

Derived class within the Variables hierarchy which employs the default data view (no variable or domain type array merging). Inheritance diagram for MixedVariables::

Public Member Functions

- MixedVariables (const ProblemDescDB &problem_db, const std::pair< short, short > &view)

  standard constructor

- MixedVariables (const SharedVariablesData &svd)

  lightweight constructor

- ~MixedVariables ()

  destructor

Protected Member Functions

- void read (std::istream &s)

  read a variables object from an std::istream

- void write (std::ostream &s) const

  write a variables object to an std::ostream

- void write_aprepro (std::ostream &s) const

  write a variables object to an std::ostream in aprepro format

- void read_tabular (std::istream &s)
- void write_tabular (std::ostream &s) const

  write a variables object in tabular format to an std::ostream

- void reshape (const SizetArray &vc_totals)

  reshapes an existing Variables object based on the incoming variablesComponents

- void build_active_views ()

  construct active views of all variables arrays

- void build_inactive_views ()

construct inactive views of all variables arrays

43.58.1 Detailed Description

Derived class within the Variables hierarchy which employs the default data view (no variable or domain type array merging). Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MixedVariables derived class separates the design, uncertain, and state variable types as well as the continuous and discrete domain types. The result is separate arrays for continuous design, discrete design, uncertain, continuous state, and discrete state variables. This is the default approach, so all iterators and strategies not specifically utilizing the All or Merged views use this approach (see Variables::get_variables(problem_db)).

43.58.2 Constructor & Destructor Documentation

43.58.2.1 MixedVariables (const ProblemDescDB & *problem_db*, const std::pair< short, short > & *view*)

standard constructor In this class, the distinct approach is used (design, uncertain, and state variable types and continuous and discrete domain types are distinct). Most iterators/strategies use this approach.

References Variables::allContinuousVars, Variables::allDiscreteIntVars, Variables::allDiscreteRealVars, Variables::build_views(), SharedVariablesData::components_totals(), Dakota::copy_data_-partial(), ProblemDescDB::get_idv(), ProblemDescDB::get_rdv(), Variables::sharedVarsData, and SharedVariablesData::vc_lookup().

43.58.3 Member Function Documentation

43.58.3.1 void read_tabular (std::istream & *s*) `[protected, virtual]`

Presumes variables object is already appropriately sized to receive!

Reimplemented from Variables.

References Variables::allContinuousVars, Variables::allDiscreteIntVars, Variables::allDiscreteRealVars, and Dakota::read_data_tabular().

The documentation for this class was generated from the following files:

- MixedVariables.H
- MixedVariables.C

43.59 Model Class Reference

Base class for the model class hierarchy. Inheritance diagram for Model::

Public Member Functions

- Model ()

  default constructor

- Model (ProblemDescDB &problem_db)

  standard constructor for envelope

- Model (const Model &model)

  copy constructor

- virtual ~Model ()

  destructor

- Model operator= (const Model &model)

  assignment operator

- virtual Iterator & subordinate_iterator ()

  return the sub-iterator in nested and surrogate models

- virtual Model & subordinate_model ()

  return a single sub-model defined from subModel in nested and recast models and truth_model() in surrogate models; used for a directed dive through model recursions that may bypass some components.

- virtual Model & surrogate_model ()

  return the approximation sub-model in surrogate models

- virtual Model & truth_model ()

  return the truth sub-model in surrogate models

- virtual void derived_subordinate_models (ModelList &ml, bool recurse_flag)

  portion of subordinate_models()() specific to derived model classes

- virtual void update_from_subordinate_model (bool recurse_flag=true)

propagate vars/labels/bounds/targets from the bottom up

- virtual Interface & interface ()

  return the interface employed by the derived model class, if present: SingleModel::userDefinedInterface, DataFitSurrModel::approxInterface, or NestedModel::optionalInterface

- virtual void primary_response_fn_weights (const RealVector &wts, bool recurse_flag=true)

  set the relative weightings for multiple objective functions or least squares terms

- virtual void surrogate_function_indices (const IntSet &surr_fn_indices)

  set the (currently active) surrogate function index set

- virtual void build_approximation ()

  build a new SurrogateModel approximation

- virtual bool build_approximation (const Variables &vars, const IntResponsePair &response_pr)

  build a new SurrogateModel approximation using/enforcing response at vars

- virtual void update_approximation (bool rebuild_flag)

  replace the approximation data within an existing surrogate based on data updates propagated elsewhere

- virtual void update_approximation (const Variables &vars, const IntResponsePair &response_pr, bool rebuild_flag)

  replace the anchor point data within an existing surrogate

- virtual void update_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map, bool rebuild_flag)

  replace the data points within an existing surrogate

- virtual void append_approximation (bool rebuild_flag)

  append to the existing approximation data within a surrogate based on data updates propagated elsewhere

- virtual void append_approximation (const Variables &vars, const IntResponsePair &response_pr, bool rebuild_flag)

  append a single point to an existing surrogate's data

- virtual void append_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map, bool rebuild_flag)

  append multiple points to an existing surrogate's data

- virtual void pop_approximation (bool save_surr_data)

  remove the previous data set addition to a surrogate (e.g., due to a previous append_approximation() call); flag manages storing of surrogate data for use in a subsequent restore_approximation()

- virtual void restore_approximation ()

  restore a previous approximation data state within a surrogate

- virtual bool restore_available ()

  query for whether a trial increment is restorable within a surrogate

- virtual void finalize_approximation ()

  finalize an approximation by applying all previous trial increments

- virtual void store_approximation ()

  move the current approximation into storage for later combination

- virtual void combine_approximation (short corr_type)

  combine the current approximation with one previously stored

- virtual bool force_rebuild ()

  determine whether a surrogate model rebuild should be forced based on changes in the inactive data

- virtual std::vector< Approximation > & approximations ()

  retrieve the set of Approximations within a DataFitSurrModel

- virtual const Pecos::SurrogateData & approximation_data (size_t index)

  retrieve the approximation data from a particular Approximation instance within a DataFitSurrModel

- virtual const RealVectorArray & approximation_coefficients ()

  retrieve the approximation coefficients from each Approximation within a DataFitSurrModel

- virtual void approximation_coefficients (const RealVectorArray &approx_coeffs)

  set the approximation coefficients for each Approximation within a DataFitSurrModel

- virtual const RealVector & approximation_variances (const RealVector &c_vars)

  retrieve the approximation variances from each Approximation within a DataFitSurrModel

- virtual void surrogate_response_mode (short mode)

  set response computation mode used in SurrogateModels for forming currentResponse

- virtual short surrogate_response_mode () const

  return response computation mode used in SurrogateModels for forming currentResponse

- virtual DiscrepancyCorrection & discrepancy_correction ()

  return the DiscrepancyCorrection object used by SurrogateModels

- virtual void component_parallel_mode (short mode)

  update component parallel mode for supporting parallelism in a model's interface component, sub-model component, or neither component [componentParallelMode = 0 (none), 1 (INTERFACE/APPROX_-INTERFACE/OPTIONAL_INTERFACE/LF_MODEL/SURROGATE_MODEL), or 2 (SUB_MODEL/ACTUAL_-MODEL/HF_MODEL/TRUTH_MODEL)].

- virtual String local_eval_synchronization ()

return derived model synchronization setting

- virtual int local_eval_concurrency ()

  return derived model asynchronous evaluation concurrency

- virtual void serve ()

  Service job requests received from the master. Completes when a termination message is received from stop_servers().

- virtual void stop_servers ()

  Executed by the master to terminate all server operations for a particular model when iteration on the model is complete.

- virtual bool derived_master_overload () const

  Return a flag indicating the combination of multiprocessor evaluations and a dedicated master iterator scheduling. Used in synchronous compute_response functions to prevent the error of trying to run a multiprocessor job on the master.

- virtual void inactive_view (short view, bool recurse_flag=true)

  update the Model's inactive view based on higher level (nested) context

- virtual const String & interface_id () const

  return the interface identifier

- virtual int evaluation_id () const

  Return the value of the evaluation id counter for the Model.

- virtual bool evaluation_cache () const

  Indicates the usage of an evaluation cache by the Model.

- virtual void set_evaluation_reference ()

  Set the reference points for the evaluation counters within the Model.

- virtual void fine_grained_evaluation_counters ()

  Request fine-grained evaluation reporting within the Model.

- virtual void print_evaluation_summary (std::ostream &s, bool minimal_header=false, bool relative_count=true) const

  Print an evaluation summary for the Model.

- ModelList & subordinate_models (bool recurse_flag=true)

  return the sub-models in nested and surrogate models

- void compute_response ()

  Compute the Response at currentVariables (default ActiveSet).

- void compute_response (const ActiveSet &set)

Compute the Response at currentVariables (specified ActiveSet).

- void asynch_compute_response ()

  Spawn an asynchronous job (or jobs) that computes the value of the Response at currentVariables (default ActiveSet).

- void asynch_compute_response (const ActiveSet &set)

  Spawn an asynchronous job (or jobs) that computes the value of the Response at currentVariables (specified ActiveSet).

- const IntResponseMap & synchronize ()

  Execute a blocking scheduling algorithm to collect the complete set of results from a group of asynchronous evaluations.

- const IntResponseMap & synchronize_nowait ()

  Execute a nonblocking scheduling algorithm to collect all available results from a group of asynchronous evaluations.

- void init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  allocate communicator partitions for a model and store configuration in modelPCIterMap

- void init_serial ()

  for cases where init_communicators() will not be called, modify some default settings to behave properly in serial.

- void set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set active parallel configuration for the model (set modelPCIter from modelPCIterMap)

- void free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  deallocate communicator partitions for a model

- void stop_configurations ()

  called from Strategy::init_iterator() for iteratorComm rank 0 to terminate serve_configurations() on other iteratorComm processors

- int serve_configurations ()

  called from Strategy::init_iterator() for iteratorComm rank != 0 to balance init_communicators() calls on iteratorComm rank 0

- void estimate_message_lengths ()

  estimate messageLengths for a model

- void assign_rep (Model *model_rep, bool ref_count_incr=true)

  replaces existing letter with a new one

- size_t tv () const

  returns total number of vars

- size_t cv () const

  returns number of active continuous variables

- size_t div () const

  returns number of active discrete integer vars

- size_t drv () const

  returns number of active discrete real vars

- size_t icv () const

  returns number of inactive continuous variables

- size_t idiv () const

  returns number of inactive discrete integer vars

- size_t idrv () const

  returns number of inactive discrete real vars

- size_t acv () const

  returns total number of continuous variables

- size_t adiv () const

  returns total number of discrete integer vars

- size_t adrv () const

  returns total number of discrete real vars

- void active_variables (const Variables &vars)

  set the active variables in currentVariables

- const RealVector & continuous_variables () const

  return the active continuous variables from currentVariables

- void continuous_variables (const RealVector &c_vars)

  set the active continuous variables in currentVariables

- void continuous_variable (const Real &c_var, const size_t &i)

  set an active continuous variable in currentVariables

- const IntVector & discrete_int_variables () const

  return the active discrete integer variables from currentVariables

- void discrete_int_variables (const IntVector &d_vars)

  set the active discrete integer variables in currentVariables

- void discrete_int_variable (const int &d_var, const size_t &i)

set an active discrete integer variable in currentVariables

- const RealVector & discrete_real_variables () const
  return the active discrete real variables from currentVariables
- void discrete_real_variables (const RealVector &d_vars)
  set the active discrete real variables in currentVariables
- void discrete_real_variable (const Real &d_var, const size_t &i)
  set an active discrete real variable in currentVariables
- UShortMultiArrayConstView continuous_variable_types () const
  return the active continuous variable types from currentVariables
- UShortMultiArrayConstView discrete_int_variable_types () const
  return the active discrete variable types from currentVariables
- UShortMultiArrayConstView discrete_real_variable_types () const
  return the active discrete variable types from currentVariables
- SizetMultiArrayConstView continuous_variable_ids () const
  return the active continuous variable identifiers from currentVariables
- const RealVector & inactive_continuous_variables () const
  return the inactive continuous variables in currentVariables
- void inactive_continuous_variables (const RealVector &i_c_vars)
  set the inactive continuous variables in currentVariables
- const IntVector & inactive_discrete_int_variables () const
  return the inactive discrete variables in currentVariables
- void inactive_discrete_int_variables (const IntVector &i_d_vars)
  set the inactive discrete variables in currentVariables
- const RealVector & inactive_discrete_real_variables () const
  return the inactive discrete variables in currentVariables
- void inactive_discrete_real_variables (const RealVector &i_d_vars)
  set the inactive discrete variables in currentVariables
- SizetMultiArrayConstView inactive_continuous_variable_ids () const
  return the inactive continuous variable identifiers from currentVariables
- const RealVector & all_continuous_variables () const
  return all continuous variables in currentVariables

- void all_continuous_variables (const RealVector &a_c_vars)
  set all continuous variables in currentVariables
- void all_continuous_variable (const Real &a_c_var, const size_t &i)
  set a variable within the all continuous variables in currentVariables
- const IntVector & all_discrete_int_variables () const
  return all discrete variables in currentVariables
- void all_discrete_int_variables (const IntVector &a_d_vars)
  set all discrete variables in currentVariables
- void all_discrete_int_variable (const int &a_d_var, const size_t &i)
  set a variable within the all discrete variables in currentVariables
- const RealVector & all_discrete_real_variables () const
  return all discrete variables in currentVariables
- void all_discrete_real_variables (const RealVector &a_d_vars)
  set all discrete variables in currentVariables
- void all_discrete_real_variable (const Real &a_d_var, const size_t &i)
  set a variable within the all discrete variables in currentVariables
- UShortMultiArrayConstView all_continuous_variable_types () const
  return all continuous variable types from currentVariables
- UShortMultiArrayConstView all_discrete_int_variable_types () const
  return all discrete variable types from currentVariables
- UShortMultiArrayConstView all_discrete_real_variable_types () const
  return all discrete variable types from currentVariables
- SizetMultiArrayConstView all_continuous_variable_ids () const
  return all continuous variable identifiers from currentVariables
- const IntSetArray & discrete_design_set_int_values () const
  return the sets of values available for each of the discrete design set integer variables
- void discrete_design_set_int_values (const IntSetArray &isa)
  define the sets of values available for each of the discrete design set integer variables
- const RealSetArray & discrete_design_set_real_values () const
  return the sets of values available for each of the discrete design set integer variables

- void discrete_design_set_real_values (const RealSetArray &rsa)

  define the sets of values available for each of the discrete design set integer variables

- Pecos::DistributionParams & distribution_parameters ()

  return distribParams

- void distribution_parameters (const Pecos::DistributionParams &dp)

  set distribParams

- const IntSetArray & discrete_state_set_int_values () const

  return the sets of values available for each of the discrete state set integer variables

- void discrete_state_set_int_values (const IntSetArray &isa)

  define the sets of values available for each of the discrete state set integer variables

- const RealSetArray & discrete_state_set_real_values () const

  return the sets of values available for each of the discrete state set integer variables

- void discrete_state_set_real_values (const RealSetArray &rsa)

  define the sets of values available for each of the discrete state set integer variables

- StringMultiArrayConstView continuous_variable_labels () const

  return the active continuous variable labels from currentVariables

- void continuous_variable_labels (StringMultiArrayConstView c_v_labels)

  set the active continuous variable labels in currentVariables

- StringMultiArrayConstView discrete_int_variable_labels () const

  return the active discrete variable labels from currentVariables

- void discrete_int_variable_labels (StringMultiArrayConstView d_v_labels)

  set the active discrete variable labels in currentVariables

- StringMultiArrayConstView discrete_real_variable_labels () const

  return the active discrete variable labels from currentVariables

- void discrete_real_variable_labels (StringMultiArrayConstView d_v_labels)

  set the active discrete variable labels in currentVariables

- StringMultiArrayConstView inactive_continuous_variable_labels () const

  return the inactive continuous variable labels in currentVariables

- void inactive_continuous_variable_labels (StringMultiArrayConstView i_c_v_labels)

  set the inactive continuous variable labels in currentVariables

- StringMultiArrayConstView inactive_discrete_int_variable_labels () const

return the inactive discrete variable labels in currentVariables

- void inactive_discrete_int_variable_labels (StringMultiArrayConstView i_d_v_labels)
  set the inactive discrete variable labels in currentVariables
- StringMultiArrayConstView inactive_discrete_real_variable_labels () const
  return the inactive discrete variable labels in currentVariables
- void inactive_discrete_real_variable_labels (StringMultiArrayConstView i_d_v_labels)
  set the inactive discrete variable labels in currentVariables
- StringMultiArrayConstView all_continuous_variable_labels () const
  return all continuous variable labels in currentVariables
- void all_continuous_variable_labels (StringMultiArrayConstView a_c_v_labels)
  set all continuous variable labels in currentVariables
- void all_continuous_variable_label (const String &a_c_v_label, const size_t &i)
  set a label within the all continuous labels in currentVariables
- StringMultiArrayConstView all_discrete_int_variable_labels () const
  return all discrete variable labels in currentVariables
- void all_discrete_int_variable_labels (StringMultiArrayConstView a_d_v_labels)
  set all discrete variable labels in currentVariables
- void all_discrete_int_variable_label (const String &a_d_v_label, const size_t &i)
  set a label within the all discrete labels in currentVariables
- StringMultiArrayConstView all_discrete_real_variable_labels () const
  return all discrete variable labels in currentVariables
- void all_discrete_real_variable_labels (StringMultiArrayConstView a_d_v_labels)
  set all discrete variable labels in currentVariables
- void all_discrete_real_variable_label (const String &a_d_v_label, const size_t &i)
  set a label within the all discrete labels in currentVariables
- const StringArray & response_labels () const
  return the response labels from currentResponse
- void response_labels (const StringArray &resp_labels)
  set the response labels in currentResponse
- const RealVector & continuous_lower_bounds () const
  return the active continuous lower bounds from userDefinedConstraints

- void continuous_lower_bounds (const RealVector &c_l_bnds)
  set the active continuous lower bounds in userDefinedConstraints
- const RealVector & continuous_upper_bounds () const
  return the active continuous upper bounds from userDefinedConstraints
- void continuous_upper_bounds (const RealVector &c_u_bnds)
  set the active continuous upper bounds in userDefinedConstraints
- const IntVector & discrete_int_lower_bounds () const
  return the active discrete lower bounds from userDefinedConstraints
- void discrete_int_lower_bounds (const IntVector &d_l_bnds)
  set the active discrete lower bounds in userDefinedConstraints
- const IntVector & discrete_int_upper_bounds () const
  return the active discrete upper bounds from userDefinedConstraints
- void discrete_int_upper_bounds (const IntVector &d_u_bnds)
  set the active discrete upper bounds in userDefinedConstraints
- const RealVector & discrete_real_lower_bounds () const
  return the active discrete lower bounds from userDefinedConstraints
- void discrete_real_lower_bounds (const RealVector &d_l_bnds)
  set the active discrete lower bounds in userDefinedConstraints
- const RealVector & discrete_real_upper_bounds () const
  return the active discrete upper bounds from userDefinedConstraints
- void discrete_real_upper_bounds (const RealVector &d_u_bnds)
  set the active discrete upper bounds in userDefinedConstraints
- const RealVector & inactive_continuous_lower_bounds () const
  return the inactive continuous lower bounds in userDefinedConstraints
- void inactive_continuous_lower_bounds (const RealVector &i_c_l_bnds)
  set the inactive continuous lower bounds in userDefinedConstraints
- const RealVector & inactive_continuous_upper_bounds () const
  return the inactive continuous upper bounds in userDefinedConstraints
- void inactive_continuous_upper_bounds (const RealVector &i_c_u_bnds)
  set the inactive continuous upper bounds in userDefinedConstraints

- const IntVector & inactive_discrete_int_lower_bounds () const
  return the inactive discrete lower bounds in userDefinedConstraints
- void inactive_discrete_int_lower_bounds (const IntVector &i_d_l_bnds)
  set the inactive discrete lower bounds in userDefinedConstraints
- const IntVector & inactive_discrete_int_upper_bounds () const
  return the inactive discrete upper bounds in userDefinedConstraints
- void inactive_discrete_int_upper_bounds (const IntVector &i_d_u_bnds)
  set the inactive discrete upper bounds in userDefinedConstraints
- const RealVector & inactive_discrete_real_lower_bounds () const
  return the inactive discrete lower bounds in userDefinedConstraints
- void inactive_discrete_real_lower_bounds (const RealVector &i_d_l_bnds)
  set the inactive discrete lower bounds in userDefinedConstraints
- const RealVector & inactive_discrete_real_upper_bounds () const
  return the inactive discrete upper bounds in userDefinedConstraints
- void inactive_discrete_real_upper_bounds (const RealVector &i_d_u_bnds)
  set the inactive discrete upper bounds in userDefinedConstraints
- const RealVector & all_continuous_lower_bounds () const
  return all continuous lower bounds in userDefinedConstraints
- void all_continuous_lower_bounds (const RealVector &a_c_l_bnds)
  set all continuous lower bounds in userDefinedConstraints
- void all_continuous_lower_bound (const Real &a_c_l_bnd, const size_t &i)
  set a lower bound within continuous lower bounds in userDefinedConstraints
- const RealVector & all_continuous_upper_bounds () const
  return all continuous upper bounds in userDefinedConstraints
- void all_continuous_upper_bounds (const RealVector &a_c_u_bnds)
  set all continuous upper bounds in userDefinedConstraints
- void all_continuous_upper_bound (const Real &a_c_u_bnd, const size_t &i)
  set an upper bound within all continuous upper bounds in userDefinedConstraints
- const IntVector & all_discrete_int_lower_bounds () const
  return all discrete lower bounds in userDefinedConstraints
- void all_discrete_int_lower_bounds (const IntVector &a_d_l_bnds)

set all discrete lower bounds in userDefinedConstraints

- void all_discrete_int_lower_bound (const int &a_d_l_bnd, const size_t &i)
  set a lower bound within all discrete lower bounds in userDefinedConstraints
- const IntVector & all_discrete_int_upper_bounds () const
  return all discrete upper bounds in userDefinedConstraints
- void all_discrete_int_upper_bounds (const IntVector &a_d_u_bnds)
  set all discrete upper bounds in userDefinedConstraints
- void all_discrete_int_upper_bound (const int &a_d_u_bnd, const size_t &i)
  set an upper bound within all discrete upper bounds in userDefinedConstraints
- const RealVector & all_discrete_real_lower_bounds () const
  return all discrete lower bounds in userDefinedConstraints
- void all_discrete_real_lower_bounds (const RealVector &a_d_l_bnds)
  set all discrete lower bounds in userDefinedConstraints
- void all_discrete_real_lower_bound (const Real &a_d_l_bnd, const size_t &i)
  set a lower bound within all discrete lower bounds in userDefinedConstraints
- const RealVector & all_discrete_real_upper_bounds () const
  return all discrete upper bounds in userDefinedConstraints
- void all_discrete_real_upper_bounds (const RealVector &a_d_u_bnds)
  set all discrete upper bounds in userDefinedConstraints
- void all_discrete_real_upper_bound (const Real &a_d_u_bnd, const size_t &i)
  set an upper bound within all discrete upper bounds in userDefinedConstraints
- size_t num_linear_ineq_constraints () const
  return the number of linear inequality constraints
- size_t num_linear_eq_constraints () const
  return the number of linear equality constraints
- const RealMatrix & linear_ineq_constraint_coeffs () const
  return the linear inequality constraint coefficients
- void linear_ineq_constraint_coeffs (const RealMatrix &lin_ineq_coeffs)
  set the linear inequality constraint coefficients
- const RealVector & linear_ineq_constraint_lower_bounds () const
  return the linear inequality constraint lower bounds

- void linear_ineq_constraint_lower_bounds (const RealVector &lin_ineq_l_bnds)
  set the linear inequality constraint lower bounds
- const RealVector & linear_ineq_constraint_upper_bounds () const
  return the linear inequality constraint upper bounds
- void linear_ineq_constraint_upper_bounds (const RealVector &lin_ineq_u_bnds)
  set the linear inequality constraint upper bounds
- const RealMatrix & linear_eq_constraint_coeffs () const
  return the linear equality constraint coefficients
- void linear_eq_constraint_coeffs (const RealMatrix &lin_eq_coeffs)
  set the linear equality constraint coefficients
- const RealVector & linear_eq_constraint_targets () const
  return the linear equality constraint targets
- void linear_eq_constraint_targets (const RealVector &lin_eq_targets)
  set the linear equality constraint targets
- size_t num_nonlinear_ineq_constraints () const
  return the number of nonlinear inequality constraints
- size_t num_nonlinear_eq_constraints () const
  return the number of nonlinear equality constraints
- const RealVector & nonlinear_ineq_constraint_lower_bounds () const
  return the nonlinear inequality constraint lower bounds
- void nonlinear_ineq_constraint_lower_bounds (const RealVector &nln_ineq_l_bnds)
  set the nonlinear inequality constraint lower bounds
- const RealVector & nonlinear_ineq_constraint_upper_bounds () const
  return the nonlinear inequality constraint upper bounds
- void nonlinear_ineq_constraint_upper_bounds (const RealVector &nln_ineq_u_bnds)
  set the nonlinear inequality constraint upper bounds
- const RealVector & nonlinear_eq_constraint_targets () const
  return the nonlinear equality constraint targets
- void nonlinear_eq_constraint_targets (const RealVector &nln_eq_targets)
  set the nonlinear equality constraint targets

- const Variables & current_variables () const

  return the current variables (currentVariables)

- const Constraints & user_defined_constraints () const

  return the user-defined constraints (userDefinedConstraints)

- const Response & current_response () const

  return the current response (currentResponse)

- ProblemDescDB & problem_description_db () const

  return the problem description database (probDescDB)

- ParallelLibrary & parallel_library () const

  return the parallel library (parallelLib)

- const String & model_type () const

  return the model type (modelType)

- const String & surrogate_type () const

  return the surrogate type (surrogateType)

- const String & model_id () const

  return the model identifier (modelId)

- size_t num_functions () const

  return number of functions in currentResponse

- const String & gradient_type () const

  return the gradient evaluation type (gradType)

- const String & method_source () const

  return the numerical gradient evaluation method source (methodSrc)

- const String & interval_type () const

  return the numerical gradient evaluation interval type (intervalType)

- bool ignore_bounds () const

  option for ignoring bounds when numerically estimating derivatives

- bool central_hess () const

  option for using old 2nd-order scheme when computing finite-diff Hessian

- const RealVector & fd_gradient_step_size () const

  return the finite difference gradient step size (fdGradSS)

- const IntList & gradient_id_analytic () const

return the mixed gradient analytic IDs (gradIdAnalytic)

- const IntList & gradient_id_numerical () const

  return the mixed gradient numerical IDs (gradIdNumerical)

- const String & hessian_type () const

  return the Hessian evaluation type (hessType)

- const String & quasi_hessian_type () const

  return the Hessian evaluation type (quasiHessType)

- const RealVector & fd_hessian_by_grad_step_size () const

  return gradient-based finite difference Hessian step size (fdHessByGradSS)

- const RealVector & fd_hessian_by_fn_step_size () const

  return function-based finite difference Hessian step size (fdHessByFnSS)

- const IntList & hessian_id_analytic () const

  return the mixed Hessian analytic IDs (hessIdAnalytic)

- const IntList & hessian_id_numerical () const

  return the mixed Hessian analytic IDs (hessIdNumerical)

- const IntList & hessian_id_quasi () const

  return the mixed Hessian analytic IDs (hessIdQuasi)

- const RealVector & primary_response_fn_weights () const

  get the relative weightings for multiple objective functions or least squares terms. Used by ConcurrentStrategy for Pareto set optimization.

- bool derivative_estimation ()

  indicates potential usage of estimate_derivatives() based on gradType/hessType

- void supports_derivative_estimation (bool sed_flag)

  set whether this model should perform or pass on derivative estimation

- void init_comms_bcast_flag (bool icb_flag)

  set initCommsBcastFlag

- int evaluation_capacity () const

  return the evaluation capacity for use in iterator logic

- int derivative_concurrency () const

  return the gradient concurrency for use in parallel configuration logic

- bool asynch_flag () const

return the asynchronous evaluation flag (asynchEvalFlag)

- void asynch_flag (const bool flag)

  set the asynchronous evaluation flag (asynchEvalFlag)

- short output_level () const

  return the outputLevel

- void output_level (const short level)

  set the outputLevel

- const IntArray & message_lengths () const

  return the array of MPI packed message buffer lengths (messageLengths)

- void parallel_configuration_iterator (const ParConfigLIter &pc_iter)

  set modelPCIter

- const ParConfigLIter & parallel_configuration_iterator () const

  return modelPCIter

- void auto_graphics (const bool flag)

  set modelAutoGraphicsFlag to activate posting of graphics data within compute_response/synchronize functions (automatic graphics posting in the model as opposed to graphics posting at the strategy level).

- bool is_null () const

  function to check modelRep (does this envelope contain a letter)

- Model * model_rep () const

  returns modelRep for access to derived class member functions that are not mapped to the top Model level

- Real FDstep1 (const Real &x0_j, const Real &lb_j, const Real &ub_j, Real h_mag)

  function returning finite-difference step size (affected by bounds)

- Real FDstep2 (const Real &x0_j, const Real &lb_j, const Real &ub_j, Real h)

  function returning second central-difference step size (affected by bounds)

Public Attributes

- bool shortStep

  flags finite-difference step size adjusted by bounds

Protected Member Functions

- Model (BaseConstructor, ProblemDescDB &problem_db)

  constructor initializing the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- Model (NoDBBaseConstructor, ParallelLibrary &parallel_lib, const SharedVariablesData &svd, const ActiveSet &set, short output_level)

  constructor initializing base class for derived model class instances constructed on the fly

- Model (RecastBaseConstructor, ProblemDescDB &problem_db, ParallelLibrary &parallel_lib)

  constructor initializing base class for recast model class instances constructed on the fly

- virtual void derived_compute_response (const ActiveSet &set)

  portion of compute_response() specific to derived model classes

- virtual void derived_asynch_compute_response (const ActiveSet &set)

  portion of asynch_compute_response() specific to derived model classes

- virtual const IntResponseMap & derived_synchronize ()

  portion of synchronize() specific to derived model classes

- virtual const IntResponseMap & derived_synchronize_nowait ()

  portion of synchronize_nowait() specific to derived model classes

- virtual void derived_init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  portion of init_communicators() specific to derived model classes

- virtual void derived_init_serial ()

  portion of init_serial() specific to derived model classes

- virtual void derived_set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  portion of set_communicators() specific to derived model classes

- virtual void derived_free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  portion of free_communicators() specific to derived model classes

Protected Attributes

- Variables currentVariables

  the set of current variables used by the model for performing function evaluations

- size_t numDerivVars

  the number of active continuous variables used in computing most response derivatives (i.e., in places such as quasi-Hessians and response corrections where only the active continuous variables are supported)

- Response currentResponse

  the set of current responses that holds the results of model function evaluations

- size_t numFns

  the number of functions in currentResponse

- Constraints userDefinedConstraints

  Explicit constraints on variables are maintained in the Constraints class hierarchy. Currently, this includes linear constraints and bounds, but could be extended in the future to include other explicit constraints which (1) have their form specified by the user, and (2) are not catalogued in Response since their form and coefficients are published to an iterator at startup.

- String modelType

  type of model: single, nested, or surrogate

- String surrogateType

  type of surrogate model: local_∗, multipoint_∗, global_∗, or hierarchical

- String gradType

  grad type: none,numerical,analytic,mixed

- String methodSrc

  method source: dakota,vendor

- String intervalType

  interval type: forward,central

- bool ignoreBounds

  option to ignore bounds when computing < finite differences

- bool centralHess

  option to use old 2nd-order finite diffs for Hessians

- RealVector fdGradSS

  relative step sizes for numerical gradients

- IntList gradIdAnalytic

  analytic id's for mixed gradients

- IntList gradIdNumerical

  numerical id's for mixed gradients

- String hessType

  Hess type: none,numerical,quasi,analytic,mixed.

- String quasiHessType

  quasi-Hessian type: bfgs, damped_bfgs, sr1

- RealVector fdHessByGradSS

  relative step sizes for numerical Hessians < estimated with 1st-order grad differences

- RealVector fdHessByFnSS

  relative step sizes for numerical Hessians < estimated with 2nd-order fn differences

- IntList hessIdAnalytic

  analytic id's for mixed Hessians

- IntList hessIdNumerical

  numerical id's for mixed Hessians

- IntList hessIdQuasi

  quasi id's for mixed Hessians

- bool supportsEstimDerivs

  whether model should perform or forward < derivative estimation

- IntArray messageLengths

  length of packed MPI buffers containing vars, vars/set, response, and PRPair

- ProblemDescDB & probDescDB

  class member reference to the problem description database

- ParallelLibrary & parallelLib

  class member reference to the parallel library

- ParConfigLIter modelPCIter

  the ParallelConfiguration node used by this model instance

- short componentParallelMode

  the component parallelism mode: 0 (none), 1 (INTERFACE/LF_MODEL), or 2 (SUB_MODEL/HF_-MODEL/TRUTH_MODEL)

- bool asynchEvalFlag

  flags asynch evaluations (local or distributed)

- short outputLevel

  output verbosity level: {SILENT,QUIET,NORMAL,VERBOSE,DEBUG}_OUTPUT

- IntSetArray discreteDesignSetIntValues

  array of IntSet's, each containing the set of allowable integer values corresponding to discrete design integer set variable

- RealSetArray discreteDesignSetRealValues

  array of RealSet's, each containing the set of allowable real values corresponding to discrete design real set variable

- Pecos::DistributionParams distParams

  container for random variable distribution parameters

- IntSetArray discreteStateSetIntValues

  array of IntSet's, each containing the set of allowable integer values corresponding to discrete state integer set variable

- RealSetArray discreteStateSetRealValues

  array of RealSet's, each containing the set of allowable real values corresponding to discrete state real set variable

- RealVector primaryRespFnWts

  primary response function weightings (either weights for multiobjective optimization or weighted least squares)

Private Member Functions

- Model * get_model (ProblemDescDB &problem_db)

  Used by the envelope to instantiate the correct letter class.

- int estimate_derivatives (const ShortArray &map_asv, const ShortArray &fd_grad_asv, const ShortArray &fd_hess_asv, const ShortArray &quasi_hess_asv, const ActiveSet &original_set, const bool asynch_flag)

  evaluate numerical gradients using finite differences. This routine is selected with "method_source dakota" (the default method_source) in the numerical gradient specification.

- void synchronize_derivatives (const Variables &vars, const IntResponseMap &fd_responses, Response &new_response, const ShortArray &fd_grad_asv, const ShortArray &fd_hess_asv, const ShortArray &quasi_hess_asv, const ActiveSet &original_set)

  combine results from an array of finite difference response objects (fd_grad_responses) into a single response (new_response)

- void update_response (const Variables &vars, Response &new_response, const ShortArray &fd_grad_asv, const ShortArray &fd_hess_asv, const ShortArray &quasi_hess_asv, const ActiveSet &original_set, Response &initial_map_response, const RealMatrix &new_fn_grads, const RealSymMatrixArray &new_fn_hessians)

  overlay results to update a response object

- void update_quasi_hessians (const Variables &vars, Response &new_response, const ActiveSet &original_set)

  perform quasi-Newton Hessian updates

- bool manage_asv (const ShortArray &asv_in, ShortArray &map_asv_out, ShortArray &fd_grad_asv_out, ShortArray &fd_hess_asv_out, ShortArray &quasi_hess_asv_out)

  Coordinates usage of estimate_derivatives() calls based on asv_in.

Private Attributes

- String modelId

  model identifier string from the input file

- int modelEvalCntr

  evaluation counter for top-level compute_response() and asynch_compute_response() calls. Differs from lower level counters in case of numerical derivative estimation (several lower level evaluations are assimilated into a single higher level evaluation)

- bool estDerivsFlag

  flags presence of estimated derivatives within a set of calls to asynch_compute_response()

- int evaluationCapacity

  capacity for concurrent evaluations supported by the Model

- std::map< int, ParConfigLIter > modelPCIterMap

  map<> used for tracking modelPCIter instances using concurrency level as the lookup key

- bool initCommsBcastFlag

  flag for determining need to bcast the max concurrency from init_communicators(); set from Strategy::init_iterator()

- bool modelAutoGraphicsFlag

  flag for posting of graphics data within compute_response (automatic graphics posting in the model as opposed to graphics posting at the strategy level)

- ModelList modelList

  used to collect sub-models for subordinate_models()

- VariablesList varsList

  history of vars populated in asynch_compute_response() and used in synchronize().

- std::list< ShortArray > asvList

  if estimate_derivatives() is used, transfers ASVs from asynch_compute_response() to synchronize()

- std::list< ActiveSet > setList

  if estimate_derivatives() is used, transfers ActiveSets from asynch_compute_response() to synchronize()

- BoolList initialMapList

  transfers initial_map flag values from estimate_derivatives() to synchronize_derivatives()

- BoolList dbCaptureList

  transfers db_capture flag values from estimate_derivatives() to synchronize_derivatives()

- ResponseList dbResponseList

  transfers database captures from estimate_derivatives() to synchronize_derivatives()

- RealList deltaList

  transfers deltas from estimate_derivatives() to synchronize_derivatives()

- IntIntMap numFDEvalsMap

  tracks the number of evaluations used within estimate_derivatives(). Used in synchronize() as a key for combining finite difference responses into numerical gradients.

- IntIntMap rawEvalIdMap

  maps from the raw evaluation ids returned by derived_synchronize() and derived_synchronize_nowait() to the corresponding modelEvalCntr id. Used for rekeying responseMap.

- RealVectorArray xPrev

  previous parameter vectors used in computing s for quasi-Newton updates

- RealMatrix fnGradsPrev

  previous gradient vectors used in computing y for quasi-Newton updates

- RealSymMatrixArray quasiHessians

  quasi-Newton Hessian approximations

- SizetArray numQuasiUpdates

  number of quasi-Newton Hessian updates applied

- IntResponseMap responseMap

  used to return a map of responses for asynchronous evaluations in final concatenated form. The similar map in Interface contains raw responses.

- IntResponseMap graphicsRespMap

  used to cache the data returned from derived_synchronize_nowait() prior to sequential input into the graphics

- Model * modelRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing modelRep

43.59.1 Detailed Description

Base class for the model class hierarchy. The Model class is the base class for one of the primary class hierarchies in DAKOTA. The model hierarchy contains a set of variables, an interface, and a set of responses, and an iterator operates on the model to map the variables into responses using the interface. For memory efficiency and enhanced polymorphism, the model hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Model) serves as the envelope and one of the derived classes (selected in Model::get_model()) serves as the letter.

43.59.2 Constructor & Destructor Documentation

43.59.2.1 Model ()

default constructor The default constructor is used in vector<Model> instantiations and for initialization of Model objects contained in Iterator and derived Strategy classes. modelRep is NULL in this case (a populated problem_db is needed to build a meaningful Model object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

43.59.2.2 Model (ProblemDescDB & *problem_db*)

standard constructor for envelope Used in model instantiations within strategy constructors. Envelope constructor only needs to extract enough data to properly execute get_model, since Model(BaseConstructor, problem_db) builds the actual base class data for the derived models.

References Dakota::abort_handler(), Model::get_model(), and Model::modelRep.

43.59.2.3 Model (const Model & *model*)

copy constructor Copy constructor manages sharing of modelRep and incrementing of referenceCount.

References Model::modelRep, and Model::referenceCount.

43.59.2.4 ~Model () `[virtual]`

destructor Destructor decrements referenceCount and only deletes modelRep when referenceCount reaches zero.

References Model::modelRep, and Model::referenceCount.

43.59.2.5 Model (BaseConstructor, ProblemDescDB & *problem_db*) `[protected]`

constructor initializing the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor builds the base class data for all inherited models. get_model() instantiates a derived class and the derived class selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_model() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~Model).

References Model::fdGradSS, Model::fdHessByFnSS, Model::fdHessByGradSS, Model::gradIdNumerical, Model::gradType, Model::hessIdNumerical, and Model::hessType.

43.59.2.6 Model (RecastBaseConstructor, ProblemDescDB & *problem_db*, ParallelLibrary & *parallel_lib*) [protected]

constructor initializing base class for recast model class instances constructed on the fly This constructor also builds the base class data for inherited models. However, it is used for recast models which are instantiated on the fly. Therefore it only initializes a small subset of attributes. Note that parallel_lib is managed separately from problem_db since parallel_lib is needed even in cases where problem_db is an empty envelope (i.e., use of dummy_db in Model(NoDBBaseConstructor) above.

43.59.3 Member Function Documentation

43.59.3.1 Model operator= (const Model & *model*)

assignment operator Assignment operator decrements referenceCount for old modelRep, assigns new modelRep, and increments referenceCount for new modelRep.

References Model::modelRep, and Model::referenceCount.

43.59.3.2 Iterator & subordinate_iterator () [virtual]

return the sub-iterator in nested and surrogate models return by reference requires use of dummy objects, but is important to allow use of assign_rep() since this operation must be performed on the original envelope object.

Reimplemented in DataFitSurrModel, NestedModel, and RecastModel.

References Dakota::dummy_iterator, Model::modelRep, and Model::subordinate_iterator().

Referenced by NonDExpansion::compute_expansion(), NonDExpansion::compute_print_converged_results(), NonDExpansion::compute_print_iteration_results(), NonDExpansion::finalize_sets(), NonDGlobalReliability::get_best_sample(), NonDPolynomialChaos::increment_expansion(), NonDExpansion::increment_sets(), NLPQLPOptimizer::initialize(), NCSUOptimizer::initialize(), DOTOptimizer::initialize(), CONMINOptimizer::initialize(), NonDExpansion::initialize_expansion(), NonDExpansion::initialize_sets(), NonDStochCollocation::initialize_u_space_model(), NonDPolynomialChaos::initialize_u_space_model(), NonDExpansion::initialize_u_space_model(), SurrBasedLocalMinimizer::minimize_surrogates(), SurrBasedGlobalMinimizer::minimize_surrogates(), NonDLocalInterval::NonDLocalInterval(), NonDLocalReliability::NonDLocalReliability(), NonDGlobalReliability::optimize_gaussian_process(), NonDExpansion::refine_expansion(), SOLBase::SOLBase(), RecastModel::subordinate_iterator(), Model::subordinate_iterator(), and NonDExpansion::update_hierarchy().

43.59.3.3 Model & subordinate_model () [virtual]

return a single sub-model defined from subModel in nested and recast models and truth_model() in surrogate models; used for a directed dive through model recursions that may bypass some components. return by reference requires use of dummy objects, but is important to allow use of assign_rep() since this operation must be performed on the original envelope object.

Reimplemented in NestedModel, RecastModel, and SurrogateModel.

References Dakota::dummy_model, Model::modelRep, and Model::subordinate_model().

Referenced by NonDGlobalReliability::expected_feasibility(), NonDGlobalReliability::expected_improvement(), SurrogateModel::force_rebuild(), Minimizer::initialize_scaling(), NonDExpansion::initialize_u_space_model(), NonDGlobalReliability::optimize_gaussian_process(), LeastSq::post_run(), COLINOptimizer::post_run(), Optimizer::primary_resp_recast(), LeastSq::primary_resp_recast(), Model::subordinate_model(), and DataFitSurrModel::update_global().

43.59.3.4 Model & surrogate_model () `[virtual]`

return the approximation sub-model in surrogate models return by reference requires use of dummy objects, but is important to allow use of assign_rep() since this operation must be performed on the original envelope object.

Reimplemented in DataFitSurrModel, HierarchSurrModel, and RecastModel.

References Dakota::dummy_model, Model::modelRep, and Model::surrogate_model().

Referenced by SurrBasedLocalMinimizer::find_center_approx(), SurrBasedLocalMinimizer::minimize_surrogates(), SurrBasedGlobalMinimizer::minimize_surrogates(), SurrBasedLocalMinimizer::SurrBasedLocalMinimizer(), RecastModel::surrogate_model(), and Model::surrogate_model().

43.59.3.5 Model & truth_model () `[virtual]`

return the truth sub-model in surrogate models return by reference requires use of dummy objects, but is important to allow use of assign_rep() since this operation must be performed on the original envelope object.

Reimplemented in DataFitSurrModel, HierarchSurrModel, and RecastModel.

References Dakota::dummy_model, Model::modelRep, and Model::truth_model().

Referenced by SurrogateModel::force_rebuild(), SurrBasedMinimizer::initialize_graphics(), SurrBasedLocalMinimizer::minimize_surrogates(), SurrBasedGlobalMinimizer::minimize_surrogates(), SurrBasedMinimizer::print_results(), SurrogateModel::subordinate_model(), SurrBasedLocalMinimizer::SurrBasedLocalMinimizer(), RecastModel::truth_model(), and Model::truth_model().

43.59.3.6 void update_from_subordinate_model (bool *recurse_flag* = `true`) `[virtual]`

propagate vars/labels/bounds/targets from the bottom up used only for instantiate-on-the-fly model recursions (all RecastModel instantiations and alternate DataFitSurrModel instantiations). Single, Hierarchical, and Nested Models do not redefine the function since they do not support instantiate-on-the-fly. This means that the recursion will stop as soon as it encounters a Model that was instantiated normally, which is appropriate since ProblemDescDB-constructed Models use top-down information flow and do not require bottom-up updating.

Reimplemented in DataFitSurrModel, and RecastModel.

References Model::modelRep, and Model::update_from_subordinate_model().

Referenced by NonDLocalReliability::initialize_class_data(), NonDExpansion::initialize_expansion(), Optimizer::initialize_run(), LeastSq::initialize_run(), EffGlobalMinimizer::minimize_surrogates_on_model(), NonDGlobalReliability::optimize_gaussian_process(), NonDLocalInterval::quantify_uncertainty(), NonDGlobalInterval::quantify_uncertainty(), RecastModel::update_from_subordinate_model(), DataFitSurrModel::update_from_subordinate_model(), and Model::update_from_subordinate_model().

43.59.3.7 Interface & interface () [virtual]

return the interface employed by the derived model class, if present: SingleModel::userDefinedInterface, DataFitSurrModel::approxInterface, or NestedModel::optionalInterface return by reference requires use of dummy objects, but is important to allow use of assign_rep() since this operation must be performed on the original envelope object.

Reimplemented in DataFitSurrModel, NestedModel, RecastModel, and SingleModel.

References Dakota::dummy_interface, Model::interface(), and Model::modelRep.

Referenced by RecastModel::interface(), Model::interface(), and SurrBasedGlobalMinimizer::minimize_surrogates().

43.59.3.8 String local_eval_synchronization () [virtual]

return derived model synchronization setting SingleModels and HierarchSurrModels redefine this virtual function. A default value of "synchronous" prevents asynch local operations for:

- NestedModels: a subIterator can support message passing parallelism, but not asynch local.
- DataFitSurrModels: while asynch evals on approximations will work due to some added bookkeeping, avoiding them is preferable.

Reimplemented in RecastModel, and SingleModel.

References Model::local_eval_synchronization(), and Model::modelRep.

Referenced by Model::init_serial(), RecastModel::local_eval_synchronization(), Model::local_eval_synchronization(), and Model::set_communicators().

43.59.3.9 int local_eval_concurrency () [virtual]

return derived model asynchronous evaluation concurrency SingleModels and HierarchSurrModels redefine this virtual function.

Reimplemented in RecastModel, and SingleModel.

References Model::local_eval_concurrency(), and Model::modelRep.

Referenced by RecastModel::local_eval_concurrency(), Model::local_eval_concurrency(), and Model::set_communicators().

43.59.3.10 const String & interface_id () const [virtual]

return the interface identifier return by reference requires use of dummy objects, but is important to allow use of assign_rep() since this operation must be performed on the original envelope object.

Reimplemented in DataFitSurrModel, NestedModel, RecastModel, and SingleModel.

References Dakota::dummy_interface, Interface::interface_id(), Model::interface_id(), and Model::modelRep.

Referenced by DataFitSurrModel::build_global(), DataFitSurrModel::DataFitSurrModel(), Model::estimate_derivatives(), Model::estimate_message_lengths(), SurrBasedLocalMinimizer::find_center_approx(),

RecastModel::interface_id(), Model::interface_id(), Optimizer::local_objective_recast_retrieve(), SNLLLeastSq::post_run(), COLINOptimizer::post_run(), SurrBasedMinimizer::print_results(), Optimizer::print_results(), LeastSq::print_results(), SequentialHybridStrategy::run_sequential(), DiscrepancyCorrection::search_db(), Analyzer::update_best(), SequentialHybridStrategy::update_local_results(), ConcurrentStrategy::update_local_results(), and NonDLocalReliability::update_mpp_search_data().

43.59.3.11 bool evaluation_cache () const `[virtual]`

Indicates the usage of an evaluation cache by the Model. Only Models including ApplicationInterfaces support an evaluation cache: surrogate, nested, and recast mappings are not stored in the cache. Possible exceptions: HierarchSurrModel, NestedModel::optionalInterface.

Reimplemented in SingleModel.

References Model::evaluation_cache(), and Model::modelRep.

Referenced by DataFitSurrModel::DataFitSurrModel(), and Model::evaluation_cache().

43.59.3.12 ModelList & subordinate_models (bool *recurse_flag* = `true`)

return the sub-models in nested and surrogate models since modelList is built with list insertions (using envelope copies), these models may not be used for model.assign_rep() since this operation must be performed on the original envelope object. They may, however, be used for letter-based operations (including assign_rep() on letter contents such as an interface).

References Model::derived_subordinate_models(), Model::modelList, Model::modelRep, and Model::subordinate_models().

Referenced by NLPQLPOptimizer::initialize(), NCSUOptimizer::initialize(), DOTOptimizer::initialize(), CONMINOptimizer::initialize(), NonDLocalInterval::NonDLocalInterval(), NonDLocalReliability::NonDLocalReliability(), SOLBase::SOLBase(), Model::subordinate_models(), and SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.59.3.13 void init_communicators (const int & *max_iterator_concurrency*, bool *recurse_flag* = `true`)

allocate communicator partitions for a model and store configuration in modelPCIterMap The init_communicators() and derived_init_communicators() functions are stuctured to avoid performing the messageLengths estimation more than once. init_communicators() (not virtual) performs the estimation and then forwards the results to derived_init_communicators (virtual) which uses the data in different contexts.

References ParallelLibrary::bcast_i(), Model::derived_init_communicators(), Model::estimate_message_lengths(), ParallelLibrary::increment_parallel_configuration(), Model::init_communicators(), Model::initCommsBcastFlag, Model::messageLengths, Model::modelPCIter, Model::modelPCIterMap, Model::modelRep, ParallelLibrary::parallel_configuration_iterator(), and Model::parallelLib.

Referenced by APPSOptimizer::APPSOptimizer(), COLINOptimizer::COLINOptimizer(), NonDExpansion::construct_expansion_sampler(), RecastModel::derived_init_communicators(), NestedModel::derived_init_communicators(), HierarchSurrModel::derived_init_communicators(), DataFitSurrModel::derived_init_communicators(), EffGlobalMinimizer::EffGlobalMinimizer(), Model::init_communicators(), Strategy::init_iterator(), JEGAOptimizer::JEGAOptimizer(), LeastSq::LeastSq(),

NonDLocalReliability::method_recourse(), NonDLocalInterval::method_recourse(), NonDBayesCalibration::NonDBayesCalibration(), NonDGlobalInterval::NonDGlobalInterval(), NonDGlobalReliability::NonDGlobalReliability(), NonDGPMSABayesCalibration::NonDGPMSABayesCalibration(), NonDLHSInterval::NonDLHSInterval(), NonDLocalInterval::NonDLocalInterval(), NonDLocalReliability::NonDLocalReliability(), NonDPolynomialChaos::NonDPolynomialChaos(), NonDStochCollocation::NonDStochCollocation(), Optimizer::Optimizer(), Model::serve_configurations(), SNLLOptimizer::SNLLOptimizer(), SurrBasedGlobalMinimizer::SurrBasedGlobalMinimizer(), and SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.59.3.14 void init_serial ()

for cases where init_communicators() will not be called, modify some default settings to behave properly in serial. The init_serial() and derived_init_serial() functions are stuctured to separate base class (common) operations from derived class (specialized) operations.

References Model::asynchEvalFlag, Model::derived_init_serial(), Model::init_serial(), Model::local_eval_synchronization(), and Model::modelRep.

Referenced by RecastModel::derived_init_serial(), NestedModel::derived_init_serial(), HierarchSurrModel::derived_init_serial(), DataFitSurrModel::derived_init_serial(), and Model::init_serial().

43.59.3.15 void estimate_message_lengths ()

estimate messageLengths for a model This functionality has been pulled out of init_communicators() and defined separately so that it may be used in those cases when messageLengths is needed but model.init_communicators() is not called, e.g., for the master processor in the self-scheduling of a concurrent iterator strategy.

References Response::active_set_derivative_vector(), Response::copy(), Model::currentResponse, Model::currentVariables, Model::estimate_message_lengths(), Model::interface_id(), Model::messageLengths, Model::modelRep, Model::numFns, Model::parallelLib, MPIPackBuffer::reset(), MPIPackBuffer::size(), and ParallelLibrary::world_size().

Referenced by ConcurrentStrategy::ConcurrentStrategy(), Model::estimate_message_lengths(), and Model::init_communicators().

43.59.3.16 void assign_rep (Model * *model_rep*, bool *ref_count_incr* = `true`)

replaces existing letter with a new one Similar to the assignment operator, the assign_rep() function decrements referenceCount for the old modelRep and assigns the new modelRep. It is different in that it is used for publishing derived class letters to existing envelopes, as opposed to sharing representations among multiple envelopes (in particular, assign_rep is passed a letter object and operator= is passed an envelope object). Letter assignment supports two models as governed by ref_count_incr:

- ref_count_incr = true (default): the incoming letter belongs to another envelope. In this case, increment the reference count in the normal manner so that deallocation of the letter is handled properly.
- ref_count_incr = false: the incoming letter is instantiated on the fly and has no envelope. This case is modeled after get_model(): a letter is dynamically allocated using new and passed into assign_rep, the letter's reference count is not incremented, and the letter is not remotely deleted (its memory management is passed over to the envelope).

References Dakota::abort_handler(), Model::modelRep, and Model::referenceCount.

Referenced by NonD::construct_u_space_model(), EffGlobalMinimizer::EffGlobalMinimizer(), LeastSq::LeastSq(), NonDBayesCalibration::NonDBayesCalibration(), NonDGlobalInterval::NonDGlobalInterval(), NonDGlobalReliability::NonDGlobalReliability(), NonDLocalInterval::NonDLocalInterval(), NonDLocalReliability::NonDLocalReliability(), NonDPolynomialChaos::NonDPolynomialChaos(), NonDStochCollocation::NonDStochCollocation(), Optimizer::Optimizer(), and SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.59.3.17 int derivative_concurrency () const

return the gradient concurrency for use in parallel configuration logic This function assumes derivatives with respect to the active continuous variables. Therefore, concurrency with respect to the inactive continuous variables is not captured.

References Dakota::contains(), Model::derivative_concurrency(), Model::gradIdAnalytic, Model::gradType, Model::hessIdNumerical, Model::hessType, Model::intervalType, Model::methodSrc, Model::modelRep, and Model::numDerivVars.

Referenced by Model::derivative_concurrency(), HierarchSurrModel::derived_free_communicators(), HierarchSurrModel::derived_init_communicators(), DataFitSurrModel::derived_init_communicators(), HierarchSurrModel::derived_set_communicators(), NonDExpansion::initialize_u_space_model(), NonDPolynomialChaos::NonDPolynomialChaos(), NonDStochCollocation::NonDStochCollocation(), Iterator::num_samples(), NonDPolynomialChaos::∼NonDPolynomialChaos(), and NonDStochCollocation::∼NonDStochCollocation().

43.59.3.18 Real FDstep1 (const Real & *x0_j*, const Real & *lb_j*, const Real & *ub_j*, Real *h_mag*)

function returning finite-difference step size (affected by bounds) Auxiliary function to compute forward or first central-difference step size.

References Model::shortStep.

Referenced by Model::estimate_derivatives().

43.59.3.19 Real FDstep2 (const Real & *x0_j*, const Real & *lb_j*, const Real & *ub_j*, Real *h*)

function returning second central-difference step size (affected by bounds) Auxiliary function to second central-difference step size, honoring bounds.

References Model::ignoreBounds, and Model::shortStep.

Referenced by Model::estimate_derivatives().

43.59.3.20 Model ∗ get_model (ProblemDescDB & *problem_db*) `[private]`

Used by the envelope to instantiate the correct letter class. Used only by the envelope constructor to initialize modelRep to the appropriate derived type, as given by the modelType attribute.

References ProblemDescDB::get_string(), Model::model_type(), and Model::modelType.

Referenced by Model::Model().

43.59.3.21 int estimate_derivatives (const ShortArray & *map_asv*, const ShortArray & *fd_grad_asv*, const ShortArray & *fd_hess_asv*, const ShortArray & *quasi_hess_asv*, const ActiveSet & *original_set*, const bool *asynch_flag*) [private]

evaluate numerical gradients using finite differences. This routine is selected with "method_source dakota" (the default method_source) in the numerical gradient specification. Estimate derivatives by computing finite difference gradients, finite difference Hessians, and/or quasi-Newton Hessians. The total number of finite difference evaluations is returned for use by synchronize() to track response arrays, and it could be used to improve management of max_function_evaluations within the iterators.

! new logic

References Model::all_continuous_lower_bounds(), Model::all_continuous_upper_bounds(), Model::all_continuous_variable_ids(), Variables::all_continuous_variables(), Model::centralHess, Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::continuous_variable_ids(), Variables::continuous_variable_ids(), Variables::continuous_variables(), Response::copy(), Dakota::copy_data(), Model::currentResponse, Model::currentVariables, Dakota::data_pairs, Model::dbCaptureList, Model::dbResponseList, Model::deltaList, ActiveSet::derivative_vector(), Model::derived_asynch_compute_response(), Model::derived_compute_response(), Model::fdGradSS, Model::fdHessByFnSS, Model::fdHessByGradSS, Model::FDstep1(), Model::FDstep2(), Dakota::find_index(), Response::function_gradients(), Response::function_values(), Model::ignoreBounds, Model::inactive_continuous_lower_bounds(), Model::inactive_continuous_upper_bounds(), Model::inactive_continuous_variable_ids(), Variables::inactive_continuous_variable_ids(), Variables::inactive_continuous_variables(), Model::initialMapList, Model::interface_id(), Model::intervalType, Dakota::lookup_by_val(), Model::numFns, Model::outputLevel, ActiveSet::request_vector(), Model::shortStep, and Model::update_response().

Referenced by Model::asynch_compute_response(), and Model::compute_response().

43.59.3.22 void synchronize_derivatives (const Variables & *vars*, const IntResponseMap & *fd_responses*, Response & *new_response*, const ShortArray & *fd_grad_asv*, const ShortArray & *fd_hess_asv*, const ShortArray & *quasi_hess_asv*, const ActiveSet & *original_set*) [private]

combine results from an array of finite difference response objects (fd_grad_responses) into a single response (new_response) Merge an array of fd_responses into a single new_response. This function is used both by synchronous compute_response() for the case of asynchronous estimate_derivatives() and by synchronize() for the case where one or more asynch_compute_response() calls has employed asynchronous estimate_derivatives().

!

References Response::active_set(), Model::acv(), Variables::all_continuous_variable_ids(), Model::centralHess, Variables::continuous_variable_ids(), Response::copy(), Model::currentResponse, Model::currentVariables, Model::cv(), Model::dbCaptureList, Model::dbResponseList, Model::deltaList, ActiveSet::derivative_vector(), Dakota::find_index(), Response::function_gradients(), Response::function_values(), Model::icv(), Variables::inactive_continuous_variable_ids(), Model::initialMapList, Model::intervalType, Model::numFns, ActiveSet::request_values(), Response::reset_inactive(), and Model::update_response().

Referenced by Model::compute_response(), and Model::synchronize().

43.59.3.23 void update_response (const Variables & *vars*, Response & *new_response*, const ShortArray & *fd_grad_asv*, const ShortArray & *fd_hess_asv*, const ShortArray & *quasi_hess_asv*, const ActiveSet & *original_set*, Response & *initial_map_response*, const RealMatrix & *new_fn_grads*, const RealSymMatrixArray & *new_fn_hessians*) `[private]`

overlay results to update a response object Overlay the initial_map_response with numerically estimated new_fn_-grads and new_fn_hessians to populate new_response as governed by asv vectors. Quasi-Newton secant Hessian updates are also performed here, since this is where the gradient data needed for the updates is first consolidated. Convenience function used by estimate_derivatives() for the synchronous case and by synchronize_derivatives() for the asynchronous case.

References Response::active_set_request_vector(), Variables::continuous_variable_ids(), Response::copy(), Model::currentResponse, Model::currentVariables, ActiveSet::derivative_vector(), Response::function_-gradients(), Response::function_hessians(), Response::function_values(), Model::hessIdQuasi, Model::hessType, Response::is_null(), Model::numFns, Model::outputLevel, Model::quasiHessians, ActiveSet::request_-vector(), Response::reset_inactive(), Model::supportsEstimDerivs, Model::surrogate_response_mode(), and Model::update_quasi_hessians().

Referenced by Model::estimate_derivatives(), and Model::synchronize_derivatives().

43.59.3.24 void update_quasi_hessians (const Variables & *vars*, Response & *new_response*, const ActiveSet & *original_set*) `[private]`

perform quasi-Newton Hessian updates quasi-Newton updates are performed for approximating response function Hessians using BFGS or SR1 formulations. These Hessians are supported only for the active continuous variables, and a check is performed on the DVV prior to invoking the function.

References Dakota::contains(), Variables::continuous_variables(), Dakota::copy_data(), Model::fnGradsPrev, Response::function_gradients(), Model::hessIdQuasi, Model::hessType, Model::modelType, Model::numDerivVars, Model::numFns, Model::numQuasiUpdates, Model::outputLevel, Model::quasiHessians, Model::quasiHessType, ActiveSet::request_vector(), and Model::xPrev.

Referenced by Model::update_response().

43.59.3.25 bool manage_asv (const ShortArray & *asv_in*, ShortArray & *map_asv_out*, ShortArray & *fd_grad_asv_out*, ShortArray & *fd_hess_asv_out*, ShortArray & *quasi_hess_asv_out*) `[private]`

Coordinates usage of estimate_derivatives() calls based on asv_in. Splits asv_in total request into map_asv_out, fd_grad_asv_out, fd_hess_asv_out, and quasi_hess_asv_out as governed by the responses specification. If the returned use_est_deriv is true, then these asv outputs are used by estimate_derivatives() for the initial map, finite difference gradient evals, finite difference Hessian evals, and quasi-Hessian updates, respectively. If the returned use_est_deriv is false, then only map_asv_out is used.

References Dakota::abort_handler(), Dakota::contains(), Model::gradIdAnalytic, Model::gradIdNumerical, Model::gradType, Model::hessIdAnalytic, Model::hessIdNumerical, Model::hessIdQuasi, Model::hessType, Model::intervalType, Model::methodSrc, Model::supportsEstimDerivs, and Model::surrogate_response_mode().

Referenced by Model::asynch_compute_response(), and Model::compute_response().

The documentation for this class was generated from the following files:

- DakotaModel.H
- DakotaModel.C

43.60 MPIPackBuffer Class Reference

Class for packing MPI message buffers.

Public Member Functions

- MPIPackBuffer (int size_=1024)

  Constructor, which allows the default buffer size to be set.

- ~MPIPackBuffer ()

  Desctructor.

- const char * buf ()

  Returns a pointer to the internal buffer that has been packed.

- int size ()

  The number of bytes of packed data.

- int capacity ()

  the allocated size of Buffer.

- void reset ()

  Resets the buffer index in order to reuse the internal buffer.

- void pack (const int *data, const int num=1)

  *Pack one or more **int's**.*

- void pack (const u_int *data, const int num=1)

  *Pack one or more **unsigned int's**.*

- void pack (const long *data, const int num=1)

  *Pack one or more **long's**.*

- void pack (const u_long *data, const int num=1)

  *Pack one or more **unsigned long's**.*

- void pack (const short *data, const int num=1)

  *Pack one or more **short's**.*

- void pack (const u_short *data, const int num=1)

  *Pack one or more **unsigned short's**.*

- void pack (const char *data, const int num=1)

  *Pack one or more **char's**.*

- void pack (const u_char *data, const int num=1)

  *Pack one or more **unsigned char's**.*

- void pack (const double *data, const int num=1)

  *Pack one or more **double's**.*

- void pack (const float *data, const int num=1)

  *Pack one or more **float's**.*

- void pack (const bool *data, const int num=1)

  *Pack one or more **bool's**.*

- void pack (const int &data)

  *Pack a **int**.*

- void pack (const u_int &data)

  *Pack a **unsigned int**.*

- void pack (const long &data)

  *Pack a **long**.*

- void pack (const u_long &data)

  *Pack a **unsigned long**.*

- void pack (const short &data)

  *Pack a **short**.*

- void pack (const u_short &data)

  *Pack a **unsigned short**.*

- void pack (const char &data)

  *Pack a **char**.*

- void pack (const u_char &data)

  *Pack a **unsigned char**.*

- void pack (const double &data)

  *Pack a **double**.*

- void pack (const float &data)

  *Pack a **float**.*

- void pack (const bool &data)

  *Pack a **bool**.*

Protected Member Functions

- void resize (const int newsize)

  Resizes the internal buffer.

Protected Attributes

- char * Buffer

  The internal buffer for packing.

- int Index

  The index into the current buffer.

- int Size

  The total size that has been allocated for the buffer.

43.60.1 Detailed Description

Class for packing MPI message buffers. A class that provides a facility for packing message buffers using the MPI_Pack facility. The `MPIPackBuffer` class dynamically resizes the internal buffer to contain enough memory to pack the entire object. When deleted, the `MPIPackBuffer` object deletes this internal buffer. This class is based on the Dakota_Version_3_0 version of utilib::PackBuffer from utilib/src/io/PackBuf.[cpp,h]

The documentation for this class was generated from the following files:

- MPIPackBuffer.H
- MPIPackBuffer.C

43.61 MPIUnpackBuffer Class Reference

Class for unpacking MPI message buffers.

Public Member Functions

- void setup (char *buf_, int size_, bool flag_=false)

  Method that does the setup for the constructors.

- MPIUnpackBuffer ()

  Default constructor.

- MPIUnpackBuffer (int size_)

  Constructor that specifies the size of the buffer.

- MPIUnpackBuffer (char *buf_, int size_, bool flag_=false)

  Constructor that sets the internal buffer to the given array.

- ~MPIUnpackBuffer ()

  Destructor.

- void resize (const int newsize)

  Resizes the internal buffer.

- const char * buf ()

  Returns a pointer to the internal buffer.

- int size ()

  Returns the length of the buffer.

- int curr ()

  Returns the number of bytes that have been unpacked from the buffer.

- void reset ()

  Resets the index of the internal buffer.

- void unpack (int *data, const int num=1)

  *Unpack one or more **int's**.*

- void unpack (u_int *data, const int num=1)

  *Unpack one or more **unsigned int's**.*

- void unpack (long *data, const int num=1)

  *Unpack one or more **long's**.*

- void unpack (u_long *data, const int num=1)
  *Unpack one or more **unsigned long's**.*
- void unpack (short *data, const int num=1)
  *Unpack one or more **short's**.*
- void unpack (u_short *data, const int num=1)
  *Unpack one or more **unsigned short's**.*
- void unpack (char *data, const int num=1)
  *Unpack one or more **char's**.*
- void unpack (u_char *data, const int num=1)
  *Unpack one or more **unsigned char's**.*
- void unpack (double *data, const int num=1)
  *Unpack one or more **double's**.*
- void unpack (float *data, const int num=1)
  *Unpack one or more **float's**.*
- void unpack (bool *data, const int num=1)
  *Unpack one or more **bool's**.*
- void unpack (int &data)
  *Unpack a **int**.*
- void unpack (u_int &data)
  *Unpack a **unsigned int**.*
- void unpack (long &data)
  *Unpack a **long**.*
- void unpack (u_long &data)
  *Unpack a **unsigned long**.*
- void unpack (short &data)
  *Unpack a **short**.*
- void unpack (u_short &data)
  *Unpack a **unsigned short**.*
- void unpack (char &data)
  *Unpack a **char**.*
- void unpack (u_char &data)

*Unpack a **unsigned char**.*

- void unpack (double &data)

  *Unpack a **double**.*

- void unpack (float &data)

  *Unpack a **float**.*

- void unpack (bool &data)

  *Unpack a **bool**.*

Protected Attributes

- char * Buffer

  The internal buffer for unpacking.

- int Index

  The index into the current buffer.

- int Size

  The total size that has been allocated for the buffer.

- bool ownFlag

  If `TRUE`, then this class owns the internal buffer.

43.61.1 Detailed Description

Class for unpacking MPI message buffers. A class that provides a facility for unpacking message buffers using the MPI_Unpack facility. This class is based on the Dakota_Version_3_0 version of utilib::UnPackBuffer from utilib/src/io/PackBuf.[cpp,h]

The documentation for this class was generated from the following files:

- MPIPackBuffer.H
- MPIPackBuffer.C

43.62 NCSUOptimizer Class Reference

Wrapper class for the NCSU DIRECT optimization library. Inheritance diagram for NCSUOptimizer::

Public Member Functions

- NCSUOptimizer (Model &model)

  standard constructor

- NCSUOptimizer (Model &model, const int &max_iter, const int &max_eval, double min_box_size=-1., double vol_box_size=-1., double solution_target=-DBL_MAX)

  alternate constructor for instantiations "on the fly"

- NCSUOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor for Iterator instantiations by name

- NCSUOptimizer (const RealVector &var_l_bnds, const RealVector &var_u_bnds, const int &max_iter, const int &max_eval, double(∗user_obj_eval)(const RealVector &x), double min_box_size=-1., double vol_box_size=-1., double solution_target=-DBL_MAX)

  alternate constructor for instantiations "on the fly"

- ∼NCSUOptimizer ()

  destructor

- void find_optimum ()

  Used within the optimizer branch for computing the optimal solution. Redefines the run virtual function for the optimizer branch.

Private Member Functions

- void initialize ()

  shared code among model-based constructors

- void check_inputs ()

verify problem respects NCSU DIRECT Fortran limits

Static Private Member Functions

- static int objective_eval (int ∗n, double c[ ], double l[ ], double u[ ], int point[ ], int ∗maxI, int ∗start, int ∗maxfunc, double fvec[ ], int iidata[ ], int ∗iisize, double ddata[ ], int ∗idsize, char cdata[ ], int ∗icsize)

  'fep' in Griffin-modified NCSUDirect: computes the value of the objective function (potentially at multiple points, passed by function pointer to NCSUDirect). Include unscaling from DIRECT.

Private Attributes

- short setUpType

  controls iteration mode: SETUP_MODEL (normal usage) or SETUP_USERFUNC (user-supplied functions mode for "on the fly" instantiations). see enum in NCSUOptimizer.C NonDGlobalReliability currently uses the model mode. GaussProcApproximation currently uses the user_functions mode.

- Real minBoxSize

  holds the minimum boxsize

- Real volBoxSize

  hold the minimum volume boxsize

- Real solutionTarget

  holds the solution target minimum to drive towards

- RealVector lowerBounds

  holds variable lower bounds passed in for "user_functions" mode.

- RealVector upperBounds

  holds variable upper bounds passed in for "user_functions" mode.

- double(∗ userObjectiveEval)(const RealVector &x)

  holds function pointer for objective function evaluator passed in for "user_functions" mode.

Static Private Attributes

- static NCSUOptimizer ∗ ncsudirectInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.62.1 Detailed Description

Wrapper class for the NCSU DIRECT optimization library. The NCSUOptimizer class provides a wrapper for a Fortran 77 implementation of the DIRECT algorithm developed at North Carolina State University. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or accessed through a static pointer.

The user input mappings are as follows:

43.62.2 Constructor & Destructor Documentation

43.62.2.1 NCSUOptimizer (Model & *model*)

standard constructor This is the standard constructor with method specification support.

References NCSUOptimizer::check_inputs(), and NCSUOptimizer::initialize().

43.62.2.2 NCSUOptimizer (Model & *model*, const int & *max_iter*, const int & *max_eval*, double *min_box_size* = `-1.`, double *vol_box_size* = `-1.`, double *solution_target* = `-DBL_MAX`)

alternate constructor for instantiations "on the fly" This is an alternate constructor for instantiations on the fly using a Model but no ProblemDescDB.

References NCSUOptimizer::check_inputs(), NCSUOptimizer::initialize(), Iterator::maxFunctionEvals, and Iterator::maxIterations.

43.62.2.3 NCSUOptimizer (NoDBBaseConstructor, Model & *model*)

alternate constructor for Iterator instantiations by name This is an alternate constructor for Iterator instantiations by name using a Model but no ProblemDescDB.

References NCSUOptimizer::check_inputs(), and NCSUOptimizer::initialize().

43.62.2.4 NCSUOptimizer (const RealVector & *var_l_bnds*, const RealVector & *var_u_bnds*, const int & *max_iter*, const int & *max_eval*, double(∗)(const RealVector &x) *user_obj_eval*, double *min_box_size* = `-1.`, double *vol_box_size* = `-1.`, double *solution_target* = `-DBL_MAX`)

alternate constructor for instantiations "on the fly" This is an alternate constructor for performing an optimization using the passed in objective function pointer.

References NCSUOptimizer::check_inputs(), Iterator::maxFunctionEvals, and Iterator::maxIterations.

43.62.3 Member Function Documentation

43.62.3.1 int objective_eval (int * *n*, double *c*[ ], double *l*[ ], double *u*[ ], int *point*[ ], int * *maxI*, int * *start*, int * *maxfunc*, double *fvec*[ ], int *iidata*[ ], int * *iisize*, double *ddata*[ ], int * *idsize*, char *cdata*[ ], int * *icsize*) `[static, private]`

'fep' in Griffin-modified NCSUDirect: computes the value of the objective function (potentially at multiple points, passed by function pointer to NCSUDirect). Include unscaling from DIRECT. Modified batch evaluator that accepts multiple points and returns corresponding vector of functions in fvec. Must be used with modified DIRECT src (DIRbatch.f).

References Model::asynch_compute_response(), Model::asynch_flag(), Model::compute_response(), Model::continuous_variables(), Model::current_response(), Response::function_value(), Iterator::iteratedModel, NCSUOptimizer::ncsudirectInstance, NCSUOptimizer::setUpType, Model::synchronize(), and NCSUOptimizer::userObjectiveEval.

Referenced by NCSUOptimizer::find_optimum().

The documentation for this class was generated from the following files:

- NCSUOptimizer.H
- NCSUOptimizer.C

43.63 NestedModel Class Reference

Derived model class which performs a complete sub-iterator execution within every evaluation of the model. Inheritance diagram for NestedModel::

Public Member Functions

- NestedModel (ProblemDescDB &problem_db)

  constructor

- ~NestedModel ()

  destructor

Protected Member Functions

- void derived_compute_response (const ActiveSet &set)

  portion of compute_response() specific to NestedModel

- void derived_asynch_compute_response (const ActiveSet &set)

  portion of asynch_compute_response() specific to NestedModel

- Iterator & subordinate_iterator ()

  return subIterator

- Model & subordinate_model ()

  return subModel

- void derived_subordinate_models (ModelList &ml, bool recurse_flag)

  return subModel

- Interface & interface ()

  return optionalInterface

- void surrogate_response_mode (bool mode)

  pass a bypass request on to the subModel for any lower-level surrogates

- void component_parallel_mode (short mode)

  update component parallel mode for supporting parallelism in optionalInterface and subModel

- bool derived_master_overload () const

  flag which prevents overloading the master with a multiprocessor evaluation (forwarded to optionalInterface)

- void derived_init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set up optionalInterface and subModel for parallel operations

- void derived_init_serial ()

  set up optionalInterface and subModel for serial operations.

- void derived_set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set active parallel configuration within subModel

- void derived_free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  deallocate communicator partitions for the NestedModel (forwarded to optionalInterface and subModel)

- void serve ()

  Service optionalInterface and subModel job requests received from the master. Completes when a termination message is received from stop_servers().

- void stop_servers ()

  Executed by the master to terminate server operations for subModel and optionalInterface when iteration on the NestedModel is complete.

- const String & interface_id () const

  return the optionalInterface identifier

- int evaluation_id () const

  Return the current evaluation id for the NestedModel.

- void set_evaluation_reference ()

  set the evaluation counter reference points for the NestedModel (request forwarded to optionalInterface and subModel)

- void fine_grained_evaluation_counters ()

  request fine-grained evaluation reporting within optionalInterface and subModel

- void print_evaluation_summary (std::ostream &s, bool minimal_header=false, bool relative_count=true) const

  print the evaluation summary for the NestedModel (request forwarded to optionalInterface and subModel)

Private Member Functions

- void resolve_real_variable_mapping (const String &map1, const String &map2, size_t curr_index, short &inactive_sm_view)

for a named real mapping, resolve primary index and secondary target

- void resolve_integer_variable_mapping (const String &map1, const String &map2, size_t curr_index, short &inactive_sm_view)

  for a named integer mapping, resolve primary index and secondary target

- size_t sm_acv_index_map (size_t pacvm_index, short sacvm_target)

  offset pacvm_index based on sacvm_target to create mapped_index

- size_t sm_adiv_index_map (size_t padivm_index, short sadivm_target)

  offset padivm_index based on sadivm_target to create mapped_index

- size_t sm_adrv_index_map (size_t padrvm_index, short sadrvm_target)

  offset padrvm_index based on sadrvm_target to create mapped_index

- size_t cv_index_map (size_t cv_index)

  offset cv_index to create index into aggregated primary/secondary arrays

- size_t div_index_map (size_t div_index)

  offset div_index to create index into aggregated primary/secondary arrays

- size_t drv_index_map (size_t drv_index)

  offset drv_index to create index into aggregated primary/secondary arrays

- size_t ccv_index_map (size_t ccv_index)

  offset active complement ccv_index to create index into all continuous arrays

- size_t cdiv_index_map (size_t cdiv_index)

  offset active complement cdiv_index to create index into all discrete int arrays

- size_t cdrv_index_map (size_t cdrv_index)

  offset active complement cdrv_index to create index into all discrete real arrays

- void real_variable_mapping (const Real &r_var, size_t mapped_index, short svm_target)

  insert r_var into appropriate recipient

- void integer_variable_mapping (const int &i_var, size_t mapped_index, short svm_target)

  insert i_var into appropriate recipient

- void set_mapping (const ActiveSet &mapped_set, ActiveSet &interface_set, bool &opt_interface_map, ActiveSet &sub_iterator_set, bool &sub_iterator_map)

  define the evaluation requirements for the optionalInterface (interface_set) and the subIterator (sub_iterator_set) from the total model evaluation requirements (mapped_set)

- void response_mapping (const Response &interface_response, const Response &sub_iterator_response, Response &mapped_response)

combine the response from the optional interface evaluation with the response from the sub-iteration using the primaryCoeffs/secondaryCoeffs mappings to create the total response for the model

- void update_inactive_view (short new_view, short &view)

  update inactive variables view for subIterator based on new_view

- void update_inactive_view (unsigned short type, short &view)

  update inactive variables view for subIterator based on type

- void update_sub_model ()

  update subModel with current variable values/bounds/labels

Private Attributes

- int nestedModelEvalCntr

  number of calls to derived_compute_response()/ derived_asynch_compute_response()

- Iterator subIterator

  the sub-iterator that is executed on every evaluation of this model

- Model subModel

  the sub-model used in sub-iterator evaluations

- size_t numSubIterFns

  number of sub-iterator response functions prior to mapping

- size_t numSubIterMappedIneqCon

  number of top-level inequality constraints mapped from the sub-iteration results

- size_t numSubIterMappedEqCon

  number of top-level equality constraints mapped from the sub-iteration results

- Interface optionalInterface

  the optional interface contributes nonnested response data to the total model response

- String optInterfacePointer

  the optional interface pointer from the nested model specification

- Response optInterfaceResponse

  the response object resulting from optional interface evaluations

- size_t numOptInterfPrimary

  number of primary response functions (objective/least squares/generic functions) resulting from optional interface evaluations

- size_t numOptInterfIneqCon

  number of inequality constraints resulting from optional interface evaluations

- size_t numOptInterfEqCon

  number of equality constraints resulting from the optional interface evaluations

- SizetArray active1ACVarMapIndices

  "primary" variable mappings for inserting active continuous currentVariables within all continuous subModel variables. If there are no secondary mappings defined, then the insertions replace the subModel variable values.

- SizetArray active1ADIVarMapIndices

  "primary" variable mappings for inserting active discrete int currentVariables within all discrete int subModel variables. No secondary mappings are defined for discrete int variables, so the insertions replace the subModel variable values.

- SizetArray active1ADRVarMapIndices

  "primary" variable mappings for inserting active discrete real currentVariables within all discrete real subModel variables. No secondary mappings are defined for discrete real variables, so the insertions replace the subModel variable values.

- ShortArray active2ACVarMapTargets

  "secondary" variable mappings for inserting active continuous currentVariables into sub-parameters (e.g., distribution parameters for uncertain variables or bounds for continuous design/state variables) within all continuous subModel variables.

- ShortArray active2ADIVarMapTargets

  "secondary" variable mappings for inserting active discrete int currentVariables into sub-parameters (e.g., bounds for discrete design/state variables) within all discrete int subModel variables.

- ShortArray active2ADRVarMapTargets

  "secondary" variable mappings for inserting active discrete real currentVariables into sub-parameters (e.g., bounds for discrete design/state variables) within all discrete real subModel variables.

- SizetArray complement1ACVarMapIndices

  "primary" variable mappings for inserting the complement of the active continuous currentVariables within all continuous subModel variables

- SizetArray complement1ADIVarMapIndices

  "primary" variable mappings for inserting the complement of the active discrete int currentVariables within all discrete int subModel variables

- SizetArray complement1ADRVarMapIndices

  "primary" variable mappings for inserting the complement of the active discrete real currentVariables within all discrete real subModel variables

- BoolDeque extraCVarsData

  flags for updating subModel continuous bounds and labels, one for each active continuous variable in currentVariables

- BoolDeque extraDIVarsData

  flags for updating subModel discrete int bounds and labels, one for each active discrete int variable in currentVariables

- BoolDeque extraDRVarsData

  flags for updating subModel discrete real bounds and labels, one for each active discrete real variable in currentVariables

- RealMatrix primaryRespCoeffs

  "primary" response_mapping matrix applied to the sub-iterator response functions. For OUU, the matrix is applied to UQ statistics to create contributions to the top-level objective functions/least squares/ generic response terms.

- RealMatrix secondaryRespCoeffs

  "secondary" response_mapping matrix applied to the sub-iterator response functions. For OUU, the matrix is applied to UQ statistics to create contributions to the top-level inequality and equality constraints.

43.63.1 Detailed Description

Derived model class which performs a complete sub-iterator execution within every evaluation of the model. The NestedModel class nests a sub-iterator execution within every model evaluation. This capability is most commonly used for optimization under uncertainty, in which a nondeterministic iterator is executed on every optimization function evaluation. The NestedModel also contains an optional interface, for portions of the model evaluation which are independent from the sub-iterator, and a set of mappings for combining sub-iterator and optional interface data into a top level response for the model.

43.63.2 Member Function Documentation

43.63.2.1 void derived_compute_response (const ActiveSet & *set*) `[protected, virtual]`

portion of compute_response() specific to NestedModel Update subModel's inactive variables with active variables from currentVariables, compute the optional interface and sub-iterator responses, and map these to the total model response.

Reimplemented from Model.

References Response::active_set(), NestedModel::component_parallel_mode(), Model::currentResponse, Model::currentVariables, Interface::map(), NestedModel::nestedModelEvalCntr, NestedModel::optInterfaceResponse, NestedModel::optionalInterface, NestedModel::response_mapping(), Iterator::response_results(), Iterator::response_results_active_set(), Iterator::run_iterator(), NestedModel::set_mapping(), NestedModel::subIterator, and NestedModel::update_sub_model().

43.63.2.2 void derived_asynch_compute_response (const ActiveSet & *set*) `[protected, virtual]`

portion of asynch_compute_response() specific to NestedModel Not currently supported by NestedModels (need to add concurrent iterator support). As a result, derived_synchronize() and derived_synchronize_nowait() are inactive as well).

Reimplemented from Model.

References Dakota::abort_handler(), Response::active_set(), Model::currentResponse, Model::currentVariables, Interface::map(), NestedModel::nestedModelEvalCntr, NestedModel::optInterfaceResponse, NestedModel::optionalInterface, and NestedModel::set_mapping().

43.63.2.3 bool derived_master_overload () const [inline, protected, virtual]

flag which prevents overloading the master with a multiprocessor evaluation (forwarded to optionalInterface) Derived master overload for subModel is handled separately in subModel.compute_response() within subIterator.run().

Reimplemented from Model.

References Interface::iterator_eval_dedicated_master_flag(), Interface::multi_proc_eval_flag(), NestedModel::optInterfacePointer, and NestedModel::optionalInterface.

43.63.2.4 void derived_init_communicators (const int & *max_iterator_concurrency*, bool *recurse_flag* = true) [inline, protected, virtual]

set up optionalInterface and subModel for parallel operations Asynchronous flags need to be initialized for the subModel. In addition, max_iterator_concurrency is the outer level iterator concurrency, not the subIterator concurrency that subModel will see, and recomputing the message_lengths on the subModel is probably not a bad idea either. Therefore, recompute everything on subModel using init_communicators().

Reimplemented from Model.

References Model::init_communicators(), Interface::init_communicators(), Iterator::maximum_concurrency(), Model::messageLengths, NestedModel::optInterfacePointer, NestedModel::optionalInterface, NestedModel::subIterator, and NestedModel::subModel.

43.63.2.5 int evaluation_id () const [inline, protected, virtual]

Return the current evaluation id for the NestedModel. return the top level nested evaluation count. To get the lower level eval count, the subModel must be explicitly queried. This is consistent with the eval counter definitions in surrogate models.

Reimplemented from Model.

References NestedModel::nestedModelEvalCntr.

43.63.2.6 size_t cv_index_map (size_t *cv_index*) [private]

offset cv_index to create index into aggregated primary/secondary arrays maps index within active continuous variables to index within aggregated active continuous/discrete-int/discrete-real variables.

References Model::currentVariables, Variables::variables_components_totals(), and Variables::view().

Referenced by NestedModel::NestedModel(), and NestedModel::update_sub_model().

43.63.2.7 size_t div_index_map (size_t *div_index*) [private]

offset div_index to create index into aggregated primary/secondary arrays maps index within active discrete int variables to index within aggregated active continuous/discrete-int/discrete-real variables.

References Model::currentVariables, Variables::cv(), Variables::variables_components_totals(), and Variables::view().

Referenced by NestedModel::NestedModel(), and NestedModel::update_sub_model().

43.63.2.8 size_t drv_index_map (size_t *drv_index*) [private]

offset drv_index to create index into aggregated primary/secondary arrays maps index within active discrete real variables to index within aggregated active continuous/discrete-int/discrete-real variables.

References Model::currentVariables, Variables::cv(), Variables::div(), Variables::variables_components_totals(), and Variables::view().

Referenced by NestedModel::NestedModel(), and NestedModel::update_sub_model().

43.63.2.9 size_t ccv_index_map (size_t *ccv_index*) [private]

offset active complement ccv_index to create index into all continuous arrays maps index within complement of active continuous variables to index within all continuous variables.

References Dakota::abort_handler(), Model::currentVariables, Variables::variables_components_totals(), and Variables::view().

Referenced by NestedModel::NestedModel(), and NestedModel::update_sub_model().

43.63.2.10 size_t cdiv_index_map (size_t *cdiv_index*) [private]

offset active complement cdiv_index to create index into all discrete int arrays maps index within complement of active discrete int variables to index within all discrete int variables.

References Dakota::abort_handler(), Model::currentVariables, Variables::variables_components_totals(), and Variables::view().

Referenced by NestedModel::NestedModel(), and NestedModel::update_sub_model().

43.63.2.11 size_t cdrv_index_map (size_t *cdrv_index*) [private]

offset active complement cdrv_index to create index into all discrete real arrays maps index within complement of active discrete real variables to index within all discrete real variables.

References Dakota::abort_handler(), Model::currentVariables, Variables::variables_components_totals(), and Variables::view().

Referenced by NestedModel::NestedModel(), and NestedModel::update_sub_model().

43.63.2.12 void response_mapping (const Response & *opt_interface_response*, const Response & *sub_iterator_response*, Response & *mapped_response*) [private]

combine the response from the optional interface evaluation with the response from the sub-iteration using the primaryCoeffs/secondaryCoeffs mappings to create the total response for the model In the OUU case,

```
optionalInterface fns = {f}, {g} (deterministic primary functions, constraints)
subIterator fns       = {S}      (UQ response statistics)

Problem formulation for mapped functions:
                  minimize    {f} + [W]{S}
                  subject to  {g_l} <= {g}    <= {g_u}
                              {a_l} <= [A]{S} <= {a_u}
                              {g}    == {g_t}
                              [A]{S} == {a_t}
```

where [W] is the primary_mapping_matrix user input (primaryRespCoeffs class attribute), [A] is the secondary_-mapping_matrix user input (secondaryRespCoeffs class attribute), {{g_l},{a_l}} are the top level inequality constraint lower bounds, {{g_u},{a_u}} are the top level inequality constraint upper bounds, and {{g_t},{a_t}} are the top level equality constraint targets.

NOTE: optionalInterface/subIterator primary fns (obj/lsq/generic fns) overlap but optionalInterface/subIterator secondary fns (ineq/eq constraints) do not. The [W] matrix can be specified so as to allow

- some purely deterministic primary functions and some combined: [W] filled and [W].num_rows() < {f}.length() [combined first] *or* [W].num_rows() == {f}.length() and [W] contains rows of zeros [combined last]
- some combined and some purely stochastic primary functions: [W] filled and [W].num_rows() > {f}.length()
- separate deterministic and stochastic primary functions: [W].num_rows() > {f}.length() and [W] contains {f}.length() rows of zeros.

If the need arises, could change constraint definition to allow overlap as well: {g_l} <= {g} + [A]{S} <= {g_u} with [A] usage the same as for [W] above.

In the UOO case, things are simpler, just compute statistics of each optimization response function: [W] = [I], {f}/{g}/[A] are empty.

References Dakota::abort_handler(), Response::active_set_derivative_vector(), Response::active_set_-request_vector(), Dakota::copy_data(), Response::function_gradient(), Response::function_gradient_-view(), Response::function_gradients(), Response::function_hessian(), Response::function_hessian_-view(), Response::function_hessians(), Response::function_value(), Response::function_values(), Response::function_values_view(), Response::num_functions(), NestedModel::numOptInterfEqCon, NestedModel::numOptInterfIneqCon, NestedModel::numOptInterfPrimary, NestedModel::numSubIterFns, NestedModel::numSubIterMappedEqCon, NestedModel::numSubIterMappedIneqCon, NestedModel::optInterfacePointer, NestedModel::primaryRespCoeffs, Response::reset_inactive(), and NestedModel::secondaryRespCoeffs.

Referenced by NestedModel::derived_compute_response().

43.63.3 Member Data Documentation

43.63.3.1 Model subModel [private]

the sub-model used in sub-iterator evaluations There are no restrictions on subModel, so arbitrary nestings are possible. This is commonly used to support surrogate-based optimization under uncertainty by having NestedModels contain SurrogateModels and vice versa.

Referenced by NestedModel::component_parallel_mode(), NestedModel::derived_free_communicators(), NestedModel::derived_init_communicators(), NestedModel::derived_init_serial(), NestedModel::derived_set_communicators(), NestedModel::derived_subordinate_models(), NestedModel::fine_grained_evaluation_counters(), NestedModel::integer_variable_mapping(), NestedModel::NestedModel(), NestedModel::print_evaluation_summary(), NestedModel::real_variable_mapping(), NestedModel::resolve_integer_variable_mapping(), NestedModel::resolve_real_variable_mapping(), NestedModel::serve(), NestedModel::set_mapping(), NestedModel::sm_acv_index_map(), NestedModel::sm_adiv_index_map(), NestedModel::subordinate_model(), NestedModel::surrogate_response_mode(), NestedModel::update_inactive_view(), and NestedModel::update_sub_model().

The documentation for this class was generated from the following files:

- NestedModel.H
- NestedModel.C

43.64 NIDRProblemDescDB Class Reference

The derived input file database utilizing the new IDR parser. Inheritance diagram for NIDRProblemDescDB::

Public Member Functions

- NIDRProblemDescDB (ParallelLibrary &parallel_lib)

  constructor

- ~NIDRProblemDescDB ()

  destructor

- void derived_parse_inputs (const char *dakota_input_file, const char *parser_options)

  parses the input file and populates the problem description database using NIDR.

- void derived_broadcast ()

  perform any data processing that must be coordinated with DB buffer broadcasting (performed prior to broadcasting the DB buffer on rank 0 and after receiving the DB buffer on other processor ranks)

- void derived_post_process ()

  perform any additional data post-processing

- **KWH** (iface_Rlit)
- **KWH** (iface_false)
- **KWH** (iface_ilit)
- **KWH** (iface_pint)
- **KWH** (iface_lit)
- **KWH** (iface_start)
- **KWH** (iface_stop)
- **KWH** (iface_str)
- **KWH** (iface_str2D)
- **KWH** (iface_strL)
- **KWH** (iface_true)
- **KWH** (method_Ii)
- **KWH** (method_Real)
- **KWH** (method_Real01)
- **KWH** (method_RealDL)
- **KWH** (method_RealLlit)
- **KWH** (method_Realp)

- **KWH** (method_Realz)
- **KWH** (method_Ri)
- **KWH** (method_coliny_ea)
- **KWH** (method_false)
- **KWH** (method_ilit2)
- **KWH** (method_ilit2p)
- **KWH** (method_int)
- **KWH** (method_intDL)
- **KWH** (method_lit)
- **KWH** (method_lit2)
- **KWH** (method_litc)
- **KWH** (method_liti)
- **KWH** (method_litp)
- **KWH** (method_litpp)
- **KWH** (method_litpp_final)
- **KWH** (method_litr)
- **KWH** (method_litz)
- **KWH** (method_meritFn)
- **KWH** (method_moga_begin)
- **KWH** (method_moga_final)
- **KWH** (method_nnint)
- **KWH** (method_nnintz)
- **KWH** (method_num_resplevs)
- **KWH** (method_piecewise)
- **KWH** (method_pint)
- **KWH** (method_pintz)
- **KWH** (method_resplevs)
- **KWH** (method_resplevs01)
- **KWH** (method_shint)
- **KWH** (method_slit2)
- **KWH** (method_soga_begin)
- **KWH** (method_soga_final)
- **KWH** (method_start)
- **KWH** (method_stop)
- **KWH** (method_str)
- **KWH** (method_strL)
- **KWH** (method_true)
- **KWH** (method_tr_final)
- **KWH** (method_type)
- **KWH** (method_ushint)
- **KWH** (method_ushintL)
- **KWH** (model_Real)
- **KWH** (model_RealDL)
- **KWH** (model_false)
- **KWH** (model_int)
- **KWH** (model_intset)

- **KWH** (model_lit)
- **KWH** (model_order)
- **KWH** (model_shint)
- **KWH** (model_start)
- **KWH** (model_stop)
- **KWH** (model_str)
- **KWH** (model_strL)
- **KWH** (model_true)
- **KWH** (model_type)
- **KWH** (resp_RealDL)
- **KWH** (resp_RealL)
- **KWH** (resp_false)
- **KWH** (resp_intL)
- **KWH** (resp_lit)
- **KWH** (resp_nnintz)
- **KWH** (resp_start)
- **KWH** (resp_stop)
- **KWH** (resp_str)
- **KWH** (resp_strL)
- **KWH** (resp_true)
- **KWH** (strategy_Real)
- **KWH** (strategy_RealL)
- **KWH** (strategy_int)
- **KWH** (strategy_lit)
- **KWH** (strategy_start)
- **KWH** (strategy_str)
- **KWH** (strategy_strL)
- **KWH** (strategy_true)
- **KWH** (var_RealLb)
- **KWH** (var_RealLd)
- **KWH** (var_RealUb)
- **KWH** (var_caulbl)
- **KWH** (var_ceulbl)
- **KWH** (var_dailbl)
- **KWH** (var_darlbl)
- **KWH** (var_intDL)
- **KWH** (var_intL)
- **KWH** (var_intz)
- **KWH** (var_start)
- **KWH** (var_stop)
- **KWH** (var_str)
- **KWH** (var_strL)
- **KWH** (var_true)
- **KWH** (var_vil)
- **KWH** (var_vrl)

Static Public Member Functions

- static void **Var_boundchk** (DataVariablesRep ∗)
- static void **Var_boundgen** (DataVariablesRep ∗)
- static void **Var_iboundchk** (DataVariablesRep ∗)
- static void **Var_iboundgen** (DataVariablesRep ∗)
- static void **botch** (const char ∗fmt,...)
- static void **check_variables** (std::list< DataVariables > ∗)
- static void **check_responses** (std::list< DataResponses > ∗)
- static void **make_variable_defaults** (std::list< DataVariables > ∗)
- static void **make_response_defaults** (std::list< DataResponses > ∗)
- static void **squawk** (const char ∗fmt,...)
- static void **warn** (const char ∗fmt,...)

Static Public Attributes

- static NIDRProblemDescDB ∗ pDDBInstance

  pointer to the active object instance used within the static kwhandler functions in order to avoid the need for static data

- static int **nerr** = 0

Static Private Member Functions

- static void **var_stop1** (void ∗)

Private Attributes

- std::list< void ∗ > **VIL**

43.64.1 Detailed Description

The derived input file database utilizing the new IDR parser. The NIDRProblemDescDB class is derived from ProblemDescDB for use by the NIDR parser in processing DAKOTA input file data. For information on modifying the NIDR input parsing procedures, refer to Dakota/docs/Dev_Spec_Change.dox. For more on the parsing technology, see "Specifying and Reading Program Input with NIDR" by David M. Gay (report SAND2008-2261P, which is available in PDF form as `http://www.sandia.gov/~dmgay/nidr08.pdf`). Source for the routines declared herein is NIDRProblemDescDB.C, in which most routines are so short that a description seems unnecessary.

43.64.2 Member Function Documentation

43.64.2.1 void derived_parse_inputs (const char * *dakota_input_file*, const char * *parser_options*) [virtual]

parses the input file and populates the problem description database using NIDR. Parse the input file using the Input Deck Reader (IDR) parsing system. IDR populates the IDRProblemDescDB object with the input file data.

Reimplemented from ProblemDescDB.

References Dakota::abort_handler(), ProblemDescDB::dataInterfaceList, ProblemDescDB::dataMethodList, DataMethodRep::dlDetails, DataMethodRep::dlLib, ProblemDescDB::parallel_library(), and NIDRProblemDescDB::pDDBInstance.

The documentation for this class was generated from the following files:

- NIDRProblemDescDB.H
- NIDRProblemDescDB.C

43.65 NL2Res Struct Reference

Auxiliary information passed to calcr and calcj via ur.

Public Attributes

- Real * r

  residual r = r(x)

- Real * J

  Jacobian J = J(x).

- Real * x

  corresponding parameter vector

- int nf

  function invocation count for r(x)

43.65.1 Detailed Description

Auxiliary information passed to calcr and calcj via ur.

The documentation for this struct was generated from the following file:

- NL2SOLLeastSq.C

43.66 NL2SOLLeastSq Class Reference

Wrapper class for the NL2SOL nonlinear least squares library. Inheritance diagram for NL2SOLLeastSq::

Public Member Functions

- NL2SOLLeastSq (Model &model)

  standard constructor

- NL2SOLLeastSq (NoDBBaseConstructor, Model &model)

  alternate constructor

- ~NL2SOLLeastSq ()

  destructor

- void minimize_residuals ()

  Used within the least squares branch for minimizing the sum of squares residuals. Redefines the run virtual function for the least squares branch.

Static Private Member Functions

- static void calcr (int *np, int *pp, Real *x, int *nfp, Real *r, int *ui, void *ur, Vf vf)

  evaluator function for residual vector

- static void calcj (int *np, int *pp, Real *x, int *nfp, Real *J, int *ui, void *ur, Vf vf)

  evaluator function for residual Jacobian

Private Attributes

- int auxprt

  auxilary printing bits (see Dakota Ref Manual): sum of < 1 = x0prt (print initial guess) < 2 = solprt (print final solution) < 4 = statpr (print solution statistics) < 8 = parprt (print nondefault parameters) < 16 = dradpr (print bound constraint drops/adds) < debug/verbose/normal use default = 31 (everything), < quiet uses 3, silent uses 0.

- int outlev

  frequency of output summary lines in number of iterations < (debug/verbose/normal/quiet use default = 1, silent uses 0)

- Real dltfdj

  finite-diff step size for computing Jacobian approximation < (`fd_gradient_step_size`)

- Real delta0

  finite-diff step size for gradient differences for H < (a component of some covariance approximations, if desired) < (`fd_hessian_step_size`)

- Real dltfdc

  finite-diff step size for function differences for H < (`fd_hessian_step_size`)

- int mxfcal

  function-evaluation limit (`max_function_evaluations`)

- int mxiter

  iteration limit (`max_iterations`)

- Real rfctol

  relative fn convergence tolerance (`convergence_tolerance`)

- Real afctol

  absolute fn convergence tolerance (`absolute_conv_tol`)

- Real xctol

  x-convergence tolerance (`x_conv_tol`)

- Real sctol

  singular convergence tolerance (`singular_conv_tol`)

- Real lmaxs

  radius for singular-convergence test (`singular_radius`)

- Real xftol

  false-convergence tolerance (`false_conv_tol`)

- int covreq

  kind of covariance required (`covariance`): < 1 or -1 ==> sigma^2 H^-1 J^T J H^-1 < 2 or -2 ==> sigma^2 H^-1 < 3 or -3 ==> sigma^2 (J^T J)^-1 < 1 or 2 ==> use gradient diffs to estimate H < -1 or -2 ==> use function diffs to estimate H < default = 0 (no covariance)

- int rdreq

  whether to compute the regression diagnostic vector < (`regression_diagnostics`)

- Real fprec

  expected response function precision (`function_precision`)

- Real lmax0

  initial trust-region radius (`initial_trust_radius`)

Static Private Attributes

- static NL2SOLLeastSq * nl2solInstance

  pointer to the active object instance used within the static evaluator functions

43.66.1 Detailed Description

Wrapper class for the NL2SOL nonlinear least squares library. The NL2SOLLeastSq class provides a wrapper for NL2SOL (TOMS Algorithm 573), in the updated form of Port Library routines dn[fg][b] from Bell Labs; see http://www.netlib.org/port/readme. The Fortran from Port has been turned into C by f2c. NL2SOL uses a function pointer approach for which passed functions must be either global functions or static member functions.

43.66.2 Member Function Documentation

43.66.2.1 void minimize_residuals () [virtual]

Used within the least squares branch for minimizing the sum of squares residuals. Redefines the run virtual function for the least squares branch.

Details on the following subscript values appear in "Usage Summary for Selected Optimization Routines" by David M. Gay, Computing Science Technical Report No. 153, AT&T Bell Laboratories, 1990. http://netlib.bell-labs.com/cm/cs/cstr/153.ps.gz

Implements LeastSq.

References NL2SOLLeastSq::afctol, NL2SOLLeastSq::auxprt, Iterator::bestResponseArray, Iterator::bestVariablesArray, Minimizer::boundConstraintFlag, NL2SOLLeastSq::calcj(), NL2SOLLeastSq::calcr(), Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::continuous_variables(), Dakota::copy_data(), NL2SOLLeastSq::covreq, NL2SOLLeastSq::delta0, NL2SOLLeastSq::dltfdc, NL2SOLLeastSq::dltfdj, NL2SOLLeastSq::fprec, LeastSq::get_confidence_intervals(), Iterator::gradientType, Iterator::iteratedModel, NL2SOLLeastSq::lmax0, NL2SOLLeastSq::lmaxs, NL2SOLLeastSq::mxfcal, NL2SOLLeastSq::mxiter, NL2SOLLeastSq::nl2solInstance, Iterator::numContinuousVars, LeastSq::numLeastSqTerms, NL2SOLLeastSq::outlev, NL2SOLLeastSq::rdreq, NL2SOLLeastSq::rfctol, NL2SOLLeastSq::sctol, Minimizer::speculativeFlag, Minimizer::vendorNumericalGradFlag, NL2SOLLeastSq::xctol, and NL2SOLLeastSq::xftol.

The documentation for this class was generated from the following files:

- NL2SOLLeastSq.H
- NL2SOLLeastSq.C

43.67 NLPQLPOptimizer Class Reference

Wrapper class for the NLPQLP optimization library, Version 2.0. Inheritance diagram for NLPQLPOptimizer::

Public Member Functions

- NLPQLPOptimizer (Model &model)

  standard constructor

- NLPQLPOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor

- ~NLPQLPOptimizer ()

  destructor

- void find_optimum ()

  Used within the optimizer branch for computing the optimal solution. Redefines the run virtual function for the optimizer branch.

Protected Member Functions

- void initialize_run ()

  performs run-time set up

Private Member Functions

- void initialize ()

  Shared constructor code.

- void allocate_workspace ()

  Allocates workspace for the optimizer.

- void deallocate_workspace ()

  Releases workspace memory.

- void allocate_constraints ()

  Allocates constraint mappings.

Private Attributes

- int L

  L : Number of parallel systems, i.e. function calls during line search at predetermined iterates. HINT: If only less than 10 parallel function evaluations are possible, it is recommended to apply the serial version by setting L=1.

- int numEqConstraints

  numEqConstraints : Number of equality constraints.

- int MMAX

  MMAX : Row dimension of array DG containing Jacobian of constraints. MMAX must be at least one and greater or equal to M.

- int N

  N : Number of optimization variables.

- int NMAX

  NMAX : Row dimension of C. NMAX must be at least two and greater than N.

- int MNN2

  MNN2 : Must be equal to M+N+N+2.

- double ∗ X

  X(NMAX,L) : Initially, the first column of X has to contain starting values for the optimal solution. On return, X is replaced by the current iterate. In the driving program the row dimension of X has to be equal to NMAX. X is used internally to store L different arguments for which function values should be computed simultaneously.

- double ∗ F

  F(L) : On return, F(1) contains the final objective function value. F is used also to store L different objective function values to be computed from L iterates stored in X.

- double ∗ G

  G(MMAX,L) : On return, the first column of G contains the constraint function values at the final iterate X. In the driving program the row dimension of G has to be equal to MMAX. G is used internally to store L different set of constraint function values to be computed from L iterates stored in X.

- double ∗ DF

  DF(NMAX) : DF contains the current gradient of the objective function. In case of numerical differentiation and a distributed system (L>1), it is recommended to apply parallel evaluations of F to compute DF.

- double * DG

  DG(MMAX,NMAX) : DG contains the gradients of the active constraints (ACTIVE(J)=.true.) at a current iterate X. The remaining rows are filled with previously computed gradients. In the driving program the row dimension of DG has to be equal to MMAX.

- double * U

  U(MNN2) : U contains the multipliers with respect to the actual iterate stored in the first column of X. The first M locations contain the multipliers of the M nonlinear constraints, the subsequent N locations the multipliers of the lower bounds, and the final N locations the multipliers of the upper bounds. At an optimal solution, all multipliers with respect to inequality constraints should be nonnegative.

- double * C

  C(NMAX,NMAX) : On return, C contains the last computed approximation of the Hessian matrix of the Lagrangian function stored in form of an LDL decomposition. C contains the lower triangular factor of an LDL factorization of the final quasi-Newton matrix (without diagonal elements, which are always one). In the driving program, the row dimension of C has to be equal to NMAX.

- double * D

  D(NMAX) : The elements of the diagonal matrix of the LDL decomposition of the quasi-Newton matrix are stored in the one-dimensional array D.

- double ACC

  ACC : The user has to specify the desired final accuracy (e.g. 1.0D-7). The termination accuracy should not be smaller than the accuracy by which gradients are computed.

- double ACCQP

  ACCQP : The tolerance is needed for the QP solver to perform several tests, for example whether optimality conditions are satisfied or whether a number is considered as zero or not. If ACCQP is less or equal to zero, then the machine precision is computed by NLPQLP and subsequently multiplied by 1.0D+4.

- double STPMIN

  *STPMIN : Minimum steplength in case of L>1. Recommended is any value in the order of the accuracy by which functions are computed. The value is needed to compute a steplength reduction factor by STPMIN**(1/L-1). If STPMIN<=0, then STPMIN=ACC is used.*

- int MAXFUN

  MAXFUN : The integer variable defines an upper bound for the number of function calls during the line search (e.g. 20). MAXFUN is only needed in case of L=1, and must not be greater than 50.

- int MAXIT

  MAXIT : Maximum number of outer iterations, where one iteration corresponds to one formulation and solution of the quadratic programming subproblem, or, alternatively, one evaluation of gradients (e.g. 100).

- int MAX_NM

  MAX_NM : Stack size for storing merit function values at previous iterations for non-monotone line search (e.g. 10). In case of MAX_NM=0, monotone line search is performed.

- double TOL_NM

TOL_NM : Relative bound for increase of merit function value, if line search is not successful during the very first step. Must be non-negative (e.g. 0.1).

- int IPRINT

  IPRINT : Specification of the desired output level. IPRINT = 0 : No output of the program. IPRINT = 1 : Only a final convergence analysis is given. IPRINT = 2 : One line of intermediate results is printed in each iteration. IPRINT = 3 : More detailed information is printed in each iteration step, e.g. variable, constraint and multiplier values. IPRINT = 4 : In addition to 'IPRINT=3', merit function and steplength values are displayed during the line search.

- int MODE

  MODE : The parameter specifies the desired version of NLPQLP. MODE = 0 : Normal execution (reverse communication!). MODE = 1 : The user wants to provide an initial guess for the multipliers in U and for the Hessian of the Lagrangian function in C and D in form of an LDL decomposition.

- int IOUT

  IOUT : Integer indicating the desired output unit number, i.e. all write-statements start with 'WRITE(IOUT,... '.

- int IFAIL

  IFAIL : The parameter shows the reason for terminating a solution process. Initially IFAIL must be set to zero. On return IFAIL could contain the following values: IFAIL =-2 : Compute gradient values w.r.t. the variables stored in first column of X, and store them in DF and DG. Only derivatives for active constraints ACTIVE(J)=.TRUE. need to be computed. Then call NLPQLP again, see below. IFAIL =-1 : Compute objective fn and all constraint values subject the variables found in the first L columns of X, and store them in F and G. Then call NLPQLP again, see below. IFAIL = 0 : The optimality conditions are satisfied. IFAIL = 1 : The algorithm has been stopped after MAXIT iterations. IFAIL = 2 : The algorithm computed an uphill search direction. IFAIL = 3 : Underflow occurred when determining a new approxi- mation matrix for the Hessian of the Lagrangian. IFAIL = 4 : The line search could not be terminated successfully. IFAIL = 5 : Length of a working array is too short. More detailed error information is obtained with 'IPRINT>0'. IFAIL = 6 : There are false dimensions, for example M>MMAX, N>=NMAX, or MNN2<>M+N+N+2. IFAIL = 7 : The search direction is close to zero, but the current iterate is still infeasible. IFAIL = 8 : The starting point violates a lower or upper bound. IFAIL = 9 : Wrong input parameter, i.e., MODE, LDL decomposition in D and C (in case of MODE=1), IPRINT, IOUT IFAIL = 10 : Internal inconsistency of the quadratic subproblem, division by zero. IFAIL > 100 : The solution of the quadratic programming subproblem has been terminated with an error message and IFAIL is set to IFQL+100, where IFQL denotes the index of an inconsistent constraint.

- double * WA

  WA(LWA) : WA is a real working array of length LWA.

- int LWA

  LWA : LWA value extracted from NLPQLP20.f.

- int * KWA

  KWA(LKWA) : The user has to provide working space for an integer array.

- int LKWA

  LKWA : LKWA should be at least N+10.

- int * ACTIVE

ACTIVE(LACTIV) : The logical array shows a user the constraints, which NLPQLP considers to be active at the last computed iterate, i.e. G(J,X) is active, if and only if ACTIVE(J)=.TRUE., J=1,...,M.

- int LACTIVE

  LACTIV : The length LACTIV of the logical array should be at least 2∗M+10.

- int LQL

  LQL : If LQL = .TRUE., the quadratic programming subproblem is to be solved with a full positive definite quasi-Newton matrix. Otherwise, a Cholesky decomposition is performed and updated, so that the subproblem matrix contains only an upper triangular factor.

- int numNlpqlConstr

  total number of constraints seen by NLPQL

- SizetList nonlinIneqConMappingIndices

  a list of indices for referencing the DAKOTA nonlinear inequality constraints used in computing the corresponding NLPQL constraints.

- RealList nonlinIneqConMappingMultipliers

  a list of multipliers for mapping the DAKOTA nonlinear inequality constraints to the corresponding NLPQL constraints.

- RealList nonlinIneqConMappingOffsets

  a list of offsets for mapping the DAKOTA nonlinear inequality constraints to the corresponding NLPQL constraints.

- SizetList linIneqConMappingIndices

  a list of indices for referencing the DAKOTA linear inequality constraints used in computing the corresponding NLPQL constraints.

- RealList linIneqConMappingMultipliers

  a list of multipliers for mapping the DAKOTA linear inequality constraints to the corresponding NLPQL constraints.

- RealList linIneqConMappingOffsets

  a list of offsets for mapping the DAKOTA linear inequality constraints to the corresponding NLPQL constraints.

43.67.1 Detailed Description

Wrapper class for the NLPQLP optimization library, Version 2.0. **********

AN IMPLEMENTATION OF A SEQUENTIAL QUADRATIC PROGRAMMING METHOD FOR SOLVING NONLINEAR OPTIMIZATION PROBLEMS BY DISTRIBUTED COMPUTING AND NON-MONOTONE LINE SEARCH

This subroutine solves the general nonlinear programming problem

minimize F(X) subject to G(J,X) = 0 , J=1,...,ME G(J,X) >= 0 , J=ME+1,...,M XL <= X <= XU

and is an extension of the code NLPQLD. NLPQLP is specifically tuned to run under distributed systems. A new input parameter L is introduced for the number of parallel computers, that is the number of function calls to be

executed simultaneously. In case of L=1, NLPQLP is identical to NLPQLD. Otherwise the line search is modified to allow L parallel function calls in advance. Moreover the user has the opportunity to used distributed function calls for evaluating gradients.

The algorithm is a modification of the method of Wilson, Han, and Powell. In each iteration step, a linearly constrained quadratic programming problem is formulated by approximating the Lagrangian function quadratically and by linearizing the constraints. Subsequently, a one-dimensional line search is performed with respect to an augmented Lagrangian merit function to obtain a new iterate. Also the modified line search algorithm guarantees convergence under the same assumptions as before.

For the new version, a non-monotone line search is implemented which allows to increase the merit function in case of instabilities, for example caused by round-off errors, errors in gradient approximations, etc.

The subroutine contains the option to predetermine initial guesses for the multipliers or the Hessian of the Lagrangian function and is called by reverse communication.

The documentation for this class was generated from the following files:

- NLPQLPOptimizer.H
- NLPQLPOptimizer.C

43.68 NLSSOLLeastSq Class Reference

Wrapper class for the NLSSOL nonlinear least squares library. Inheritance diagram for NLSSOLLeastSq::

Public Member Functions

- NLSSOLLeastSq (Model &model)

  standard constructor

- NLSSOLLeastSq (NoDBBaseConstructor, Model &model)

  alternate constructor

- ~NLSSOLLeastSq ()

  destructor

- void minimize_residuals ()

  Used within the least squares branch for minimizing the sum of squares residuals. Redefines the run virtual function for the least squares branch.

Static Private Member Functions

- static void least_sq_eval (int &mode, int &m, int &n, int &nrowfj, double *x, double *f, double *gradf, int &nstate)

  Evaluator for NLSSOL: computes the values and first derivatives of the least squares terms (passed by function pointer to NLSSOL).

Static Private Attributes

- static NLSSOLLeastSq * nlssolInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.68.1 Detailed Description

Wrapper class for the NLSSOL nonlinear least squares library. The NLSSOLLeastSq class provides a wrapper for NLSSOL, a Fortran 77 sequential quadratic programming library from Stanford University marketed by Stanford Business Associates. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any nonstatic attribute used within static member functions must be either local to that function or accessed through a static pointer.

The user input mappings are as follows: `max_function_evaluations` is implemented directly in NLSSOLLeastSq's evaluator functions since there is no NLSSOL parameter equivalent, and `max_iterations`, `convergence_tolerance`, `output` verbosity, `verify_level`, `function_precision`, and `linesearch_tolerance` are mapped into NLSSOL's "Major Iteration Limit", "Optimality Tolerance", "Major Print Level" (`verbose`: Major Print Level = 20; `quiet`: Major Print Level = 10), "Verify Level", "Function Precision", and "Linesearch Tolerance" parameters, respectively, using NLSSOL's npoptn() subroutine (as wrapped by npoptn2() from the npoptn_wrapper.f file). Refer to [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] for information on NLSSOL's optional input parameters and the npoptn() subroutine.

43.68.2 Constructor & Destructor Documentation

43.68.2.1 NLSSOLLeastSq (Model & *model*)

standard constructor This is the primary constructor. It accepts a Model reference.

References Minimizer::constraintTol, Iterator::convergenceTol, Iterator::fdGradStepSize, ProblemDescDB::get_int(), ProblemDescDB::get_real(), Iterator::gradientType, Iterator::maxIterations, Iterator::outputLevel, Iterator::probDescDB, SOLBase::set_options(), Minimizer::speculativeFlag, and Minimizer::vendorNumericalGradFlag.

43.68.2.2 NLSSOLLeastSq (NoDBBaseConstructor, Model & *model*)

alternate constructor This is an alternate constructor which accepts a Model but does not have a supporting method specification from the ProblemDescDB.

References Minimizer::constraintTol, Iterator::convergenceTol, Iterator::fdGradStepSize, Iterator::gradientType, Iterator::maxIterations, Iterator::outputLevel, SOLBase::set_options(), Minimizer::speculativeFlag, and Minimizer::vendorNumericalGradFlag.

The documentation for this class was generated from the following files:

- NLSSOLLeastSq.H
- NLSSOLLeastSq.C

43.69 NoDBBaseConstructor Struct Reference

Dummy struct for overloading constructors used in on-the-fly instantiations.

Public Member Functions

- NoDBBaseConstructor (int=0)

  C++ structs can have constructors.

43.69.1 Detailed Description

Dummy struct for overloading constructors used in on-the-fly instantiations. NoDBBaseConstructor is used to overload the constructor used for on-the-fly instantiations in which ProblemDescDB queries cannot be used. Putting this struct here avoids circular dependencies.

The documentation for this struct was generated from the following file:

- global_defs.h

43.70 NonD Class Reference

Base class for all nondetermistic iterators (the DAKOTA/UQ branch). Inheritance diagram for NonD::

Public Member Functions

- void initialize_random_variables (short u_space_type)

  initialize natafTransform based on distribution data from iteratedModel

- void initialize_random_variables (const Pecos::ProbabilityTransformation &transform)

  alternate form: initialize natafTransform based on incoming data

- void requested_levels (const RealVectorArray &req_resp_levels, const RealVectorArray &req_prob_levels, const RealVectorArray &req_rel_levels, const RealVectorArray &req_gen_rel_levels, short resp_lev_-target, bool cdf_flag)

  set requestedRespLevels, requestedProbLevels, requestedRelLevels, requestedGenRelLevels, respLevelTarget, and cdfFlag (used in combination with alternate ctors)

- void distribution_parameter_derivatives (bool dist_param_derivs)

  set distParamDerivs

- bool pdf_output () const

  get pdfOutput

- void pdf_output (bool output)

  set pdfOutput

- Pecos::ProbabilityTransformation & variable_transformation ()

  return natafTransform

Protected Member Functions

- NonD (Model &model)

  constructor

- NonD (NoDBBaseConstructor, Model &model)

  alternate constructor for sample generation and evaluation "on the fly"

- NonD (NoDBBaseConstructor, const RealVector &lower_bnds, const RealVector &upper_bnds)

  alternate constructor for sample generation "on the fly"

- ~NonD ()

  destructor

- void initialize_run ()

  utility function to perform common operations prior to pre_run(); typically memory initialization; setting of instance pointers

- void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- void finalize_run ()

  utility function to perform common operations following post_run(); deallocation and resetting of instance pointers

- const Response & response_results () const

  return the final statistics from the nondeterministic iteration

- void response_results_active_set (const ActiveSet &set)

  set the active set within finalStatistics

- virtual void quantify_uncertainty ()=0

  performs a forward uncertainty propagation of parameter distributions into response statistics

- virtual void initialize_final_statistics ()

  initializes finalStatistics for storing NonD final results

- void initialize_random_variable_transformation ()

  instantiate natafTransform

- void initialize_random_variable_types (short u_space_type)

  initializes ranVarTypesX and ranVarTypesU within natafTransform

- void initialize_random_variable_parameters ()

  initializes ranVarMeansX, ranVarStdDevsX, ranVarLowerBndsX, ranVarUpperBndsX, and ranVarAddtlParamsX within natafTransform

- void initialize_random_variable_correlations ()

  propagate iteratedModel correlations to natafTransform

- void verify_correlation_support ()

  verify that correlation warping supported by Der Kiureghian & Liu for given variable types

- void initialize_final_statistics_gradients ()

initializes finalStatistics::functionGradients

- void initialize_distribution_mappings ()

  size computed{Resp,Prob,Rel,GenRel}Levels

- void print_distribution_mappings (std::ostream &s) const

  prints the z/p/beta/beta∗ mappings reflected in {requested,computed}{Resp,Prob,Rel,GenRel}Levels

- void construct_u_space_model (Model &x_model, Model &u_model, bool global_bounds=false, Real bound=10.)

  recast x_model from x-space to u-space to create u_model

- void construct_lhs (Iterator &u_space_sampler, Model &u_model, int num_samples, int seed, const String &rng)

  assign a NonDLHSSampling instance within u_space_sampler

Static Protected Member Functions

- static void vars_u_to_x_mapping (const Variables &u_vars, Variables &x_vars)

  static function for RecastModels used to map u-space variables from NonD Iterators to x-space variables for Model evaluations.

- static void set_u_to_x_mapping (const Variables &u_vars, const ActiveSet &u_set, ActiveSet &x_set)

  static function for RecastModels used to map u-space ActiveSets from NonD Iterators to x-space ActiveSets for Model evaluations.

- static void resp_x_to_u_mapping (const Variables &x_vars, const Variables &u_vars, const Response &x_-response, Response &u_response)

  static function for RecastModels used to map x-space responses from Model evaluations to u-space responses for return to NonD Iterators.

Protected Attributes

- NonD ∗ prevNondInstance

  pointer containing previous value of nondInstance

- Pecos::ProbabilityTransformation natafTransform

  Nonlinear variable transformation that encapsulates the required data for performing transformations from X -> Z -> U and back.

- size_t numContDesVars

  number of continuous design variables (modeled using uniform distribution for All view modes)

- size_t numDiscIntDesVars

number of discrete integer design variables (modeled using discrete histogram distributions for All view modes)

- size_t numDiscRealDesVars

  number of discrete real design variables (modeled using discrete histogram distributions for All view modes)

- size_t numDesignVars

  total number of design variables

- size_t numContStateVars

  number of continuous state variables (modeled using uniform distribution for All view modes)

- size_t numDiscIntStateVars

  number of discrete integer state variables (modeled using discrete histogram distributions for All view modes)

- size_t numDiscRealStateVars

  number of discrete real state variables (modeled using discrete histogram distributions for All view modes)

- size_t numStateVars

  total number of state variables

- size_t numNormalVars

  number of normal uncertain variables (native space)

- size_t numLognormalVars

  number of lognormal uncertain variables (native space)

- size_t numUniformVars

  number of uniform uncertain variables (native space)

- size_t numLoguniformVars

  number of loguniform uncertain variables (native space)

- size_t numTriangularVars

  number of triangular uncertain variables (native space)

- size_t numExponentialVars

  number of exponential uncertain variables (native space)

- size_t numBetaVars

  number of beta uncertain variables (native space)

- size_t numGammaVars

  number of gamma uncertain variables (native space)

- size_t numGumbelVars

  number of gumbel uncertain variables (native space)

- size_t numFrechetVars

  number of frechet uncertain variables (native space)

- size_t numWeibullVars

  number of weibull uncertain variables (native space)

- size_t numHistogramBinVars

  number of histogram bin uncertain variables (native space)

- size_t numPoissonVars

  number of Poisson uncertain variables (native space)

- size_t numBinomialVars

  number of binomial uncertain variables (native space)

- size_t numNegBinomialVars

  number of negative binomial uncertain variables (native space)

- size_t numGeometricVars

  number of geometric uncertain variables (native space)

- size_t numHyperGeomVars

  number of hypergeometric uncertain variables (native space)

- size_t numHistogramPtVars

  number of histogram point uncertain variables (native space)

- size_t numIntervalVars

  number of interval uncertain variables (native space)

- size_t numContAleatUncVars

  total number of aleatory uncertain variables (native space)

- size_t numDiscIntAleatUncVars

  total number of aleatory uncertain variables (native space)

- size_t numDiscRealAleatUncVars

  total number of aleatory uncertain variables (native space)

- size_t numAleatoryUncVars

  total number of aleatory uncertain variables (native space)

- size_t numContEpistUncVars

  total number of epistemic uncertain variables (native space)

- size_t numDiscIntEpistUncVars

  total number of epistemic uncertain variables (native space)

- size_t numDiscRealEpistUncVars

  total number of epistemic uncertain variables (native space)

- size_t numEpistemicUncVars

  total number of epistemic uncertain variables (native space)

- size_t numUncertainVars

  total number of uncertain variables (native space)

- RealVectorArray requestedRespLevels

  requested response levels for all response functions

- RealVectorArray computedProbLevels

  output probability levels for all response functions resulting from requestedRespLevels

- RealVectorArray computedRelLevels

  output reliability levels for all response functions resulting from requestedRespLevels

- RealVectorArray computedGenRelLevels

  output generalized reliability levels for all response functions resulting from requestedRespLevels

- short respLevelTarget

  indicates mapping of z->p (PROBABILITIES), z->beta (RELIABILITIES), or z->beta∗ (GEN_RELIABILITIES)

- RealVectorArray requestedProbLevels

  requested probability levels for all response functions

- RealVectorArray requestedRelLevels

  requested reliability levels for all response functions

- RealVectorArray requestedGenRelLevels

  requested generalized reliability levels for all response functions

- RealVectorArray computedRespLevels

  output response levels for all response functions resulting from requestedProbLevels, requestedRelLevels, or requestedGenRelLevels

- size_t totalLevelRequests

  total number of levels specified within requestedRespLevels, requestedProbLevels, and requestedRelLevels

- bool cdfFlag

  flag for type of probabilities/reliabilities used in mappings: cumulative/CDF (true) or complementary/CCDF (false)

- bool pdfOutput

  flag for managing output of response probability density functions (PDFs)

- Response finalStatistics

  final statistics from the uncertainty propagation used in strategies: response means, standard deviations, and probabilities of failure

Static Protected Attributes

- static NonD * nondInstance

  pointer to the active object instance used within static evaluator functions in order to avoid the need for static data

Private Member Functions

- void distribute_levels (RealVectorArray &levels, bool ascending=true)

  convenience function for distributing a vector of levels among multiple response functions if a short-hand specification is employed.

Private Attributes

- bool distParamDerivs

  flags calculation of derivatives with respect to distribution parameters s within resp_x_to_u_mapping() using the chain rule df/dx dx/ds. The default is to calculate derivatives with respect to standard random variables u using the chain rule df/dx dx/du.

43.70.1 Detailed Description

Base class for all nondetermistic iterators (the DAKOTA/UQ branch). The base class for nondeterministic iterators consolidates uncertain variable data and probabilistic utilities for inherited classes.

43.70.2 Member Function Documentation

43.70.2.1 void initialize_random_variables (short *u_space_type*)

initialize natafTransform based on distribution data from iteratedModel Build ProbabilityTransformation::ranVar arrays containing the uncertain variable distribution types and their corresponding means/standard deviations. This function is used when the Model variables are in x-space.

References NonD::initialize_random_variable_correlations(), NonD::initialize_random_variable_parameters(), NonD::initialize_random_variable_transformation(), NonD::initialize_random_variable_types(), NonD::natafTransform, and NonD::verify_correlation_support().

Referenced by NonDExpansion::compute_statistics(), NonDGlobalReliability::importance_sampling(), NonDLocalReliability::initialize_class_data(), NonDExpansion::initialize_expansion(), NonDAdaptImpSampling::NonDAdaptImpSampling(), and NonDGlobalReliability::optimize_gaussian_process().

43.70.2.2 void initialize_random_variables (const Pecos::ProbabilityTransformation & *transform*)

alternate form: initialize natafTransform based on incoming data This function is commonly used to publish tranformation data when the Model variables are in a transformed space (e.g., u-space) and ProbabilityTransformation::ranVarTypes et al. may not be generated directly. This allows for the use of inverse transformations to return the transformed space variables to their original states.

References NonD::initialize_random_variable_transformation(), NonD::natafTransform, NonD::numContDesVars, and NonD::numContStateVars.

43.70.2.3 void initialize_run () `[inline, protected, virtual]`

utility function to perform common operations prior to pre_run(); typically memory initialization; setting of instance pointers Perform initialization phases of run sequence, like allocating memory and setting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's initialize_run(), typically _before_ performing its own implementation steps.

Reimplemented from Iterator.

References NonD::nondInstance, and NonD::prevNondInstance.

43.70.2.4 void run () `[inline, protected, virtual]`

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Iterator.

References Analyzer::bestVarsRespMap, and NonD::quantify_uncertainty().

43.70.2.5 void finalize_run () `[inline, protected, virtual]`

utility function to perform common operations following post_run(); deallocation and resetting of instance pointers Optional: perform finalization phases of run sequence, like deallocating memory and resetting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's finalize_run(), typically _after_ performing its own implementation steps.

Reimplemented from Iterator.

References NonD::nondInstance, and NonD::prevNondInstance.

43.70.2.6 void initialize_final_statistics () `[protected, virtual]`

initializes finalStatistics for storing NonD final results Default definition of virtual function (used by sampling, reliability, and stochastic expansion methods) defines the set of statistical results to include means, standard deviations, and level mappings.

Reimplemented in NonDInterval.

References NonD::cdfFlag, Model::cv(), ActiveSet::derivative_vector(), NonD::finalStatistics, Response::function_labels(), Model::inactive_continuous_variable_ids(), Iterator::iteratedModel, NonD::numEpistemicUncVars, Iterator::numFunctions, NonD::requestedGenRelLevels, NonD::requestedProbLevels, NonD::requestedRelLevels, NonD::requestedRespLevels, NonD::respLevelTarget, and NonD::totalLevelRequests.

Referenced by NonDExpansion::NonDExpansion(), NonDIntegration::NonDIntegration(), NonDReliability::NonDReliability(), NonDSampling::NonDSampling(), and NonD::requested_levels().

43.70.2.7 void initialize_random_variable_types (short *u_space_type*) [protected]

initializes ranVarTypesX and ranVarTypesU within natafTransform Build ProbabilityTransformation::ranVar arrays containing the uncertain variable distribution types and their corresponding means/standard deviations. This function is used when the Model variables are in x-space.

References Dakota::abort_handler(), Model::cv(), Model::distribution_parameters(), Iterator::iteratedModel, NonD::natafTransform, NonD::numBetaVars, NonD::numContDesVars, NonD::numContStateVars, NonD::numExponentialVars, NonD::numFrechetVars, NonD::numGammaVars, NonD::numGumbelVars, NonD::numHistogramBinVars, NonD::numHistogramPtVars, NonD::numIntervalVars, NonD::numLognormalVars, NonD::numLoguniformVars, NonD::numNormalVars, NonD::numTriangularVars, NonD::numUniformVars, and NonD::numWeibullVars.

Referenced by NonDExpansion::initialize(), NonD::initialize_random_variables(), NonDBayesCalibration::NonDBayesCalibration(), NonDIntegration::NonDIntegration(), and NonDReliability::NonDReliability().

43.70.2.8 void initialize_random_variable_parameters () [protected]

initializes ranVarMeansX, ranVarStdDevsX, ranVarLowerBndsX, ranVarUpperBndsX, and ranVarAddtlParamsX within natafTransform Build ProbabilityTransformation::ranVar arrays containing the uncertain variable distribution types and their corresponding means/standard deviations. This function is used when the Model variables are in x-space.

References Dakota::abort_handler(), Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::cv(), Model::distribution_parameters(), Iterator::iteratedModel, NonD::natafTransform, NonD::numBetaVars, NonD::numContDesVars, NonD::numContStateVars, NonD::numExponentialVars, NonD::numFrechetVars, NonD::numGammaVars, NonD::numGumbelVars, NonD::numHistogramBinVars, NonD::numHistogramPtVars, NonD::numIntervalVars, NonD::numLognormalVars, NonD::numLoguniformVars, NonD::numNormalVars, NonD::numTriangularVars, NonD::numUniformVars, and NonD::numWeibullVars.

Referenced by NonDExpansion::initialize_expansion(), NonD::initialize_random_variables(), NonDLocalReliability::quantify_uncertainty(), NonDGlobalReliability::quantify_uncertainty(), and NonDBayesCalibration::quantify_uncertainty().

43.70.2.9 void vars_u_to_x_mapping (const Variables & *u_vars*, Variables & *x_vars*) [static, protected]

static function for RecastModels used to map u-space variables from NonD Iterators to x-space variables for Model evaluations. Map the variables from iterator space (u) to simulation space (x).

References Variables::continuous_variables(), NonD::natafTransform, and NonD::nondInstance.

Referenced by NonD::construct_u_space_model().

43.70.2.10 void set_u_to_x_mapping (const Variables & *u_vars*, const ActiveSet & *u_set*, ActiveSet & *x_set*) `[static, protected]`

static function for RecastModels used to map u-space ActiveSets from NonD Iterators to x-space ActiveSets for Model evaluations. Define the DVV for x-space derivative evaluations by augmenting the iterator requests to account for correlations.

References Dakota::_NPOS, Variables::all_continuous_variable_ids(), Dakota::contains(), Variables::continuous_variable_ids(), ActiveSet::derivative_vector(), Dakota::find_index(), Variables::inactive_-continuous_variable_ids(), NonD::natafTransform, and NonD::nondInstance.

Referenced by NonD::construct_u_space_model().

The documentation for this class was generated from the following files:

- DakotaNonD.H
- DakotaNonD.C

43.71 NonDAdaptImpSampling Class Reference

Class for the Adaptive Importance Sampling methods within DAKOTA. Inheritance diagram for NonDAdaptImpSampling::

Public Member Functions

- NonDAdaptImpSampling (Model &model)

  constructors

- NonDAdaptImpSampling (Model &model, const String &sample_type, int samples, int seed, const String &rng, short is_type, bool cdf_flag, bool x_space_data, bool x_space_model, bool bounded_model)
- ~NonDAdaptImpSampling ()

  destructor

- void quantify_uncertainty ()

  performs an adaptive importance sampling and returns probability of failure.

- void initialize (const RealVectorArray &initial_points, int resp_fn, const Real &initial_prob, const Real &failure_threshold)

  initializes data needed for importance sampling: an initial set of points around which to sample, a failure threshold, an initial probability to refine, and flags to control transformations

- void initialize (const RealVector &initial_point, int resp_fn, const Real &initial_prob, const Real &failure_threshold)

  initializes data needed for importance sampling: an initial point around which to sample, a failure threshold, an initial probability to refine, and flags to control transformations

- const Real & get_probability ()

  returns the probability calculated by the importance sampling

Private Member Functions

- void converge_cov ()

  iteratively generate samples and select representative points until coefficient of variation converges

- void converge_probability ()

  iteratively generate samples from final set of representative points until probability converges

- void select_init_rep_points (const RealVectorArray &samples)

  select representative points from initial set of samples

- void select_rep_points (const RealVectorArray &samples)

  select representative points from a set of samples

- void calculate_rep_weights ()

  calculate relative weights of representative points

- void generate_samples (RealVectorArray &samples)

  generate a set of samples based on multimodal sampling density

- void calculate_statistics (const RealVectorArray &samples, const size_t &total_sample_number, Real &probability_sum, Real &probability, bool cov_flag, Real &variance_sum, Real &coeff_of_variation)

  calculate the probability of exceeding the failure threshold and the coefficent of variation (if requested)

Private Attributes

- short importanceSamplingType

  integration type (is, ais, mmais) provided by input specification

- bool invertProb

  flag for inversion of probability values using 1.-p

- size_t numRepPoints

  the number of representative points around which to sample

- size_t respFn

  the response function in the model to be sampled

- RealVectorArray initPoints

  the original set of samples passed into the MMAIS routine

- RealVectorArray repPoints

  the set of representative points around which to sample

- RealVector repWeights

the weight associated with each representative point

- RealVector designPoint

  design point at which uncertain space is being sampled

- bool transInitPoints

  flag to control if x->u transformation should be performed for initial points

- bool transPoints

  flag to control if u->x transformation should be performed before evaluation

- bool useModelBounds

  flag to control if the sampler should respect the model bounds

- bool initLHS

  flag to identify if initial points are generated from an LHS sample

- Real initProb

  the initial probability (from FORM or SORM)

- Real finalProb

  the final calculated probability (p)

- Real failThresh

  the failure threshold (z-bar) for the problem.

43.71.1 Detailed Description

Class for the Adaptive Importance Sampling methods within DAKOTA. The NonDAdaptImpSampling implements the multi-modal adaptive importance sampling used for reliability calculations. (eventually we will want to broaden this). Need to add more detail to this description.

43.71.2 Constructor & Destructor Documentation

43.71.2.1 NonDAdaptImpSampling (Model & *model*)

constructors standard constructor

This is the primary constructor. It accepts a Model reference.

References NonD::initialize_random_variables(), and NonDSampling::samplingVarsMode.

43.71.2.2 NonDAdaptImpSampling (Model & *model*, const String & *sample_type*, int *samples*, int *seed*, const String & *rng*, short *is_type*, bool *cdf_flag*, bool *x_space_data*, bool *x_space_model*, bool *bounded_model*)

This is an alternate constructor for instantiations on the fly using a Model but no ProblemDescDB.

References NonD::cdfFlag, and NonDSampling::samplingVarsMode.

43.71.3 Member Function Documentation

43.71.3.1 void initialize (const RealVectorArray & *initial_points*, int *resp_fn*, const Real & *initial_prob*, const Real & *failure_threshold*)

initializes data needed for importance sampling: an initial set of points around which to sample, a failure threshold, an initial probability to refine, and flags to control transformations Initializes data using a set of starting points.

References NonDAdaptImpSampling::designPoint, NonDAdaptImpSampling::failThresh, NonDAdaptImpSampling::initPoints, NonDAdaptImpSampling::initProb, NonD::natafTransform, NonD::numContDesVars, Iterator::numContinuousVars, NonD::numUncertainVars, NonDAdaptImpSampling::respFn, and NonDAdaptImpSampling::transInitPoints.

Referenced by NonDExpansion::compute_statistics(), NonDGlobalReliability::importance_sampling(), NonDLocalReliability::probability(), and NonDAdaptImpSampling::quantify_uncertainty().

43.71.3.2 void initialize (const RealVector & *initial_point*, int *resp_fn*, const Real & *initial_prob*, const Real & *failure_threshold*)

initializes data needed for importance sampling: an initial point around which to sample, a failure threshold, an initial probability to refine, and flags to control transformations Initializes data using only one starting point.

References NonDAdaptImpSampling::designPoint, NonDAdaptImpSampling::failThresh, NonDAdaptImpSampling::initPoints, NonDAdaptImpSampling::initProb, NonD::natafTransform, NonD::numContDesVars, Iterator::numContinuousVars, NonD::numUncertainVars, NonDAdaptImpSampling::respFn, and NonDAdaptImpSampling::transInitPoints.

The documentation for this class was generated from the following files:

- NonDAdaptImpSampling.H
- NonDAdaptImpSampling.C

43.72 NonDBayesCalibration Class Reference

Base class for Bayesian inference: generates posterior distribution on model parameters given experimental data. Inheritance diagram for NonDBayesCalibration::

Public Member Functions

- NonDBayesCalibration (Model &model)

  standard constructor

- ~NonDBayesCalibration ()

  destructor

Protected Member Functions

- void quantify_uncertainty ()

  performs a forward uncertainty propagation of parameter distributions into response statistics

- const Model & algorithm_space_model () const

  return the result of any recasting or surrogate model recursion layered on top of iteratedModel by the derived Iterator ctor chain

Protected Attributes

- Model emulatorModel

  Model instance employed in the likelihood function; provides response function values from Gaussian processes, stochastic expansions (PCE/SC), or direct access to simulations (no surrogate option).

- bool standardizedSpace

  flag indicating use of a variable transformation to standardized probability space

- Iterator stochExpIterator

  NonDPolynomialChaos or NonDStochCollocation instance for defining a PCE/SC-based emulatorModel.

Private Attributes

- short emulatorType

  the emulator type: NO_EMULATOR, GAUSSIAN_PROCESS, POLYNOMIAL_CHAOS, or STOCHASTIC_-COLLOCATION

43.72.1 Detailed Description

Base class for Bayesian inference: generates posterior distribution on model parameters given experimental data. This class will eventually provide a general-purpose framework for Bayesian inference. In the short term, it only collects shared code between QUESO and GPMSA implementations.

43.72.2 Constructor & Destructor Documentation

43.72.2.1 NonDBayesCalibration (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

References Iterator::algorithm_space_model(), Model::assign_rep(), Iterator::assign_rep(), NonD::cdfFlag, NonD::construct_lhs(), NonD::construct_u_space_model(), NonDBayesCalibration::emulatorModel, NonDBayesCalibration::emulatorType, ProblemDescDB::get_bool(), ProblemDescDB::get_dusa(), ProblemDescDB::get_int(), ProblemDescDB::get_string(), Iterator::gradientType, Iterator::hessianType, Model::init_communicators(), NonD::initialize_random_variable_correlations(), NonD::initialize_random_-variable_transformation(), NonD::initialize_random_variable_types(), Iterator::iteratedModel, Iterator::iterator_-rep(), Iterator::probDescDB, NonD::requested_levels(), NonD::respLevelTarget, NonDBayesCalibration::standardizedSpace, NonDBayesCalibration::stochExpIterator, and NonD::verify_correlation_support().

The documentation for this class was generated from the following files:

- NonDBayesCalibration.H
- NonDBayesCalibration.C

43.73 NonDCalibration Class Reference

Inheritance diagram for NonDCalibration::

Public Member Functions

- NonDCalibration (Model &model)

  standard constructor

- ~NonDCalibration ()

  destructor

Protected Member Functions

- void set_configuration_vars (Model &model, const RealVector &x)

  set the passed configuration variables into the model's state vars

Protected Attributes

- RealVector expStdDeviations

  1 or numFunctions standard deviations

- String expDataFileName

  filename from which to read experimental data; optionally configuration vars x and standard deviations sigma

- bool expDataFileAnnotated

  whether the data file is in annotated format

- size_t numExperiments

  number of experiments to read from data file

- size_t numExpConfigVars

  number of columns in data file which are state variables

- size_t numExpStdDeviationsRead

  how many sigmas to read from the data file (1 or numFunctions)

Private Member Functions

- bool find_state_index (unsigned short state_type, UShortMultiArrayConstView variable_types, std::string context_message, size_t &start_index)

  helper function to lookup a state_type enum variable type in the array of variables_types to find its start_index into the all array

Private Attributes

- size_t continuousConfigVars

  number of continuous configuration variables

- size_t discreteIntConfigVars

  number of discrete integer configuration variables

- size_t discreteRealConfigVars

  number of discrete real configuration variables

- size_t continuousConfigStart

  index of configuration variables in all continuous array

- size_t discreteIntConfigStart

  index of configuration variables in all discrete integer array

- size_t discreteRealConfigStart

  index of configuration variables in all discrete real array

43.73.1 Detailed Description

This class ...

43.73.2 Constructor & Destructor Documentation

43.73.2.1 NonDCalibration (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

References Dakota::abort_handler(), Model::all_continuous_variable_types(), Model::all_discrete_int_variable_-types(), Model::all_discrete_real_variable_types(), NonDCalibration::continuousConfigStart, NonDCalibration::continuousConfigVars, NonDCalibration::discreteIntConfigStart, NonDCalibration::discreteIntConfigVars, NonDCalibration::discreteRealConfigStart, NonDCalibration::discreteRealConfigVars, NonDCalibration::expDataFileName, NonDCalibration::expStdDeviations, NonDCalibration::find_state_index(), ProblemDescDB::get_sizet(), Iterator::iteratedModel, NonDCalibration::numExpConfigVars, NonDCalibration::numExperiments, Iterator::numFunctions, and Iterator::probDescDB.

The documentation for this class was generated from the following files:

- NonDCalibration.H
- NonDCalibration.C

43.74 NonDCubature Class Reference

Derived nondeterministic class that generates N-dimensional numerical cubature points for evaluation of expectation integrals. Inheritance diagram for NonDCubature::

Public Member Functions

- NonDCubature (Model &model, const Pecos::ShortArray &u_types, unsigned short cub_int_order)
- unsigned short integrand_order () const

  return cubIntOrder

Protected Member Functions

- NonDCubature (Model &model)

  constructor

- ~NonDCubature ()

  destructor

- void initialize_grid (const std::vector< Pecos::BasisPolynomial > &poly_basis)

  initialize integration grid by drawing from polynomial basis settings

- void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)
- void increment_grid ()

  increment SSG level/TPQ order

- void increment_grid_preference (const RealVector &dim_pref)
- int num_samples () const

get the current number of samples

Private Member Functions

- void anisotropic_preference (const RealVector &dim_pref)

  update cubIntOrder based on an updated dimension preference

- void check_integration (const Pecos::ShortArray &u_types, const Pecos::DistributionParams &dp)

  verify self-consistency of integration specification

- void increment_reference ()

  increment each cubIntOrderRef entry by 1

Private Attributes

- Pecos::CubatureDriver * cubDriver

  convenience pointer to the numIntDriver representation

- unsigned short cubIntOrderRef

  reference point for Pecos::CubatureDriver::cubIntOrder: the original user specification for the number of Gauss points per dimension, plus any refinements posted by increment_grid()

- unsigned short cubIntRule

  the isotropic cubature integration rule

43.74.1 Detailed Description

Derived nondeterministic class that generates N-dimensional numerical cubature points for evaluation of expectation integrals. This class is used by NonDPolynomialChaos, but could also be used for general numerical integration of moments. It employs Stroud cubature rules and extensions by D. Xiu.

43.74.2 Constructor & Destructor Documentation

43.74.2.1 NonDCubature (Model & *model*, const Pecos::ShortArray & *u_types*, unsigned short *cub_int_order*)

This alternate constructor is used for on-the-fly generation and evaluation of numerical cubature points.

References NonDCubature::check_integration(), NonDCubature::cubDriver, NonDCubature::cubIntOrderRef, Model::distribution_parameters(), Iterator::iteratedModel, and NonDIntegration::numIntDriver.

43.74.2.2 NonDCubature (Model & *model*) [protected]

constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_-list_nodes has been called and probDescDB can be queried for settings from the method specification. It is not currently used, as there is not yet a separate nond_cubature method specification.

References NonDCubature::check_integration(), NonDIntegration::check_variables(), NonDCubature::cubDriver, NonDCubature::cubIntOrderRef, NonDCubature::cubIntRule, Model::distribution_parameters(), Iterator::iteratedModel, Iterator::maxConcurrency, NonD::natafTransform, and NonDIntegration::numIntDriver.

43.74.3 Member Function Documentation

43.74.3.1 void sampling_reset (int *min_samples*, bool *all_data_flag*, bool *stats_flag*) [protected, virtual]

used by DataFitSurrModel::build_global() to publish the minimum number of points needed from the cubature routine in order to build a particular global approximation.

Reimplemented from Iterator.

References NonDCubature::cubDriver, and NonDCubature::cubIntOrderRef.

43.74.3.2 void increment_grid_preference (const RealVector & *dim_pref*) [inline, protected, virtual]

Should not be used, but pure virtual must be defined.

Reimplemented from NonDIntegration.

References NonDCubature::increment_grid().

43.74.3.3 int num_samples () const [inline, protected, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References NonDCubature::cubDriver.

43.74.3.4 void increment_reference () [inline, private]

increment each cubIntOrderRef entry by 1 cubIntOrderRef is a reference point for CubatureDriver::cubIntOrder, e.g., a lower bound

References NonDCubature::cubIntOrderRef.

Referenced by NonDCubature::increment_grid().

The documentation for this class was generated from the following files:

- NonDCubature.H
- NonDCubature.C

43.75 NonDExpansion Class Reference

Base class for polynomial chaos expansions (PCE) and stochastic collocation (SC). Inheritance diagram for NonDExpansion::

Public Member Functions

- NonDExpansion (Model &model)

  standard constructor

- NonDExpansion (Model &model, short exp_coeffs_approach, short u_space_type, bool piecewise_basis, bool use_derivs)

  alternate constructor

- ~NonDExpansion ()

  destructor

- void quantify_uncertainty ()

  perform a forward uncertainty propagation using PCE/SC methods

- void print_results (std::ostream &s)

  print the final statistics

- const Model & algorithm_space_model () const

  return the result of any recasting or surrogate model recursion layered on top of iteratedModel by the derived Iterator ctor chain

Protected Member Functions

- virtual void resolve_inputs (short &u_space_type, short &data_order)

  perform error checks and mode overrides

- virtual void initialize_u_space_model ()

  initialize uSpaceModel polynomial approximations with PCE/SC data

- virtual void initialize_expansion ()

  initialize random variable definitions and final stats arrays

- virtual void compute_expansion ()

  form the expansion by calling uSpaceModel.build_approximation()

- virtual void increment_expansion ()

  uniformly increment the expansion order (PCE only)

- virtual void print_coefficients (std::ostream &s)

  print expansion coefficients, as supported by derived instance

- void initialize (short u_space_type)

  common constructor code for initialization of natafTransform

- void refine_expansion ()

  refine the reference expansion found by compute_expansion() using uniform/adaptive p-/h-refinement strategies

- void update_expansion ()

  update the expansion by calling uSpaceModel.build_approximation(); avoids unnecessary overhead in compute_-expansion()

- void update_hierarchy ()

  update settings for subsequent passes in the case of multifidelity models

- void construct_cubature (Iterator &u_space_sampler, Model &g_u_model, unsigned short cub_int_order)

  assign a NonDCubature instance within u_space_sampler

- void construct_quadrature (Iterator &u_space_sampler, Model &g_u_model, const UShortArray &quad_-order, const RealVector &dim_pref)

  assign a NonDQuadrature instance within u_space_sampler based on a quad_order specification

- void construct_quadrature (Iterator &u_space_sampler, Model &g_u_model, int filtered_samples, const RealVector &dim_pref)

  assign a NonDQuadrature instance within u_space_sampler based on the size of a filtered tensor product sample set

- void construct_sparse_grid (Iterator &u_space_sampler, Model &g_u_model, const UShortArray &ssg_-level, const RealVector &ssg_dim_pref)

  assign a NonDSparsegrid instance within u_space_sampler

- void construct_expansion_sampler ()

construct the expansionSampler operating on uSpaceModel

- void compute_statistics ()

  calculate analytic and numerical statistics from the expansion

- void update_final_statistics ()

  update finalStatistics

Protected Attributes

- Model uSpaceModel

  Model representing the approximate response function in u-space, after u-space recasting and orthogonal polynomial data fit recursions.

- short expansionCoeffsApproach

  method for collocation point generation and subsequent calculation of the expanion coefficients

- size_t numUncertainQuant

  number of invocations of quantify_uncertainty()

- int numSamplesOnModel

  number of truth samples performed on g_u_model to form the expansion

- int numSamplesOnExpansion

  number of approximation samples performed on the polynomial expansion in order to estimate probabilities

- bool nestedRules

  flag for indicating state of `nested` *and* `non_nested` *overrides of default rule nesting, which depends on the type of integration driver*

- bool piecewiseBasis

  flag for `piecewise` *specification, indicating usage of local basis polynomials within the stochastic expansion*

- bool useDerivs

  flag for `use_derivatives` *specification, indicating usage of derivative data (with respect to expansion variables) to enhance the calculation of the stochastic expansion.*

- short refineType

  refinement type: NO_REFINEMENT, P_REFINEMENT, or H_REFINEMENT

- short refineControl

  refinement control: NO_CONTROL, UNIFORM_CONTROL, DIMENSION_ADAPTIVE_TOTAL_SOBOL, DIMENSION_ADAPTIVE_SPECTRAL_DECAY, or DIMENSION_ADAPTIVE_GENERALIZED_SPARSE

Private Member Functions

- void reduce_total_sobol_sets (RealVector &avg_sobol)

  compute average of total Sobol' indices (from VBD) across the response set for use as an anisotropy indicator

- void reduce_decay_rate_sets (RealVector &min_decay)

  compute minimum of spectral coefficient decay rates across the response set for use as an anisotropy indicator

- void initialize_sets ()

  initialization of adaptive refinement using generalized sparse grids

- Real increment_sets ()

  perform an adaptive refinement increment using generalized sparse grids

- void finalize_sets (bool converged_within_tol)

  finalization of adaptive refinement using generalized sparse grids

- Real compute_covariance_metric (const RealSymMatrix &resp_covar_ref)

  compute 2-norm of change in response covariance

- Real compute_final_statistics_metric (const RealVector &final_stats_ref)

  compute 2-norm of change in final statistics

- void compute_covariance ()

  calculate respCovariance

- void compute_off_diagonal_covariance ()

  calculate respCovariance(i,j) for j<i

- void print_moments (std::ostream &s)

  print expansion and numerical moments

- void print_covariance (std::ostream &s)

  print respCovariance

- void print_sobol_indices (std::ostream &s)

  print global sensitivity indices

- void print_local_sensitivity (std::ostream &s)

  print local sensitivities evaluated at initialPtU

- void compute_print_increment_results ()

  manage print of results following a refinement increment

- void compute_print_iteration_results (bool initialize)

  manage print of results following a refinement increment

- void compute_print_converged_results (bool print_override=false)

  manage print of results following convergence of iterative refinement

Private Attributes

- short ruleNestingOverride

  user override of default rule nesting: NO_NESTING_OVERRIDE, NESTED, or NON_NESTED

- short ruleGrowthOverride

  user override of default rule growth: NO_GROWTH_OVERRIDE, RESTRICTED, or UNRESTRICTED

- Iterator expansionSampler

  Iterator used for sampling on the uSpaceModel to generate approximate probability/reliability/response level statistics. Currently this is an LHS sampling instance, but AIS could also be used.

- Iterator importanceSampler

  Iterator used to refine the approximate probability estimates generated by the expansionSampler using importance sampling.

- bool expSampling

  flag to indicate calculation of numerical statistics by sampling on the expansion

- bool impSampling

  flag to use LHS sampling or MMAIS sampling on the expansion

- RealVector initialPtU

  stores the initial variables data in u-space

- RealMatrix expGradsMeanX

  derivative of the expansion with respect to the x-space variables evaluated at the means (used as uncertainty importance metrics)

- RealSymMatrix respCovariance

  symmetric matrix of analytic response covariance

- short vbdControl

  enumeration for controlling VBD calculation and output: NO_VBD, UNIVARIATE_VBD, or ALL_VBD

- Real vbdDropTol

  tolerance for omitting output of small VBD indices

43.75.1 Detailed Description

Base class for polynomial chaos expansions (PCE) and stochastic collocation (SC). The NonDExpansion class provides a base class for methods that use polynomial expansions to approximate the effect of parameter uncertainties on response functions of interest.

43.75.2 Member Function Documentation

43.75.2.1 void compute_statistics () [protected]

calculate analytic and numerical statistics from the expansion Calculate analytic and numerical statistics from the expansion and log results within final_stats for use in OUU.

References Dakota::abort_handler(), Iterator::active_set(), Response::active_set_derivative_vector(), Response::active_set_request_vector(), Iterator::all_responses(), Iterator::all_samples(), Model::approximation_data(), Model::approximations(), NonD::cdfFlag, PecosApproximation::compute_component_effects(), PecosApproximation::compute_moments(), NonDExpansion::compute_off_diagonal_covariance(), PecosApproximation::compute_total_effects(), NonD::computedGenRelLevels, NonD::computedProbLevels, NonD::computedRelLevels, NonD::computedRespLevels, Model::continuous_variable_ids(), Model::continuous_variables(), Dakota::copy_data(), PecosApproximation::expansion_coefficient_flag(), NonDExpansion::expansionSampler, NonDExpansion::expGradsMeanX, NonDExpansion::expSampling, NonD::finalStatistics, Response::function_gradient(), Response::function_value(), Response::function_values(), NonDAdaptImpSampling::get_probability(), NonDExpansion::importanceSampler, NonDExpansion::impSampling, Iterator::initial_points(), NonDAdaptImpSampling::initialize(), NonD::initialize_distribution_mappings(), NonD::initialize_random_variables(), NonDExpansion::initialPtU, Iterator::iteratedModel, Iterator::iterator_rep(), PecosApproximation::mean_gradient(), PecosApproximation::moments(), NonD::natafTransform, NonD::numContDesVars, NonD::numContEpistUncVars, Iterator::numContinuousVars, NonD::numContStateVars, Iterator::numFunctions, NonDExpansion::numSamplesOnExpansion, Iterator::outputLevel, NonDExpansion::refineControl, ActiveSet::request_vector(), NonD::requestedGenRelLevels, NonD::requestedProbLevels, NonD::requestedRelLevels, NonD::requestedRespLevels, NonDExpansion::respCovariance, NonD::respLevelTarget, Iterator::response_results(), Iterator::run_iterator(), Iterator::subIteratorFlag, NonD::totalLevelRequests, NonDSampling::update_final_statistics(), NonDExpansion::uSpaceModel, PecosApproximation::variance_gradient(), and NonDExpansion::vbdControl.

Referenced by NonDExpansion::compute_final_statistics_metric(), NonDExpansion::compute_print_converged_results(), NonDExpansion::compute_print_increment_results(), and NonDExpansion::compute_print_iteration_results().

43.75.2.2 Real compute_final_statistics_metric (const RealVector & *final_stats_ref*) [private]

compute 2-norm of change in final statistics computes a "goal-oriented" refinement metric employing finalStatistics

References NonDExpansion::compute_statistics(), NonD::finalStatistics, Response::function_values(), Iterator::numFunctions, NonD::requestedGenRelLevels, NonD::requestedProbLevels, NonD::requestedRelLevels, and NonD::requestedRespLevels.

Referenced by NonDExpansion::increment_sets().

43.75.3 Member Data Documentation

43.75.3.1 bool useDerivs [protected]

flag for `use_derivatives` specification, indicating usage of derivative data (with respect to expansion variables) to enhance the calculation of the stochastic expansion. This is part of the method specification since the instantiation of the global data fit surrogate is implicit with no user specification. This behavior is distinct from the usage of response derivatives with respect to auxilliary variables (design, epistemic) for computing derivatives of aleatory expansion statistics with respect to these variables.

Referenced by NonDExpansion::compute_expansion(), NonDStochCollocation::initialize_u_space_model(), NonDPolynomialChaos::initialize_u_space_model(), NonDStochCollocation::resolve_inputs(), NonDPolynomialChaos::resolve_inputs(), NonDPolynomialChaos::terms_ratio_to_samples(), and NonDPolynomialChaos::terms_samples_to_ratio().

The documentation for this class was generated from the following files:

- NonDExpansion.H
- NonDExpansion.C

43.76 NonDGlobalEvidence Class Reference

Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ. Inheritance diagram for NonDGlobalEvidence::

Public Member Functions

- NonDGlobalEvidence (Model &model)

  constructor

- ~NonDGlobalEvidence ()

  destructor

- void initialize ()

  perform any required initialization

- void set_cell_bounds ()

  set the optimization variable bounds for each cell

- void get_best_sample (bool find_max, bool eval_approx)

  determine truthFnStar and approxFnStar

- void post_process_cell_results (bool minimize)

  post-process a cell minimization/maximization result

- void post_process_response_fn_results ()

  post-process the interval computed for a response function

- void post_process_final_results ()

  perform final post-processing

43.76.1 Detailed Description

Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ. The NonDEvidence class implements the propagation of epistemic uncertainty using Dempster-Shafer theory of evidence. In this approach, one assigns a set of basic probability assignments (BPA) to intervals defined for the uncertain variables. Input interval combinations are calculated, along with their BPA. Currently, the response function is evaluated at a set of sample points, then a response surface is constructed which is sampled extensively to find the minimum and maximum within each input interval cell, corresponding to the belief and plausibility within that cell, respectively. This data is then aggregated to calculate cumulative distribution functions for belief and plausibility.

The documentation for this class was generated from the following files:

- NonDGlobalEvidence.H
- NonDGlobalEvidence.C

43.77 NonDGlobalInterval Class Reference

Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. Inheritance diagram for NonDGlobalInterval::

Public Member Functions

- NonDGlobalInterval (Model &model)

  constructor

- ~NonDGlobalInterval ()

  destructor

- void quantify_uncertainty ()

  Performs an optimization to determine interval bounds for an entire function or interval bounds on a particular statistical estimator.

- const Model & algorithm_space_model () const

  return the result of any recasting or surrogate model recursion layered on top of iteratedModel by the derived Iterator ctor chain

Protected Member Functions

- virtual void initialize ()

  perform any required initialization

- virtual void set_cell_bounds ()

  set the optimization variable bounds for each cell

- virtual void get_best_sample (bool find_max, bool eval_approx)

  determine truthFnStar and approxFnStar

- virtual void post_process_cell_results (bool minimize)

  post-process a cell minimization/maximization result

- virtual void post_process_response_fn_results ()

  post-process the interval computed for a response function

- virtual void post_process_final_results ()

  perform final post-processing

- void post_process_gp_results ()

  post-process a GP-based optimization iteration: output EIF maximization results, update convergence controls, and update GP approximation

Protected Attributes

- Iterator daceIterator

  LHS iterator for constructing initial GP for all response functions.

- Iterator gpOptimizer

  NCSU DIRECT optimizer for maximizing expected improvement.

- Model fHatModel

  GP model of response, one approximation per response function.

- Model eifModel

  recast model which assimilates mean and variance to solve the max(EIF) sub-problem

- Real approxFnStar

  approximate response corresponding to minimum/maximum truth response

- Real truthFnStar

  minimum/maximum truth response function value

Static Private Member Functions

- static void EIF_objective_min (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in the Expected Improvement Function (EIF) for minimizing the GP

- static void EIF_objective_max (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

static function used as the objective function in the Expected Improvement Function (EIF) for maximizing the GP

Private Attributes

- const int seedSpec

  the user seed specification (default is 0)

- int numSamples

  the number of samples used in the surrogate

- String rngName

  name of the random number generator

- size_t eifConvergenceCntr

  counter for number of successive iterations that the optimal EIF is less than the convergenceTol

- size_t distConvergenceCntr

  counter for number of successive iterations that the L_2 change in optimal solution is less than the convergenceTol

- RealVector prevCStar

  stores previous optimal points for convergence

- size_t sbIterNum

  surrogate-based minimization/maximization iteration count

- bool approxConverged

  flag indicating convergence of a GP minimization or maximization

- bool allResponsesPerIter

  flag for maximal response extraction

- short dataOrder

  order of the data used for surrogate construction, in ActiveSet request vector 3-bit format; user may override responses spec

Static Private Attributes

- static NonDGlobalInterval * nondGIInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.77.1 Detailed Description

Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. The NonDGlobalInterval class supports global nongradient-based optimization approaches to determining interval bounds for epistemic UQ. The interval bounds may be on the entire function in the case of pure interval analysis (e.g. intervals on input = intervals on output), or the intervals may be on statistics of an "inner loop" aleatory analysis such as intervals on means, variances, or percentile levels. The preliminary implementation will use a Gaussian process surrogate to determine interval bounds.

The documentation for this class was generated from the following files:

- NonDGlobalInterval.H
- NonDGlobalInterval.C

43.78 NonDGlobalReliability Class Reference

Class for global reliability methods within DAKOTA/UQ. Inheritance diagram for NonDGlobalReliability::

Public Member Functions

- NonDGlobalReliability (Model &model)

  constructor

- ~NonDGlobalReliability ()

  destructor

- void quantify_uncertainty ()

  performs an uncertainty propagation using analytical reliability methods which solve constrained optimization problems to obtain approximations of the cumulative distribution function of response

- void print_results (std::ostream &s)

  print the approximate mean, standard deviation, and importance factors when using the mean value method or the CDF/CCDF information when using MPP-search-based reliability methods

Private Member Functions

- void optimize_gaussian_process ()

  construct the GP using EGO/SKO

- void importance_sampling ()

  perform multimodal adaptive importance sampling on the GP

- void get_best_sample ()

  determine current best solution from among sample data for expected imporovement function in Performance Measure Approach (PMA)

- Real constraint_penalty (const Real &constraint, const RealVector &c_variables)

  calculate the penalty to be applied to the PMA constraint value

- Real expected_improvement (const RealVector &expected_values, const RealVector &c_variables)

  expected improvement function for the GP

- Real expected_feasibility (const RealVector &expected_values, const RealVector &c_variables)

  expected feasibility function for the GP

Static Private Member Functions

- static void EIF_objective_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in the Expected Improvement (EIF) problem formulation for PMA

- static void EFF_objective_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in the Expected Feasibility (EFF) problem formulation for RIA

Private Attributes

- Real fnStar

  minimum penalized response from among true function evaluations

- short meritFunctionType

  type of merit function used to penalize sample data

- Real lagrangeMult

  Lagrange multiplier for standard Lagrangian merit function.

- Real augLagrangeMult

  Lagrange multiplier for augmented Lagrangian merit function.

- Real penaltyParameter

  penalty parameter for augmented Lagrangian merit funciton

- Real lastConstraintViolation

  constraint violation at last iteration, used to determine if the current iterate should be accepted (must reduce violation)

- bool lastIterateAccepted

  flag to determine if last iterate was accepted this controls update of parameters for augmented Lagrangian merit fn

- short dataOrder

  order of the data used for surrogate construction, in ActiveSet request vector 3-bit format; user may override responses spec

Static Private Attributes

- static NonDGlobalReliability * nondGlobRelInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.78.1 Detailed Description

Class for global reliability methods within DAKOTA/UQ. The NonDGlobalReliability class implements EGO/SKO for global MPP search, which maximizes an expected improvement function derived from Gaussian process models. Once the limit state has been characterized, a multimodal importance sampling approach is used to compute probabilities.

The documentation for this class was generated from the following files:

- NonDGlobalReliability.H
- NonDGlobalReliability.C

43.79 NonDGlobalSingleInterval Class Reference

Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. Inheritance diagram for NonDGlobalSingleInterval::

Public Member Functions

- NonDGlobalSingleInterval (Model &model)
  constructor
- ~NonDGlobalSingleInterval ()
  destructor

Protected Member Functions

- void initialize ()
  perform any required initialization
- void post_process_cell_results (bool minimize)
  post-process a cell minimization/maximization result
- void get_best_sample (bool find_max, bool eval_approx)
  determine truthFnStar and approxFnStar

Private Attributes

- size_t statCntr
  counter for finalStatistics

43.79.1 Detailed Description

Class for using global nongradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. The NonDGlobalSingleInterval class supports global nongradient-based optimization apporaches to determining interval bounds for epistemic UQ. The interval bounds may be on the entire function in the case of pure interval analysis (e.g. intervals on input = intervals on output), or the intervals may be on statistics of an "inner loop" aleatory analysis such as intervals on means, variances, or percentile levels. The preliminary implementation will use a Gaussian process surrogate to determine interval bounds.

The documentation for this class was generated from the following files:

- NonDGlobalSingleInterval.H
- NonDGlobalSingleInterval.C

43.80 NonDGPMSABayesCalibration Class Reference

Generates posterior distribution on model parameters given experiment data. Inheritance diagram for NonDGPMSABayesCalibration::

Public Member Functions

- NonDGPMSABayesCalibration (Model &model)

  standard constructor

- ~NonDGPMSABayesCalibration ()

  destructor

Protected Member Functions

- void quantify_uncertainty ()

  performs a forward uncertainty propagation by using GPM/SA to generate a posterior distribution on parameters given a set of simulation parameter/response data, a set of experimental data, and additional variables to be specified here.

- void print_results (std::ostream &s)

  print the final statistics

Private Attributes

- Iterator lhsSampler

  LHS sampling iterator.

- int numEmulatorSamples

  number of samples to construct the GP emulator

- int numDraws

  number of draws from the MCMC chain

- const int seedSpec

  the user seed specification (default is 0)

- String rngName

  name of the random number generator Storage for the observed x data (inputs) provided by the user

43.80.1 Detailed Description

Generates posterior distribution on model parameters given experiment data. This class provides a wrapper for the functionality provided in the Los Alamos National Laboratory code called GPM/SA (Gaussian Process Models for Simulation Analysis). Although this is a code that provides input/output mapping, it DOES NOT provide the mapping that we usually think of in the NonDeterministic class hierarchy in DAKOTA, where uncertainty in parameter inputs are mapped to uncertainty in simulation responses. Instead, this class takes a pre-existing set of simulation data as well as experimental data, and maps priors on input parameters to posterior distributions on those input parameters, according to a likelihood function. The goal of the MCMC sampling is to produce posterior values of parameter estimates which will produce simulation response values that "match well" to the experimental data. The MCMC is an integral part of the calibration. The data structures in GPM/SA are fairly detailed and nested. Part of this prototyping exercise is to determine what data structures need to be specified and initialized in DAKOTA and sent to GPM/SA, and what data structures will be returned.

43.80.2 Constructor & Destructor Documentation

43.80.2.1 NonDGPMSABayesCalibration (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

References Iterator::assign_rep(), Model::init_communicators(), Iterator::iteratedModel, NonDGPMSABayesCalibration::lhsSampler, Iterator::maximum_concurrency(), NonDGPMSABayesCalibration::numEmulatorSamples, NonDGPMSABayesCalibration::rngName, and NonDGPMSABayesCalibration::seedSpec.

43.80.3 Member Function Documentation

43.80.3.1 void quantify_uncertainty () `[protected, virtual]`

performs a forward uncertainty propagation by using GPM/SA to generate a posterior distribution on parameters given a set of simulation parameter/response data, a set of experimental data, and additional variables to be specified here. This method does all the pre-processing necessary to call the GPM/SA code, including running

LHS on the model to generate the initial samples, doing some normalization, calling the GPM/SA functions, and returning the posterior parameter distributions.

Reimplemented from NonDBayesCalibration.

References Dakota::abort_handler(), Model::acv(), Iterator::all_responses(), Iterator::all_samples(), Analyzer::all_samples(), NonDCalibration::expDataFileAnnotated, NonDCalibration::expDataFileName, NonDCalibration::expStdDeviations, Iterator::iteratedModel, NonDGPMSABayesCalibration::lhsSampler, Model::num_functions(), NonDGPMSABayesCalibration::numDraws, NonDGPMSABayesCalibration::numEmulatorSamples, NonDCalibration::numExpConfigVars, NonDCalibration::numExperiments, NonDCalibration::numExpStdDeviationsRead, Iterator::numFunctions, Dakota::read_data_tabular(), and Iterator::run_iterator().

The documentation for this class was generated from the following files:

- NonDGPMSABayesCalibration.H
- NonDGPMSABayesCalibration.C

43.81 NonDIncremLHSSampling Class Reference

Performs icremental LHS sampling for uncertainty quantification. Inheritance diagram for NonDIncremLHSSampling::

Public Member Functions

- NonDIncremLHSSampling (Model &model)

  constructor

- ~NonDIncremLHSSampling ()

  destructor

- void quantify_uncertainty ()

  performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results.

- void print_results (std::ostream &s)

  print the final statistics

Static Protected Member Functions

- static bool rank_sort (const int &x, const int &y)

  sort algorithm to compute ranks for rank correlations

Private Attributes

- int previousSamples

  number of samples in previous LHS run

- bool varBasedDecompFlag

  flags computation of VBD

Static Private Attributes

- static RealArray rawData

  static data used by static rank_sort() fn

43.81.1 Detailed Description

Performs icremental LHS sampling for uncertainty quantification. The Latin Hypercube Sampling (LHS) package from Sandia Albuquerque's Risk and Reliability organization provides comprehensive capabilities for Monte Carlo and Latin Hypercube sampling within a broad array of user-specified probabilistic parameter distributions. The icremental LHS sampling capability allows one to supplement an initial sample of size n to size 2n while maintaining the correct stratification of the 2n samples and also maintaining the specified correlation structure. The icremental version of LHS will return a sample of size n, which when combined with the original sample of size n, allows one to double the size of the sample.

43.81.2 Constructor & Destructor Documentation

43.81.2.1 NonDIncremLHSSampling (Model & *model*)

constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_-list_nodes has been called and probDescDB can be queried for settings from the method specification.

References NonDSampling::samplingVarsMode.

43.81.3 Member Function Documentation

43.81.3.1 void quantify_uncertainty () `[virtual]`

performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results. Generate incremental samples. Loop over the set of samples and compute responses. Compute statistics on the set of responses if statsFlag is set.

Implements NonD.

References Dakota::abort_handler(), Analyzer::allResponses, Analyzer::allSamples, NonDSampling::compute_-statistics(), Dakota::copy_data(), Dakota::data_pairs, Model::distribution_parameters(), Analyzer::evaluate_parameter_sets(), NonDSampling::get_parameter_sets(), Iterator::iteratedModel, NonD::numBetaVars, Iterator::numContinuousVars, NonD::numExponentialVars, NonD::numGammaVars, NonD::numLognormalVars, NonD::numLoguniformVars, NonD::numNormalVars, NonDSam-pling::numSamples, NonD::numTriangularVars, NonD::numUniformVars, NonD::numWeibullVars, NonD-IncremLHSSampling::previousSamples, NonDIncremLHSSampling::rank_sort(), NonDIncremLHSSam-

pling::rawData, NonDSampling::sampleRanks, NonDSampling::sampleRanksMode, NonDSampling::samplesRef, NonDSampling::sampleType, NonDSampling::varyPattern, and Dakota::write_data().

The documentation for this class was generated from the following files:

- NonDIncremLHSSampling.H
- NonDIncremLHSSampling.C

43.82 NonDIntegration Class Reference

Derived nondeterministic class that generates N-dimensional numerical integration points for evaluation of expectation integrals. Inheritance diagram for NonDIntegration::

Public Member Functions

- virtual void initialize_grid (const std::vector< Pecos::BasisPolynomial > &poly_basis)=0

  initialize integration grid by drawing from polynomial basis settings

- virtual void increment_grid ()=0

  increment SSG level/TPQ order

- virtual void increment_grid_preference (const RealVector &dim_pref)

  increment SSG level/TPQ order and update anisotropy

- virtual void increment_grid_weights (const RealVector &aniso_wts)

  increment SSG level/TPQ order and update anisotropy

- virtual void increment_refinement_sequence ()

  increment sequenceIndex and update active orders/levels

- const Pecos::IntegrationDriver & driver () const

  return numIntDriver

Protected Member Functions

- NonDIntegration (Model &model)

  constructor

- NonDIntegration (NoDBBaseConstructor, Model &model)

alternate constructor for instantiations "on the fly"

- NonDIntegration (NoDBBaseConstructor, Model &model, const RealVector &dim_pref)
  alternate constructor for instantiations "on the fly"
- ~NonDIntegration ()
  destructor
- void quantify_uncertainty ()
  performs a forward uncertainty propagation of parameter distributions into response statistics
- void check_variables (const Pecos::ShortArray &x_types)
  verify self-consistency of variables data
- void print_points_weights (const String &tabular_name)
  output integration points and weights to a tabular file

Protected Attributes

- Pecos::IntegrationDriver numIntDriver
  Pecos utlity class for managing interface to tensor-product grids and VPISparseGrid utilities for Smolyak sparse grids and cubature.
- size_t numIntegrations
  counter for number of integration executions for this object
- size_t sequenceIndex
  index into NonDQuadrature::quadOrderSpec and NonDSparseGrid::ssgLevelSpec that defines the current instance of several possible refinement levels
- RealVector dimPrefSpec
  the user specification for anisotropic dimension preference

43.82.1 Detailed Description

Derived nondeterministic class that generates N-dimensional numerical integration points for evaluation of expectation integrals. This class provides a base class for shared code among NonDQuadrature and NonDSparseGrid.

43.82.2 Constructor & Destructor Documentation

43.82.2.1 NonDIntegration (Model & *model*) `[protected]`

constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_-list_nodes has been called and probDescDB can be queried for settings from the method specification. It is not currently used, as there are not yet separate nond_quadrature/nond_sparse_grid method specifications.

References Dakota::abort_handler(), NonD::initialize_final_statistics(), NonD::initialize_random_variable_correlations(), NonD::initialize_random_variable_transformation(), NonD::initialize_random_variable_types(), Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, and NonD::verify_correlation_support().

43.82.2.2 NonDIntegration (NoDBBaseConstructor, Model & *model*) `[protected]`

alternate constructor for instantiations "on the fly" This alternate constructor is used for on-the-fly generation and evaluation of numerical integration points.

43.82.2.3 NonDIntegration (NoDBBaseConstructor, Model & *model*, const RealVector & *dim_pref*) `[protected]`

alternate constructor for instantiations "on the fly" This alternate constructor is used for on-the-fly generation and evaluation of numerical integration points.

43.82.3 Member Function Documentation

43.82.3.1 void check_variables (const Pecos::ShortArray & *x_types*) `[protected]`

verify self-consistency of variables data Virtual function called from probDescDB-based constructors and from NonDIntegration::quantify_uncertainty()

References Dakota::abort_handler(), NonD::numContAleatUncVars, NonD::numContDesVars, NonD::numContEpistUncVars, Iterator::numContinuousVars, and NonD::numContStateVars.

Referenced by NonDCubature::NonDCubature(), NonDQuadrature::NonDQuadrature(), NonDSparseGrid::NonDSparseGrid(), and NonDIntegration::quantify_uncertainty().

The documentation for this class was generated from the following files:

- NonDIntegration.H
- NonDIntegration.C

43.83 NonDInterval Class Reference

Base class for interval-based methods within DAKOTA/UQ. Inheritance diagram for NonDInterval::

Public Member Functions

- NonDInterval (Model &model)

  constructor

- ~NonDInterval ()

  destructor

- void print_results (std::ostream &s)

  performs an epistemic uncertainty propagation using Dempster-Shafer evidence theory methods which solve for cumulative distribution functions of belief and plausibility

Protected Member Functions

- void initialize_final_statistics ()

  initialize finalStatistics for belief/plausibility results sets

- void compute_evidence_statistics ()

  method for computing belief and plausibility values for response levels or vice-versa

- void calculate_cells_and_bpas ()

  computes the interval combinations (cells) and their bpas replaces CBPIIC_F77 from wrapper calculate_basic_-prob_intervals()

- void calculate_cbf_cpf (bool complementary=true)

  function to compute (complementary) distribution functions on belief and plausibility replaces CCBFPF_F77 from wrapper calculate_cum_belief_plaus()

Protected Attributes

- bool singleIntervalFlag

  flag for SingleInterval derived class

- RealVectorArray ccBelFn

  Storage array to hold CCBF values.

- RealVectorArray ccPlausFn

  Storage array to hold CCPF values.

- RealVectorArray ccBelVal

  Storage array to hold CCB response values.

- RealVectorArray ccPlausVal

  Storage array to hold CCP response values.

- RealVectorArray cellLowerBounds

  Storage array to hold cell lower bounds.

- RealVectorArray cellUpperBounds

  Storage array to hold cell upper bounds.

- Real2DArray cellFnLowerBounds

  Storage array to hold cell min.

- Real2DArray cellFnUpperBounds

  Storage array to hold cell max.

- RealArray cellBPA

  Storage array to hold cell bpa.

- size_t respFnCntr

  response function counter

- size_t cellCntr

  cell counter

- size_t numCells

  total number of interval combinations

43.83.1 Detailed Description

Base class for interval-based methods within DAKOTA/UQ. The NonDInterval class implements the propagation of epistemic uncertainty using either pure interval propagation or Dempster-Shafer theory of evidence. In the latter approach, one assigns a set of basic probability assignments (BPA) to intervals defined for the uncertain variables. Input interval combinations are calculated, along with their BPA. Currently, the response function is evaluated at a set of sample points, then a response surface is constructed which is sampled extensively to find the minimum and maximum within each input interval cell, corresponding to the belief and plausibility within that cell, respectively. This data is then aggregated to calculate cumulative distribution functions for belief and plausibility.

43.83.2 Member Function Documentation

43.83.2.1 void print_results (std::ostream & *s*) `[virtual]`

performs an epistemic uncertainty propagation using Dempster-Shafer evidence theory methods which solve for cumulative distribution functions of belief and plausibility print the cumulative distribution functions for belief and plausibility

Reimplemented from Analyzer.

References NonDInterval::ccBelFn, NonDInterval::ccBelVal, NonDInterval::ccPlausFn, NonDInterval::ccPlausVal, NonD::cdfFlag, NonDInterval::cellBPA, NonDInterval::cellFnLowerBounds, NonDInterval::cellFnUpperBounds, NonD::computedGenRelLevels, NonD::computedProbLevels, NonD::computedRespLevels, NonD::finalStatistics, Response::function_values(), Iterator::iteratedModel, NonDInterval::numCells, Iterator::numFunctions, NonD::requestedGenRelLevels, NonD::requestedProbLevels, NonD::requestedRespLevels, NonD::respLevelTarget, Model::response_labels(), NonDInterval::singleIntervalFlag, and Dakota::write_precision.

The documentation for this class was generated from the following files:

- NonDInterval.H
- NonDInterval.C

43.84 NonDLHSEvidence Class Reference

Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ. Inheritance diagram for NonDLHSEvidence::

Public Member Functions

- NonDLHSEvidence (Model &model)

  constructor

- ~NonDLHSEvidence ()

  destructor

- void initialize ()

  perform any required initialization

- void post_process_samples ()

  post-process the output from executing lhsSampler

43.84.1 Detailed Description

Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ. The NonDEvidence class implements the propagation of epistemic uncertainty using Dempster-Shafer theory of evidence. In this approach, one assigns a set of basic probability assignments (BPA) to intervals defined for the uncertain variables. Input interval combinations are calculated, along with their BPA. Currently, the response function is evaluated at a set of sample points, then a response surface is constructed which is sampled extensively to find the minimum and maximum within each input interval cell, corresponding to the belief and plausibility within that cell, respectively. This data is then aggregated to calculate cumulative distribution functions for belief and plausibility.

The documentation for this class was generated from the following files:

- NonDLHSEvidence.H
- NonDLHSEvidence.C

43.85 NonDLHSInterval Class Reference

Class for the LHS-based interval methods within DAKOTA/UQ. Inheritance diagram for NonDLHSInterval::

Public Member Functions

- NonDLHSInterval (Model &model)

  constructor

- ~NonDLHSInterval ()

  destructor

- void quantify_uncertainty ()

  performs an epistemic uncertainty propagation using LHS samples

Protected Member Functions

- virtual void initialize ()

  perform any required initialization

- virtual void post_process_samples ()=0

  post-process the output from executing lhsSampler

Protected Attributes

- Iterator lhsSampler

  the LHS sampler instance

- const int seedSpec

  the user seed specification (default is 0)

- int numSamples

  the number of samples used

- String rngName

  name of the random number generator

43.85.1 Detailed Description

Class for the LHS-based interval methods within DAKOTA/UQ. The NonDLHSInterval class implements the propagation of epistemic uncertainty using LHS-based methods.

The documentation for this class was generated from the following files:

- NonDLHSInterval.H
- NonDLHSInterval.C

43.86 NonDLHSSampling Class Reference

Performs LHS and Monte Carlo sampling for uncertainty quantification. Inheritance diagram for NonDLHSSampling::

Public Member Functions

- NonDLHSSampling (Model &model)

  standard constructor

- NonDLHSSampling (Model &model, const String &sample_type, int samples, int seed, const String &rng, short sampling_vars_mode=ACTIVE)

  alternate constructor for sample generation and evaluation "on the fly"

- NonDLHSSampling (const String &sample_type, int samples, int seed, const String &rng, const RealVector &lower_bnds, const RealVector &upper_bnds)

  alternate constructor for sample generation "on the fly"

- ~NonDLHSSampling ()

  destructor

Protected Member Functions

- void pre_run ()

  generate LHS samples in non-VBD cases

- void post_input ()

  read tabular data for post-run mode

- void quantify_uncertainty ()

  perform the evaluate parameter sets portion of run

- void post_run (std::ostream &s)

  generate statistics for LHS runs in non-VBD cases

- void print_results (std::ostream &s)

  print the final statistics

Private Attributes

- size_t numResponseFunctions

  number of response functions; used to distinguish NonD from opt/NLS usage

- bool varBasedDecompFlag

  flags computation of variance-based decomposition indices

43.86.1 Detailed Description

Performs LHS and Monte Carlo sampling for uncertainty quantification. The Latin Hypercube Sampling (LHS) package from Sandia Albuquerque's Risk and Reliability organization provides comprehensive capabilities for Monte Carlo and Latin Hypercube sampling within a broad array of user-specified probabilistic parameter distributions. It enforces user-specified rank correlations through use of a mixing routine. The NonDLHSSampling class provides a C++ wrapper for the LHS library and is used for performing forward propagations of parameter uncertainties into response statistics.

43.86.2 Constructor & Destructor Documentation

43.86.2.1 NonDLHSSampling (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

References NonDSampling::samplingVarsMode.

43.86.2.2 NonDLHSSampling (Model & *model*, const String & *sample_type*, int *samples*, int *seed*, const String & *rng*, short *sampling_vars_mode* = `ACTIVE`)

alternate constructor for sample generation and evaluation "on the fly" This alternate constructor is used for generation and evaluation of Model-based sample sets. A set_db_list_nodes has not been performed so required data must be passed through the constructor. It's purpose is to avoid the need for a separate LHS specification within methods that use LHS sampling.

References NonDSampling::samplingVarsMode.

43.86.2.3 NonDLHSSampling (const String & *sample_type*, int *samples*, int *seed*, const String & *rng*, const RealVector & *lower_bnds*, const RealVector & *upper_bnds*)

alternate constructor for sample generation "on the fly" This alternate constructor is used by ConcurrentStrategy for generation of uniform, uncorrelated sample sets. It is _not_ a letter-envelope instantiation and a set_db_list_-nodes has not been performed. It is called with all needed data passed through the constructor and is designed to allow more flexibility in variables set definition (i.e., relax connection to a variables specification and allow sampling over parameter sets such as multiobjective weights). In this case, a Model is not used and the object must only be used for sample generation (no evaluation).

References NonDSampling::get_parameter_sets(), and NonDSampling::samplingVarsMode.

43.86.3 Member Function Documentation

43.86.3.1 void quantify_uncertainty () `[protected, virtual]`

perform the evaluate parameter sets portion of run Loop over the set of samples and compute responses. Compute statistics on the set of responses if statsFlag is set.

Implements NonD.

References NonDSampling::allDataFlag, Analyzer::evaluate_parameter_sets(), Iterator::iteratedModel, Iterator::numContinuousVars, Iterator::numDiscreteIntVars, Iterator::numDiscreteRealVars, NonDLHSSampling::numResponseFunctions, NonDSampling::numSamples, NonDSampling::statsFlag, NonDLHSSampling::varBasedDecompFlag, and Analyzer::variance_based_decomp().

The documentation for this class was generated from the following files:

- NonDLHSSampling.H
- NonDLHSSampling.C

43.87 NonDLHSSingleInterval Class Reference

Class for pure interval propagation using LHS. Inheritance diagram for NonDLHSSingleInterval::

Public Member Functions

- NonDLHSSingleInterval (Model &model)

  constructor

- ~NonDLHSSingleInterval ()

  destructor

Protected Member Functions

- void initialize ()

  perform any required initialization

- void post_process_samples ()

  post-process the output from executing lhsSampler

Private Attributes

- size_t statCntr

  counter for finalStatistics

43.87.1 Detailed Description

Class for pure interval propagation using LHS. The NonDSingleInterval class implements the propagation of epistemic uncertainty using ...

The documentation for this class was generated from the following files:

- NonDLHSSingleInterval.H
- NonDLHSSingleInterval.C

43.88 NonDLocalEvidence Class Reference

Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ. Inheritance diagram for NonDLocalEvidence::

Public Member Functions

- NonDLocalEvidence (Model &model)

  constructor

- ~NonDLocalEvidence ()

  destructor

Protected Member Functions

- void initialize ()

  perform any required initialization

- void set_cell_bounds ()

  set the optimization variable bounds for each cell

- void truncate_to_cell_bounds (RealVector &initial_pt)

  truncate initial_pt to respect current cell lower/upper bounds

- void post_process_cell_results (bool minimize)

  post-process a cell minimization/maximization result

- void post_process_response_fn_results ()

  post-process the interval computed for a response function

- void post_process_final_results ()

  perform final post-processing

43.88.1 Detailed Description

Class for the Dempster-Shafer Evidence Theory methods within DAKOTA/UQ. The NonDEvidence class implements the propagation of epistemic uncertainty using Dempster-Shafer theory of evidence. In this approach, one assigns a set of basic probability assignments (BPA) to intervals defined for the uncertain variables. Input interval combinations are calculated, along with their BPA. Currently, the response function is evaluated at a set of sample points, then a response surface is constructed which is sampled extensively to find the minimum and maximum within each input interval cell, corresponding to the belief and plausibility within that cell, respectively. This data is then aggregated to calculate cumulative distribution functions for belief and plausibility.

The documentation for this class was generated from the following files:

- NonDLocalEvidence.H
- NonDLocalEvidence.C

43.89 NonDLocalInterval Class Reference

Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. Inheritance diagram for NonDLocalInterval::

Public Member Functions

- NonDLocalInterval (Model &model)

  constructor

- ~NonDLocalInterval ()

  destructor

- void quantify_uncertainty ()

  Performs an optimization to determine interval bounds for an entire function or interval bounds on a particular statistical estimator.

- String uses_method () const

  return name of active optimizer method

- void method_recourse ()

  perform an MPP optimizer method switch due to a detected conflict

Protected Member Functions

- virtual void initialize ()

  perform any required initialization

- virtual void set_cell_bounds ()

set the optimization variable bounds for each cell

- virtual void truncate_to_cell_bounds (RealVector &initial_pt)

  truncate initial_pt to respect current cell lower/upper bounds

- virtual void post_process_cell_results (bool minimize)

  post-process a cell minimization/maximization result

- virtual void post_process_response_fn_results ()

  post-process the interval computed for a response function

- virtual void post_process_final_results ()

  perform final post-processing

Protected Attributes

- Iterator minMaxOptimizer

  local gradient-based optimizer

- Model minMaxModel

  recast model with sign flip for maximizing

Static Private Member Functions

- static void objective_min (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in minimizing for the interval lower bound

- static void objective_max (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in maximizing for the interval upper bound

Private Attributes

- bool npsolFlag

  flag representing the gradient-based optimization algorithm selection (NPSOL SQP or OPT++ NIP)

Static Private Attributes

- static NonDLocalInterval * nondLIInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.89.1 Detailed Description

Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. The NonDLocalInterval class supports local gradient-based optimization apporaches to determining interval bounds for epistemic UQ. The interval bounds may be on the entire function in the case of pure interval analysis (e.g. intervals on input = intervals on output), or the intervals may be on statistics of an "inner loop" aleatory analysis such as intervals on means, variances, or percentile levels.

The documentation for this class was generated from the following files:

- NonDLocalInterval.H
- NonDLocalInterval.C

43.90 NonDLocalReliability Class Reference

Class for the reliability methods within DAKOTA/UQ. Inheritance diagram for NonDLocalReliability::

Public Member Functions

- NonDLocalReliability (Model &model)

  constructor

- ~NonDLocalReliability ()

  destructor

- void quantify_uncertainty ()

  performs an uncertainty propagation using analytical reliability methods which solve constrained optimization problems to obtain approximations of the cumulative distribution function of response

- void print_results (std::ostream &s)

  print the approximate mean, standard deviation, and importance factors when using the mean value method or the CDF/CCDF information when using MPP-search-based reliability methods

- String uses_method () const

  return name of active MPP optimizer

- void method_recourse ()

  perform an MPP optimizer method switch due to a detected conflict

Private Member Functions

- void initial_taylor_series ()

  convenience function for performing the initial limit state Taylor-series approximation

- void mean_value ()

  convenience function for encapsulating the simple Mean Value computation of approximate statistics and importance factors

- void mpp_search ()

  convenience function for encapsulating the reliability methods that employ a search for the most probable point (AMV, AMV+, FORM, SORM)

- void initialize_class_data ()

  convenience function for initializing class scope arrays

- void initialize_level_data ()

  convenience function for initializing/warm starting MPP search data for each response function prior to level 0

- void initialize_mpp_search_data ()

  convenience function for initializing/warm starting MPP search data for each z/p/beta level for each response function

- void update_mpp_search_data (const Variables &vars_star, const Response &resp_star)

  convenience function for updating MPP search data for each z/p/beta level for each response function

- void update_level_data ()

  convenience function for updating z/p/beta level data and final statistics following MPP convergence

- void update_pma_reliability_level ()

  update requestedCDFRelLevel from prescribed probabilities or prescribed generalized reliabilities by inverting second-order integrations

- void update_limit_state_surrogate ()

  convenience function for passing the latest variables/response data to the data fit embedded within uSpaceModel

- void assign_mean_data ()

  update mostProbPointX/U, computedRespLevel, fnGradX/U, and fnHessX/U from ranVarMeansX/U, fnValsMeanX, fnGradsMeanX, and fnHessiansMeanX

- void dg_ds_eval (const RealVector &x_vars, const RealVector &fn_grad_x, RealVector &final_stat_grad)

  convenience function for evaluating dg/ds

- Real probability (const Real &beta, bool cdf_flag)

  Convert beta to a probability using either a first-order or second-order integration.

- Real reliability (const Real &p, bool cdf_flag)

  Convert probability to beta using the inverse of a first-order or second-order integration.

- bool reliability_residual (const Real &p, const Real &beta, const RealVector &kappa, Real &res)

  compute the residual for inversion of second-order probability corrections using Newton's method (called by reliability(p))

- Real reliability_residual_derivative (const Real &p, const Real &beta, const RealVector &kappa)

  compute the residual derivative for inversion of second-order probability corrections using Newton's method (called by reliability(p))

- void principal_curvatures ()

  Compute the kappaU vector of principal curvatures from fnHessU.

Private Attributes

- RealVector meanStats

  means of response functions (calculated in initial_taylor_series())

- RealVector stdDevStats

  std deviations of response functions (calculated in initial_taylor_series())

- RealVector fnGradX

  actual x-space gradient for current function from most recent response evaluation

- RealVector fnGradU

  u-space gradient for current function updated from fnGradX and Jacobian dx/du

- RealSymMatrix fnHessX

  actual x-space Hessian for current function from most recent response evaluation

- RealSymMatrix fnHessU

  u-space Hessian for current function updated from fnHessX and Jacobian dx/du

- RealVector kappaU

  principal curvatures derived from eigenvalues of orthonormal transformation of fnHessU

- RealVector fnValsMeanX

  response function values evaluated at mean x

- RealMatrix fnGradsMeanX

  response function gradients evaluated at mean x

- RealSymMatrixArray fnHessiansMeanX

  response function Hessians evaluated at mean x

- RealVector ranVarMeansU

  vector of means for all uncertain random variables in u-space

- RealVector initialPtU

  initial guess for MPP search in u-space

- RealVector mostProbPointX

  location of MPP in x-space

- RealVector mostProbPointU

  location of MPP in u-space

- RealVectorArray prevMPPULev0

  array of converged MPP's in u-space for level 0. Used for warm-starting initialPtU within RBDO.

- RealMatrix prevFnGradDLev0

  matrix of limit state sensitivities w.r.t. inactive/design variables for level 0. Used for warm-starting initialPtU within RBDO.

- RealMatrix prevFnGradULev0

  matrix of limit state sensitivities w.r.t. active/uncertain variables for level 0. Used for warm-starting initialPtU within RBDO.

- RealVector prevICVars

  previous design vector. Used for warm-starting initialPtU within RBDO.

- ShortArray prevCumASVLev0

  accumulation (using |=) of all previous design ASV's from requested finalStatistics. Used to detect availability of prevFnGradDLev0 data for warm-starting initialPtU within RBDO.

- bool npsolFlag

  flag representing the optimization MPP search algorithm selection (NPSOL SQP or OPT++ NIP)

- bool warmStartFlag

  flag indicating the use of warm starts

- bool nipModeOverrideFlag

  flag indicating the use of move overrides within OPT++ NIP

- bool curvatureDataAvailable

  flag indicating that sufficient data (i.e., fnGradU, fnHessU, mostProbPointU) is available for computing principal curvatures

- short integrationOrder

  integration order (1 or 2) provided by `integration` *specification*

- short secondOrderIntType

  type of second-order integration: Breitung, Hohenbichler-Rackwitz, or Hong

- Real curvatureThresh

  cut-off value for 1/sqrt() term in second-order probability corrections.

- short taylorOrder

  order of Taylor series approximations (1 or 2) in MV/AMV/AMV+ derived from hessianType

- RealMatrix impFactor

  importance factors predicted by MV

- int npsolDerivLevel

  derivative level for NPSOL executions (1 = analytic grads of objective fn, 2 = analytic grads of constraints, 3 = analytic grads of both).

- unsigned short warningBits

  set of warnings accumulated during execution

43.90.1 Detailed Description

Class for the reliability methods within DAKOTA/UQ. The NonDLocalReliability class implements the following reliability methods through the support of different limit state approximation and integration options: mean value (MVFOSM/MVSOSM), advanced mean value method (AMV, AMV^2) in x- or u-space, iterated advanced mean value method (AMV+, AMV^2+) in x- or u-space, two-point adaptive nonlinearity approximation (TANA) in x- or u-space, first order reliability method (FORM), and second order reliability method (SORM). All options except mean value employ an optimizer (currently NPSOL SQP or OPT++ NIP) to solve an equality-constrained optimization problem for the most probable point (MPP). The MPP search may be formulated as the reliability index approach (RIA) for mapping response levels to reliabilities/probabilities or as the performance measure approach (PMA) for performing the inverse mapping of reliability/probability levels to response levels.

43.90.2 Member Function Documentation

43.90.2.1 void initial_taylor_series () `[private]`

convenience function for performing the initial limit state Taylor-series approximation An initial first- or second-order Taylor-series approximation is required for MV/AMV/AMV+/TANA or for the case where meanStats or stdDevStats (from MV) are required within finalStatistics for subIterator usage of NonDLocalReliability.

References Response::active_set_request_vector(), Iterator::activeSet, Model::component_parallel_mode(), Model::compute_response(), Model::continuous_variables(), Dakota::copy_data(), Model::current_response(), NonD::finalStatistics, NonDLocalReliability::fnGradsMeanX, NonDLocalReliability::fnHessiansMeanX, NonDLocalReliability::fnValsMeanX, Response::function_gradients(), Response::function_hessians(), Response::function_values(), Iterator::hessianType, Iterator::iteratedModel, NonDLocalReliability::meanStats, NonDReliability::mppSearchType, NonD::natafTransform, Iterator::numFunctions, NonD::numUncertainVars, ActiveSet::request_vector(), NonD::requestedGenRelLevels, NonD::requestedProbLevels, NonD::requestedRelLevels, NonD::requestedRespLevels, NonDLocalReliability::stdDevStats, Iterator::subIteratorFlag, NonDLocalReliability::taylorOrder, and NonDReliability::uSpaceModel.

Referenced by NonDLocalReliability::quantify_uncertainty().

43.90.2.2 void initialize_class_data () [private]

convenience function for initializing class scope arrays Initialize class-scope arrays and perform other start-up activities, such as evaluating median limit state responses.

References Response::active_set_derivative_vector(), NonD::finalStatistics, NonDReliability::importanceSampler, NonD::initialize_random_variables(), NonDReliability::integrationRefinement, Iterator::iterator_rep(), NonDReliability::mppModel, NonD::natafTransform, Iterator::numFunctions, NonDReliability::numRelAnalyses, NonD::numUncertainVars, NonDLocalReliability::prevCumASVLev0, NonDLocalReliability::prevFnGradDLev0, NonDLocalReliability::prevFnGradULev0, NonDLocalReliability::prevMPPULev0, NonDLocalReliability::ranVarMeansU, Iterator::subIteratorFlag, Model::update_from_subordinate_model(), and NonDLocalReliability::warmStartFlag.

Referenced by NonDLocalReliability::mpp_search().

43.90.2.3 void initialize_level_data () [private]

convenience function for initializing/warm starting MPP search data for each response function prior to level 0 For a particular response function prior to the first z/p/beta level, initialize/warm-start optimizer initial guess (initialPtU), expansion point (mostProbPointX/U), and associated response data (computedRespLevel, fnGradX/U, and fnHessX/U).

References Iterator::activeSet, NonDLocalReliability::assign_mean_data(), Model::component_parallel_mode(), Model::compute_response(), NonDReliability::computedRespLevel, Model::continuous_variable_ids(), Model::continuous_variables(), Dakota::copy_data(), Model::current_response(), NonDLocalReliability::curvatureDataAvailable, NonDLocalReliability::fnGradU, NonDLocalReliability::fnGradX, NonDLocalReliability::fnHessU, NonDLocalReliability::fnHessX, Response::function_gradient_copy(), Response::function_hessian(), Response::function_value(), Model::inactive_continuous_variables(), NonDLocalReliability::initialPtU, Iterator::iteratedModel, NonDLocalReliability::mostProbPointU, NonDLocalReliability::mostProbPointX, NonDReliability::mppSearchType, NonD::natafTransform, NonDReliability::numRelAnalyses, NonD::numUncertainVars, NonDLocalReliability::prevCumASVLev0, NonDLocalReliability::prevFnGradDLev0, NonDLocalReliability::prevFnGradULev0, NonDLocalReliability::prevICVars, NonDLocalReliability::prevMPPULev0, ActiveSet::request_value(), ActiveSet::request_values(), NonD::requestedRespLevels, NonDReliability::respFnCount, Iterator::subIteratorFlag, Model::surrogate_function_indices(), NonDLocalReliability::taylorOrder, NonDLocalReliability::update_limit_state_surrogate(), NonDReliability::uSpaceModel, and NonDLocalReliability::warmStartFlag.

Referenced by NonDLocalReliability::mpp_search().

43.90.2.4 void initialize_mpp_search_data () [private]

convenience function for initializing/warm starting MPP search data for each z/p/beta level for each response function For a particular response function at a particular z/p/beta level, warm-start or reset the optimizer initial guess (initialPtU), expansion point (mostProbPointX/U), and associated response data (computedRespLevel, fnGradX/U, and fnHessX/U).

References NonDLocalReliability::assign_mean_data(), NonD::cdfFlag, NonD::computedRelLevels, NonDLocalReliability::fnGradU, NonDLocalReliability::initialPtU, NonDReliability::levelCount, NonDLocalReliability::mostProbPointU, NonDReliability::mppSearchType, NonD::numUncertainVars, NonDReliability::requestedCDFRelLevel, NonDReliability::requestedRespLevel, NonD::requestedRespLevels, NonDReliability::respFnCount, and NonDLocalReliability::warmStartFlag.

Referenced by NonDLocalReliability::mpp_search().

43.90.2.5 void update_mpp_search_data (const Variables & *vars_star*, const Response & *resp_star*) [private]

convenience function for updating MPP search data for each z/p/beta level for each response function Includes case-specific logic for updating MPP search data for the AMV/AMV+/TANA/NO_APPROX methods.

References Response::active_set(), Response::active_set_request_vector(), Iterator::activeSet, NonDReliability::approxConverged, NonDReliability::approxIters, NonD::cdfFlag, Model::component_parallel_mode(), Model::compute_response(), NonDReliability::computedRelLevel, NonDReliability::computedRespLevel, Model::continuous_variable_ids(), Model::continuous_variables(), Variables::continuous_variables(), Iterator::convergenceTol, Variables::copy(), Dakota::copy_data(), Model::current_response(), Model::current_variables(), NonDLocalReliability::curvatureDataAvailable, Dakota::data_pairs, NonD::finalStatistics, NonDLocalReliability::fnGradU, NonDLocalReliability::fnGradX, NonDLocalReliability::fnHessU, NonDLocalReliability::fnHessX, Response::function_gradient_copy(), Response::function_hessian(), Response::function_hessians(), Response::function_value(), Response::function_values(), NonDLocalReliability::initialPtU, NonDLocalReliability::integrationOrder, Model::interface_id(), Iterator::iteratedModel, NonDReliability::levelCount, Dakota::lookup_by_val(), Iterator::maxIterations, NonDLocalReliability::mostProbPointU, NonDLocalReliability::mostProbPointX, NonDReliability::mppSearchType, NonD::natafTransform, Iterator::numFunctions, NonD::numNormalVars, NonD::numUncertainVars, NonDLocalReliability::reliability(), ActiveSet::request_value(), ActiveSet::request_values(), ActiveSet::request_vector(), NonDReliability::requestedCDFProbLevel, NonDReliability::requestedCDFRelLevel, NonD::requestedProbLevels, NonD::requestedRelLevels, NonDReliability::requestedRespLevel, NonD::requestedRespLevels, NonDReliability::respFnCount, NonDReliability::statCount, NonDLocalReliability::taylorOrder, NonDLocalReliability::update_limit_state_surrogate(), NonDReliability::uSpaceModel, NonDLocalReliability::warmStartFlag, and NonDLocalReliability::warningBits.

Referenced by NonDLocalReliability::mpp_search().

43.90.2.6 void update_level_data () [private]

convenience function for updating z/p/beta level data and final statistics following MPP convergence Updates computedRespLevels/computedProbLevels/computedRelLevels, finalStatistics, warm start, and graphics data.

References Dakota::abort_handler(), Response::active_set_derivative_vector(), Response::active_set_request_vector(), Graphics::add_datapoint(), NonD::cdfFlag, NonD::computedGenRelLevels, NonD::computedProbLevels, NonDReliability::computedRelLevel, NonD::computedRelLevels, NonDReliability::computedRespLevel, NonD::computedRespLevels, NonDLocalReliability::curvatureThresh, Dakota::dakota_graphics, NonDLocalReliability::dg_ds_eval(), NonD::finalStatistics, NonDLocalReliability::fnGradU, NonDLocalReliability::fnGradX, Response::function_gradient(), Response::function_value(), NonDLocalReliability::integrationOrder, NonDLocalReliability::kappaU, NonDReliability::levelCount, NonDLocalReliability::mostProbPointU, NonDLocalReliability::mostProbPointX, Graphics::new_dataset(), Iterator::numFunctions, NonD::numUncertainVars, NonDLocalReliability::prevCumASVLev0, NonDLocalReliability::prevFnGradDLev0, NonDLocalReliability::prevFnGradULev0, NonDLocalReliability::prevMPPULev0, NonDLocalReliability::probability(), NonD::requestedGenRelLevels, NonD::requestedProbLevels, NonD::requestedRelLevels, NonD::requestedRespLevels, NonDReliability::respFnCount, NonD::respLevelTarget, NonDLocalReliability::secondOrderIntType, NonDReliability::statCount, Iterator::subIteratorFlag, NonDLocalReliability::warmStartFlag, and NonDLocalReliability::warningBits.

Referenced by NonDLocalReliability::mpp_search().

43.90.2.7 void update_pma_reliability_level () [private, virtual]

update requestedCDFRelLevel from prescribed probabilities or prescribed generalized reliabilities by inverting second-order integrations For PMA SORM with prescribed p-level or prescribed generalized beta-level, requestedCDFRelLevel must be updated. This virtual function redefinition is called from NonDReliability::PMA_constraint_eval().

Reimplemented from NonDReliability.

References Variables::continuous_variables(), Dakota::copy_data(), Model::current_response(), Model::current_variables(), NonDLocalReliability::curvatureDataAvailable, NonDLocalReliability::fnGradU, NonDLocalReliability::fnHessU, Response::function_gradient_copy(), Response::function_hessian(), NonDLocalReliability::integrationOrder, NonDReliability::levelCount, NonDLocalReliability::mostProbPointU, NonDReliability::mppSearchType, NonDLocalReliability::reliability(), NonDReliability::requestedCDFProbLevel, NonDReliability::requestedCDFRelLevel, NonD::requestedProbLevels, NonD::requestedRelLevels, NonD::requestedRespLevels, NonDReliability::respFnCount, and NonDReliability::uSpaceModel.

43.90.2.8 void dg_ds_eval (const RealVector & *x_vars*, const RealVector & *fn_grad_x*, RealVector & *final_stat_grad*) [private]

convenience function for evaluating dg/ds Computes dg/ds where s = design variables. Supports potentially overlapping cases of design variable augmentation and insertion.

References Response::active_set_derivative_vector(), Iterator::activeSet, Model::all_continuous_variable_ids(), Model::component_parallel_mode(), Model::compute_response(), Dakota::contains(), Model::continuous_variable_ids(), Model::continuous_variables(), Dakota::copy_data(), Model::current_response(), ActiveSet::derivative_vector(), NonD::finalStatistics, Response::function_gradient_copy(), Response::function_gradients(), Model::inactive_continuous_variable_ids(), Iterator::iteratedModel, NonDReliability::mppSearchType, NonD::natafTransform, Iterator::primaryACVarMapIndices, ActiveSet::request_value(), ActiveSet::request_values(), NonDReliability::respFnCount, Iterator::secondaryACVarMapTargets, and NonDReliability::uSpaceModel.

Referenced by NonDLocalReliability::mean_value(), NonDLocalReliability::mpp_search(), and NonDLocalReliability::update_level_data().

43.90.2.9 Real probability (const Real & *beta*, bool *cdf_flag*) [private]

Convert beta to a probability using either a first-order or second-order integration. Converts beta into a probability using either first-order (FORM) or second-order (SORM) integration. The SORM calculation first calculates the principal curvatures at the MPP (using the approach in Ch. 8 of Haldar & Mahadevan), and then applies correction formulations from the literature (Breitung, Hohenbichler-Rackwitz, or Hong).

References NonDLocalReliability::curvatureDataAvailable, NonDLocalReliability::curvatureThresh, NonDAdaptImpSampling::get_probability(), NonDReliability::importanceSampler, NonDAdaptImpSampling::initialize(), NonDLocalReliability::integrationOrder, NonDReliability::integrationRefinement, Iterator::iterator_rep(), NonDLocalReliability::kappaU, NonDLocalReliability::mostProbPointU, NonD::numUncertainVars, NonDLocalReliability::principal_curvatures(), NonDReliabil-

ity::requestedRespLevel, NonDReliability::respFnCount, Iterator::run_iterator(), NonDLocalReliability::secondOrderIntType, and NonDLocalReliability::warningBits.

Referenced by NonDLocalReliability::mean_value(), and NonDLocalReliability::update_level_data().

43.90.2.10 Real reliability (const Real & *p*, bool *cdf_flag*) `[private]`

Convert probability to beta using the inverse of a first-order or second-order integration. Converts a probability into a reliability using the inverse of the first-order or second-order integrations implemented in NonDLocalReliability::probability().

References Iterator::convergenceTol, NonDLocalReliability::curvatureDataAvailable, NonDLocalReliability::integrationOrder, NonDLocalReliability::kappaU, NonDLocalReliability::principal_curvatures(), NonDLocalReliability::reliability_residual(), NonDLocalReliability::reliability_residual_derivative(), and NonDLocalReliability::warningBits.

Referenced by NonDLocalReliability::mean_value(), NonDLocalReliability::mpp_search(), NonDLocalReliability::update_mpp_search_data(), and NonDLocalReliability::update_pma_reliability_level().

The documentation for this class was generated from the following files:

- NonDLocalReliability.H
- NonDLocalReliability.C

43.91 NonDLocalSingleInterval Class Reference

Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. Inheritance diagram for NonDLocalSingleInterval::

Public Member Functions

- NonDLocalSingleInterval (Model &model)

  constructor

- ~NonDLocalSingleInterval ()

  destructor

Protected Member Functions

- void initialize ()

  perform any required initialization

- void post_process_cell_results (bool minimize)

  post-process a cell minimization/maximization result

Private Attributes

- size_t statCntr

  counter for finalStatistics

43.91.1 Detailed Description

Class for using local gradient-based optimization approaches to calculate interval bounds for epistemic uncertainty quantification. The NonDLocalSingleInterval class supports local gradient-based optimization apporaches to determining interval bounds for epistemic UQ. The interval bounds may be on the entire function in the case of pure interval analysis (e.g. intervals on input = intervals on output), or the intervals may be on statistics of an "inner loop" aleatory analysis such as intervals on means, variances, or percentile levels.

The documentation for this class was generated from the following files:

- NonDLocalSingleInterval.H
- NonDLocalSingleInterval.C

43.92 NonDPolynomialChaos Class Reference

Nonintrusive polynomial chaos expansion approaches to uncertainty quantification. Inheritance diagram for NonDPolynomialChaos::

Public Member Functions

- NonDPolynomialChaos (Model &model)

  standard constructor

- NonDPolynomialChaos (Model &model, short exp_coeffs_approach, unsigned short num_int_level, short u_space_type, bool piecewise_basis, bool use_derivs)

  alternate constructor

- ~NonDPolynomialChaos ()

  destructor

- void resolve_inputs (short &u_space_type, short &data_order)

  perform error checks and mode overrides

- void initialize_u_space_model ()

  initialize uSpaceModel polynomial approximations with PCE/SC data

- void compute_expansion ()

  form or import an orthogonal polynomial expansion using PCE methods

- void increment_expansion ()

  uniformly increment the order of the polynomial chaos expansion

- void print_coefficients (std::ostream &s)

  print the PCE coefficient array for the orthogonal basis

Private Member Functions

- int terms_ratio_to_samples (size_t num_exp_terms, Real colloc_ratio, Real terms_order)

  convert number of expansion terms and collocation ratio to a number of collocation samples

- Real terms_samples_to_ratio (size_t num_exp_terms, int samples, Real terms_order)

  convert number of expansion terms and number of collocation samples to a collocation ratio

Private Attributes

- String expansionImportFile

  filename for import of chaos coefficients

- int expansionTerms

  user specification of PCE terms

- Real collocRatio

  factor applied to expansionTerms^termsOrder in computing number of regression points, either user specified or inferred

- Real termsOrder

  exponent applied to number of expansion terms for computing number of regression points

- bool tensorRegression

  option for regression PCE using a filtered set tensor-product points

- RealMatrix pceGradsMeanX

  derivative of the PCE with respect to the x-space variables evaluated at the means (used as uncertainty importance metrics)

43.92.1 Detailed Description

Nonintrusive polynomial chaos expansion approaches to uncertainty quantification. The NonDPolynomialChaos class uses a polynomial chaos expansion (PCE) approach to approximate the effect of parameter uncertainties on response functions of interest. It utilizes the OrthogPolyApproximation class to manage multiple types of orthogonal polynomials within a Wiener-Askey scheme to PCE. It supports PCE coefficient estimation via sampling, quadrature, point-collocation, and file import.

43.92.2 Constructor & Destructor Documentation

43.92.2.1 NonDPolynomialChaos (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation using the ProblemDescDB.

References Dakota::abort_handler(), Model::assign_rep(), NonDPolynomialChaos::collocRatio, NonDExpansion::construct_cubature(), NonDExpansion::construct_expansion_sampler(), NonD::construct_lhs(), NonDExpansion::construct_quadrature(), NonDExpansion::construct_sparse_grid(), NonD::construct_u_space_model(), Model::derivative_concurrency(), NonDExpansion::expansionCoeffsApproach, NonDPolynomialChaos::expansionImportFile, NonDPolynomialChaos::expansionTerms, ProblemDescDB::get_dusa(), ProblemDescDB::get_int(), ProblemDescDB::get_rdv(), ProblemDescDB::get_real(), ProblemDescDB::get_short(), ProblemDescDB::get_string(), ProblemDescDB::get_ushort(), Model::init_communicators(), NonDExpansion::initialize(), NonDPolynomialChaos::initialize_u_space_model(), Iterator::iteratedModel, Iterator::iterator_rep(), Iterator::maxConcurrency, NonD::numContDesVars, NonD::numContEpistUncVars, Iterator::numContinuousVars, NonD::numContStateVars, NonDExpansion::numSamplesOnExpansion, NonDExpansion::numSamplesOnModel, Iterator::outputLevel, NonDExpansion::piecewiseBasis, Iterator::probDescDB, NonDExpansion::refineControl, NonDExpansion::refineType, NonDPolynomialChaos::resolve_inputs(), NonDPolynomialChaos::tensorRegression, NonDPolynomialChaos::terms_ratio_to_samples(), NonDPolynomialChaos::terms_samples_to_ratio(), NonDPolynomialChaos::termsOrder, NonDExpansion::uSpaceModel, and Analyzer::vary_pattern().

43.92.2.2 NonDPolynomialChaos (Model & *model*, short *exp_coeffs_approach*, unsigned short *num_int_level*, short *u_space_type*, bool *piecewise_basis*, bool *use_derivs*)

alternate constructor This constructor is used for helper iterator instantiation on the fly.

References Model::assign_rep(), NonDExpansion::construct_cubature(), NonDExpansion::construct_quadrature(), NonDExpansion::construct_sparse_grid(), NonD::construct_u_space_model(), NonDExpansion::expansionCoeffsApproach, NonDExpansion::initialize(), NonDPolynomialChaos::initialize_u_space_model(), Iterator::iteratedModel, NonD::numContDesVars, NonD::numContEpistUncVars, NonD::numContStateVars, Iterator::outputLevel, NonDExpansion::piecewiseBasis, NonDPolynomialChaos::resolve_inputs(), and NonDExpansion::uSpaceModel.

43.92.3 Member Function Documentation

43.92.3.1 void increment_expansion () `[virtual]`

uniformly increment the order of the polynomial chaos expansion Used for uniform refinement of regression-based PCE.

Reimplemented from NonDExpansion.

References Model::approximations(), NonDPolynomialChaos::collocRatio, NonDQuadrature::compute_minimum_quadrature_order(), PecosApproximation::expansion_terms(), NonDQuadrature::filtered_samples(), PecosApproximation::increment_order(), Iterator::iterator_rep(), Model::model_rep(), Iterator::numFunctions, NonDExpansion::numSamplesOnModel, NonDSampling::sampling_reference(), Model::subordinate_iterator(), NonDPolynomialChaos::tensorRegression, NonDPolynomialChaos::terms_ratio_to_samples(), NonDPolynomialChaos::termsOrder, DataFitSurrModel::total_points(), and NonDExpansion::uSpaceModel.

The documentation for this class was generated from the following files:

- NonDPolynomialChaos.H
- NonDPolynomialChaos.C

43.93 NonDQuadrature Class Reference

Derived nondeterministic class that generates N-dimensional numerical quadrature points for evaluation of expectation integrals over uncorrelated standard normals/uniforms/exponentials/betas/gammas. Inheritance diagram for NonDQuadrature::

Public Member Functions

- NonDQuadrature (Model &model, const UShortArray &quad_order, const RealVector &dim_pref)

  alternate constructor for instantiations "on the fly" based on a quadrature order specification

- NonDQuadrature (Model &model, int num_filt_samples, const RealVector &dim_pref)

  alternate constructor for instantiations "on the fly" based on the size of a filtered tensor product sample set

- const Pecos::UShortArray & quadrature_order () const

  return Pecos::TensorProductDriver::quadOrder

- void filtered_samples (size_t samples)

  set numFilteredSamples

- void compute_minimum_quadrature_order ()

  calculate smallest dimQuadOrderRef with at least numFilteredSamples

- void compute_minimum_quadrature_order (size_t min_samples, const RealVector &dim_pref, UShortArray &dim_quad_order)

  calculate smallest dim_quad_order with at least min_samples

Protected Member Functions

- NonDQuadrature (Model &model)

  constructor

- ~NonDQuadrature ()

  destructor

- void initialize_grid (const std::vector< Pecos::BasisPolynomial > &poly_basis)
- void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

- void reset ()

  restore initial state for repeated sub-iterator executions

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)
- void increment_grid ()

  increment SSG level/TPQ order

- void increment_grid_preference (const RealVector &dim_pref)

  increment SSG level/TPQ order and update anisotropy

- void increment_refinement_sequence ()

  increment sequenceIndex and update active orders/levels

- int num_samples () const

  get the current number of samples

Private Member Functions

- void increment_grid (UShortArray &dim_quad_order)

  convenience function used to make increment_grid() more modular

- void increment_grid_preference (const RealVector &dim_pref, UShortArray &dim_quad_order)

  convenience function used to make increment_grid_preference() more modular

- void filter_parameter_sets ()

  prune allSamples back to size numFilteredSamples, retaining points with highest product weight

- void anisotropic_preference (unsigned short quad_order_spec, const RealVector &dim_pref_spec, UShortArray &quad_order)

  initialize quad_order vector based on quad_order_spec scalar and dim_pref_spec vector

- void anisotropic_preference (const RealVector &dim_pref, UShortArray &quad_order_ref)

  update quad_order_ref based on an updated dimension preference, enforcing previous values as a lower bound

- void initialize_dimension_quadrature_order (unsigned short quad_order_spec, const RealVector &dim_pref_spec, UShortArray &dim_quad_order)

  initialize dim_quad_order from quad_order_spec and dim_pref_spec

- void nested_quadrature_order (const UShortArray &quad_order_ref)

  update Pecos::TensorProductDriver::quadOrder from dimQuadOrderRef to account for nested rule constraints

- void increment_dimension_quadrature_order (UShortArray &dim_quad_order)

  increment each dim_quad_order entry by 1

- void increment_dimension_quadrature_order (const RealVector &dim_pref, UShortArray &dim_quad_order)

  increment the dim_quad_order entry with maximum preference by 1 and then rebalance

Private Attributes

- Pecos::TensorProductDriver * tpqDriver

  convenience pointer to the numIntDriver representation

- bool nestedRules

  for studies involving refinement strategies, allow for use of nested quadrature rules such as Gauss-Patterson

- UShortArray quadOrderSpec

  a sequence of scalar quadrature orders, one per refinement level

- UShortArray dimQuadOrderRef

  reference point for Pecos::TensorProductDriver::quadOrder: the original user specification for the number of Gauss points per dimension, plus any refinements posted by increment_grid()

- size_t numFilteredSamples

  size of a filtered set of tensor quadrature points; used by the regression PCE approach known as "probabilistic collocation"

43.93.1 Detailed Description

Derived nondeterministic class that generates N-dimensional numerical quadrature points for evaluation of expectation integrals over uncorrelated standard normals/uniforms/exponentials/betas/gammas. This class is used by NonDPolynomialChaos, but could also be used for general numerical integration of moments. It employs Gauss-Hermite, Gauss-Legendre, Gauss-Laguerre, Gauss-Jacobi and generalized Gauss-Laguerre quadrature for use with normal, uniform, exponential, beta, and gamma density functions and integration bounds. The abscissas and weights for one-dimensional integration are extracted from the appropriate OrthogonalPolynomial class and are extended to n-dimensions using a tensor product approach.

43.93.2 Constructor & Destructor Documentation

43.93.2.1 NonDQuadrature (Model & *model*, const UShortArray & *quad_order*, const RealVector & *dim_pref*)

alternate constructor for instantiations "on the fly" based on a quadrature order specification This alternate constructor is used for on-the-fly generation and evaluation of numerical quadrature points.

References NonDIntegration::numIntDriver, and NonDQuadrature::tpqDriver.

43.93.2.2 NonDQuadrature (Model & *model*, int *num_filt_samples*, const RealVector & *dim_pref*)

alternate constructor for instantiations "on the fly" based on the size of a filtered tensor product sample set This alternate constructor is used for on-the-fly generation and evaluation of numerical quadrature points.

References NonDIntegration::numIntDriver, and NonDQuadrature::tpqDriver.

43.93.2.3 NonDQuadrature (Model & *model*) `[protected]`

constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_-list_nodes has been called and probDescDB can be queried for settings from the method specification. It is not currently used, as there is not yet a separate nond_quadrature method specification.

References NonDIntegration::check_variables(), ProblemDescDB::get_bool(), ProblemDescDB::get_-short(), Iterator::maxConcurrency, NonD::natafTransform, NonDQuadrature::nestedRules, NonDIntegration::numIntDriver, Iterator::probDescDB, NonDQuadrature::reset(), and NonDQuadrature::tpqDriver.

43.93.3 Member Function Documentation

43.93.3.1 void initialize_grid (const std::vector< Pecos::BasisPolynomial > & *poly_basis*) `[protected, virtual]`

Used in combination with alternate NonDQuadrature constructor.

Implements NonDIntegration.

References NonDQuadrature::compute_minimum_quadrature_order(), Iterator::maxConcurrency, NonDQuadrature::nestedRules, Iterator::numContinuousVars, NonDQuadrature::numFilteredSamples, NonDQuadrature::reset(), and NonDQuadrature::tpqDriver.

43.93.3.2 void sampling_reset (int *min_samples*, bool *all_data_flag*, bool *stats_flag*) `[protected, virtual]`

used by DataFitSurrModel::build_global() to publish the minimum number of points needed from the quadrature routine in order to build a particular global approximation.

Reimplemented from Iterator.

References NonDQuadrature::compute_minimum_quadrature_order(), NonDIntegration::dimPrefSpec,

NonDQuadrature::dimQuadOrderRef, NonDQuadrature::nested_quadrature_order(), NonDQuadrature::nestedRules, Iterator::numContinuousVars, and NonDQuadrature::tpqDriver.

43.93.3.3 int num_samples () const [inline, protected, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References NonDQuadrature::numFilteredSamples, and NonDQuadrature::tpqDriver.

43.93.3.4 void anisotropic_preference (unsigned short *quad_order_spec*, const RealVector & *dim_pref_spec*, UShortArray & *dim_quad_order*) [private]

initialize quad_order vector based on quad_order_spec scalar and dim_pref_spec vector This version of anisotropic_preference() converts a scalar quad_order_spec and a dim_pref vector into a quad_order vector. It is used for initialization and does not enforce a reference lower bound.

References Iterator::numContinuousVars.

Referenced by NonDQuadrature::increment_dimension_quadrature_order(), and NonDQuadrature::initialize_dimension_quadrature_order().

The documentation for this class was generated from the following files:

- NonDQuadrature.H
- NonDQuadrature.C

43.94 NonDQUESOBayesCalibration Class Reference

Bayesian inference using the QUESO library from UT Austin. Inheritance diagram for NonDQUESOBayesCalibration::

Public Member Functions

- NonDQUESOBayesCalibration (Model &model)

  standard constructor

- ~NonDQUESOBayesCalibration ()

  destructor

Public Attributes

- String rejectionType

  Rejection type (standard or delayed, in the DRAM framework).

- String metropolisType

  Metropolis type (hastings or adaptive, in the DRAM framework).

- int numSamples

  number of samples in the chain (e.g. number of MCMC samples)

- Real proposalCovScale

  scale factor for proposal covariance

- Real likelihoodScale

  scale factor for likelihood

Protected Member Functions

- void quantify_uncertainty ()

  redefined from DakotaNonD

Static Protected Member Functions

- static double dakotaLikelihoodRoutine (const uqGslVectorClass &paramValues, const uqGslVectorClass ∗paramDirection, const void ∗functionDataPtr, uqGslVectorClass ∗gradVector, uqGslMatrixClass ∗hessianMatrix, uqGslVectorClass ∗hessianEffect)

  Likelihood function for call-back from QUESO to DAKOTA for evaluation.

Protected Attributes

- RealMatrix xObsData

  Matrix that holds the experimental realizations of state variables x.

- RealMatrix yObsData

  Matrix that holds the experimental realizations of responses y.

- RealMatrix yStdData

  Matrix that holds the experimental realizations of std deviations of responses y.

- int randomSeed

  random seed to pass to QUESO

Private Attributes

- short emulatorType

  the emulator type: NO_EMULATOR, GAUSSIAN_PROCESS, POLYNOMIAL_CHAOS, or STOCHASTIC_-COLLOCATION

Static Private Attributes

- static NonDQUESOBayesCalibration ∗ NonDQUESOInstance

  Pointer to current class instance for use in static callback functions.

43.94.1 Detailed Description

Bayesian inference using the QUESO library from UT Austin. This class provides a wrapper to the QUESO library developed as part of the Predictive Science Academic Alliance Program (PSAAP), specifically the PECOS (Predictive Engineering and Computational Sciences) Center at UT Austin. The name QUESO stands for Quantification of Uncertainty for Estimation, Simulation, and Optimization.

43.94.2 Constructor & Destructor Documentation

43.94.2.1 NonDQUESOBayesCalibration (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

43.94.3 Member Function Documentation

43.94.3.1 void quantify_uncertainty () `[protected, virtual]`

redefined from DakotaNonD Perform the uncertainty quantification

Reimplemented from NonDBayesCalibration.

References Dakota::abort_handler(), Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), NonDQUESOBayesCalibration::dakotaLikelihoodRoutine(), NonDBayesCalibration::emulatorModel, NonDQUESOBayesCalibration::emulatorType, NonDCalibration::expDataFileAnnotated, NonDCalibration::expDataFileName, NonDCalibration::expStdDeviations, Iterator::iterator_rep(), NonDQUESOBayesCalibration::metropolisType, NonDQUESOBayesCalibration::NonDQUESOInstance, Iterator::numContinuousVars, NonDCalibration::numExpConfigVars, NonDCalibration::numExperiments, NonDCalibration::numExpStdDeviationsRead, Iterator::numFunctions, NonDQUESOBayesCalibration::numSamples, NonDQUESOBayesCalibration::proposalCovScale, NonDQUESOBayesCalibration::randomSeed, Dakota::read_data_tabular(), NonDQUESOBayesCalibration::rejectionType, NonDBayesCalibration::standardizedSpace, NonDBayesCalibration::stochExpIterator, NonDQUESOBayesCalibration::xObsData, NonDQUESOBayesCalibration::yObsData, and NonDQUESOBayesCalibration::yStdData.

The documentation for this class was generated from the following files:

- NonDQUESOBayesCalibration.H
- NonDQUESOBayesCalibration.C

43.95 NonDReliability Class Reference

Base class for the reliability methods within DAKOTA/UQ. Inheritance diagram for NonDReliability::

Protected Member Functions

- NonDReliability (Model &model)

  constructor

- ~NonDReliability ()

  destructor

- void initialize_graphics (bool graph_2d, bool tabular_data, const String &tabular_file)

  initialize graphics customized for reliability methods

- const Model & algorithm_space_model () const

  return the result of any recasting or surrogate model recursion layered on top of iteratedModel by the derived Iterator ctor chain

- virtual void update_pma_reliability_level ()

  update requestedCDFRelLevel for use in PMA_constraint_eval()

Static Protected Member Functions

- static void RIA_objective_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in the Reliability Index Approach (RIA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the objective function of (norm u)^2.

- static void RIA_constraint_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

static function used as the constraint function in the Reliability Index Approach (RIA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the constraint of G(u) = response level.

- static void PMA_objective_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the objective function in the Performance Measure Approach (PMA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the objective function of G(u).

- static void PMA_constraint_eval (const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  static function used as the constraint function in the Performance Measure Approach (PMA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the constraint of (norm u)^2 = beta^2.

- static void PMA2_set_mapping (const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set)

  static function used to augment the sub-model ASV requests when a beta-bar constraint target update is required for second-order PMA

Protected Attributes

- Model uSpaceModel

  Model representing the limit state in u-space, after any recastings and data fits.

- Model mppModel

  RecastModel which formulates the optimization subproblem: RIA, PMA, EGO.

- Iterator mppOptimizer

  Iterator which optimizes the mppModel.

- short mppSearchType

  the MPP search type selection: MV, x/u-space AMV, x/u-space AMV+, x/u-space TANA, x/u-space EGO, or NO_-APPROX

- Iterator importanceSampler

  importance sampling instance used to compute/refine probabilities

- short integrationRefinement

  integration refinement type (NO_INT_REFINE, IS, AIS, or MMAIS) provided by `refinement` *specification*

- size_t numRelAnalyses

  number of invocations of quantify_uncertainty()

- size_t approxIters

number of approximation cycles for the current respFnCount/levelCount

- bool approxConverged

  indicates convergence of approximation-based iterations

- int respFnCount

  counter for which response function is being analyzed

- size_t levelCount

  counter for which response/probability level is being analyzed

- size_t statCount

  counter for which final statistic is being computed

- Real requestedRespLevel

  the response level target for the current response function

- Real requestedCDFProbLevel

  the CDF probability level target for the current response function

- Real requestedCDFRelLevel

  the CDF reliability level target for the current response function

- Real computedRespLevel

  output response level calculated

- Real computedRelLevel

  output reliability level calculated

Static Protected Attributes

- static NonDReliability * nondRelInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.95.1 Detailed Description

Base class for the reliability methods within DAKOTA/UQ. The NonDReliability class provides a base class for NonDLocalReliability, which implements traditional MPP-based reliability methods, and NonDGlobalReliability, which implements global limit state search using Gaussian process models in combination with multimodal importance sampling.

43.95.2 Member Function Documentation

43.95.2.1 void RIA_objective_eval (const Variables & *sub_model_vars*, const Variables & *recast_vars*, const Response & *sub_model_response*, Response & *recast_response*) [static, protected]

static function used as the objective function in the Reliability Index Approach (RIA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the objective function of (norm u)^2. This function recasts a G(u) response set (already transformed and approximated in other recursions) into an RIA objective function.

References Response::active_set_request_vector(), Variables::continuous_variables(), Response::function_gradient_view(), Response::function_hessian_view(), and Response::function_value().

Referenced by NonDLocalReliability::mpp_search().

43.95.2.2 void RIA_constraint_eval (const Variables & *sub_model_vars*, const Variables & *recast_vars*, const Response & *sub_model_response*, Response & *recast_response*) [static, protected]

static function used as the constraint function in the Reliability Index Approach (RIA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the constraint of G(u) = response level. This function recasts a G(u) response set (already transformed and approximated in other recursions) into an RIA equality constraint.

References Response::active_set_request_vector(), Response::function_gradient(), Response::function_gradient_view(), Response::function_hessian(), Response::function_value(), NonDReliability::nondRelInstance, NonDReliability::requestedRespLevel, and NonDReliability::respFnCount.

Referenced by NonDLocalReliability::mpp_search().

43.95.2.3 void PMA_objective_eval (const Variables & *sub_model_vars*, const Variables & *recast_vars*, const Response & *sub_model_response*, Response & *recast_response*) [static, protected]

static function used as the objective function in the Performance Measure Approach (PMA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the objective function of G(u). This function recasts a G(u) response set (already transformed and approximated in other recursions) into an PMA objective function.

References Response::active_set_request_vector(), Response::function_gradient(), Response::function_gradient_view(), Response::function_gradients(), Response::function_hessian(), Response::function_hessian_view(), Response::function_value(), NonDReliability::nondRelInstance, NonDReliability::requestedCDFRelLevel, and NonDReliability::respFnCount.

Referenced by NonDLocalReliability::mpp_search().

43.95.2.4 void PMA_constraint_eval (const Variables & *sub_model_vars*, const Variables & *recast_vars*, const Response & *sub_model_response*, Response & *recast_response*) `[static, protected]`

static function used as the constraint function in the Performance Measure Approach (PMA) problem formulation. This equality-constrained optimization problem performs the search for the most probable point (MPP) with the constraint of (norm u)^2 = beta^2. This function recasts a G(u) response set (already transformed and approximated in other recursions) into a PMA equality constraint.

References Response::active_set_request_vector(), Variables::continuous_variables(), Response::function_gradient_view(), Response::function_hessian_view(), Response::function_value(), NonDReliability::nondRelInstance, NonDReliability::requestedCDFRelLevel, and NonDReliability::update_pma_reliability_level().

Referenced by NonDLocalReliability::mpp_search().

The documentation for this class was generated from the following files:

- NonDReliability.H
- NonDReliability.C

43.96 NonDSampling Class Reference

Base class for common code between NonDLHSSampling, NonDIncremLHSSampling, and NonDAdaptImpSampling. Inheritance diagram for NonDSampling::

Public Member Functions

- void moments (const RealVector &means, const RealVector &std_devs)

  set meanStats and stdDevStats

- void compute_distribution_mappings (const IntResponseMap &samples)

  called by compute_statistics() to calculate CDF/CCDF mappings of z to p/beta and of p/beta to z

- void update_final_statistics ()

  update finalStatistics from minValues/maxValues, meanStats/stdDevStats, and computedProbLevels/computedRelLevels/computedRespLevels

- void print_pdf_mappings (std::ostream &s) const

  prints the PDFs computed in compute_statistics()

- void sampling_reference (int samples_ref)

  set samplesRef

Protected Member Functions

- NonDSampling (Model &model)

  constructor

- NonDSampling (NoDBBaseConstructor, Model &model, const String &sample_type, int samples, int seed, const String &rng)

  alternate constructor for sample generation and evaluation "on the fly"

- NonDSampling (NoDBBaseConstructor, const String &sample_type, int samples, int seed, const String &rng, const RealVector &lower_bnds, const RealVector &upper_bnds)

  alternate constructor for sample generation "on the fly"

- ~NonDSampling ()

  destructor

- int num_samples () const

  get the current number of samples

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)

  resets number of samples and sampling flags

- const String & sampling_scheme () const

  return sampleType: "lhs" or "random"

- void vary_pattern (bool pattern_flag)

  set varyPattern

- void get_parameter_sets (Model &model)

  Uses lhsDriver to generate a set of samples from the distributions/bounds defined in the incoming model.

- void get_parameter_sets (const RealVector &lower_bnds, const RealVector &upper_bnds)

  Uses lhsDriver to generate a set of uniform samples over lower_bnds/upper_bnds.

- void update_model_from_sample (Model &model, const Real *sample_vars)

  Override default update of continuous vars only.

- void initialize_lhs (bool write_message)

  increments numLHSRuns, sets random seed, and initializes lhsDriver

- void compute_statistics (const RealMatrix &vars_samples, const IntResponseMap &resp_samples)

  For the input sample set, computes mean, standard deviation, and probability/reliability/response levels (aleatory uncertainties) or intervals (epsitemic or mixed uncertainties).

- void compute_intervals (const IntResponseMap &samples)

  called by compute_statistics() to calculate min/max intervals

- void compute_moments (const IntResponseMap &samples)

  called by compute_statistics() to calculate means, std deviations, and confidence intervals

- void print_statistics (std::ostream &s) const

  prints the statistics computed in compute_statistics()

- void print_intervals (std::ostream &s) const

prints the intervals computed in compute_intervals()

- void print_moments (std::ostream &s) const

  prints the moments computed in compute_moments()

- void view_counts (const Model &model, size_t &cv_start, size_t &num_cv, size_t &div_start, size_t &num_div, size_t &drv_start, size_t &num_drv) const

  compute sampled subsets of all cv/div/drv from samplingVarsMode and model

Protected Attributes

- const int seedSpec

  the user seed specification (default is 0)

- int randomSeed

  the current seed

- const int samplesSpec

  initial specification of number of samples

- int samplesRef

  reference number of samples updated for refinement

- int numSamples

  the current number of samples to evaluate

- String rngName

  name of the random number generator

- String sampleType

  the sample type: random, lhs, or incremental_lhs

- Pecos::LHSDriver lhsDriver

  the C++ wrapper for the F90 LHS library

- bool statsFlag

  flags computation/output of statistics

- bool allDataFlag

  flags update of allResponses < (allVariables or allSamples already defined)

- short samplingVarsMode

  the sampling mode: ACTIVE, ACTIVE_UNIFORM, ALL, or ALL_UNIFORM

- short sampleRanksMode

mode for input/output of LHS sample ranks: IGNORE_RANKS, GET_RANKS, SET_RANKS, or SET_GET_RANKS

- bool varyPattern

  flag for generating a sequence of seed values within multiple get_parameter_sets() calls so that these executions (e.g., for SBO/SBNLS) are not repeated, but are still repeatable

- RealMatrix sampleRanks

  data structure to hold the sample ranks

- RealVector mean95CIDeltas

  Plus/minus deltas on response function means for 95% confidence intervals (calculated in compute_moments()).

- RealVector stdDev95CILowerBnds

  Lower bound for 95% confidence interval on std deviation (calculated in compute_moments()).

- RealVector stdDev95CIUpperBnds

  Upper bound for 95% confidence interval on std deviation (calculated in compute_moments()).

- SensAnalysisGlobal nonDSampCorr

  initialize statistical post processing

Private Attributes

- size_t numLHSRuns

  counter for number of executions of get_parameter_sets() for this object

- RealVector meanStats

  means of response functions (calculated in compute_moments())

- RealVector stdDevStats

  std deviations of response functions (calculated in compute_moments())

- RealVector skewnessStats

  skewness of response functions (calculated in compute_moments())

- RealVector kurtosisStats

  kurtosis of response functions (calculated in compute_moments())

- RealVector minValues

  Min values of response functions for epistemic calculations (calculated in compute_intervals()).

- RealVector maxValues

  Max values of response functions for epistemic calculations (calculated in compute_intervals()).

- RealVectorArray computedPDFAbscissas

sorted response PDF intervals bounds extracted from min/max sample and requested/computedRespLevels (vector lengths = num bins + 1)

- RealVectorArray computedPDFOrdinates

  response PDF densities computed from bin counts divided by (unequal) bin widths (vector lengths = num bins)

43.96.1 Detailed Description

Base class for common code between NonDLHSSampling, NonDIncremLHSSampling, and NonDAdaptImpSampling. This base class provides common code for sampling methods which employ the Latin Hypercube Sampling (LHS) package from Sandia Albuquerque's Risk and Reliability organization. NonDSampling now exclusively utilizes the 1998 Fortran 90 LHS version as documented in SAND98-0210, which was converted to a UNIX link library in 2001. The 1970's vintage LHS (that had been f2c'd and converted to incomplete classes) has been removed.

43.96.2 Constructor & Destructor Documentation

43.96.2.1 NonDSampling (Model & *model*) `[protected]`

constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

References Dakota::abort_handler(), NonD::initialize_final_statistics(), Iterator::maxConcurrency, NonD::numEpistemicUncVars, NonDSampling::numSamples, NonDSampling::sampleType, and NonD::totalLevelRequests.

43.96.2.2 NonDSampling (NoDBBaseConstructor, Model & *model*, const String & *sample_type*, int *samples*, int *seed*, const String & *rng*) `[protected]`

alternate constructor for sample generation and evaluation "on the fly" This alternate constructor is used for generation and evaluation of on-the-fly sample sets.

References Iterator::maxConcurrency, NonDSampling::numSamples, NonDSampling::sampleType, and Iterator::subIteratorFlag.

43.96.2.3 NonDSampling (NoDBBaseConstructor, const String & *sample_type*, int *samples*, int *seed*, const String & *rng*, const RealVector & *lower_bnds*, const RealVector & *upper_bnds*) `[protected]`

alternate constructor for sample generation "on the fly" This alternate constructor is used by ConcurrentStrategy for generation of uniform, uncorrelated sample sets.

References Iterator::maxConcurrency, NonDSampling::numSamples, NonDSampling::sampleType, and Iterator::subIteratorFlag.

43.96.3 Member Function Documentation

43.96.3.1 int num_samples () const [inline, protected, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References NonDSampling::numSamples.

Referenced by NonDAdaptImpSampling::generate_samples(), NonDAdaptImpSampling::select_init_rep_points(), and NonDAdaptImpSampling::select_rep_points().

43.96.3.2 void sampling_reset (int *min_samples*, bool *all_data_flag*, bool *stats_flag*) [inline, protected, virtual]

resets number of samples and sampling flags used by DataFitSurrModel::build_global() to publish the minimum number of samples needed from the sampling routine (to build a particular global approximation) and to set allDataFlag and statsFlag. In this case, allDataFlag is set to true (vectors of variable and response sets must be returned to build the global approximation) and statsFlag is set to false (statistics computations are not needed).

Reimplemented from Iterator.

References NonDSampling::allDataFlag, NonDSampling::numSamples, NonDSampling::samplesRef, and NonDSampling::statsFlag.

43.96.3.3 void get_parameter_sets (Model & *model*) [protected, virtual]

Uses lhsDriver to generate a set of samples from the distributions/bounds defined in the incoming model. This version of get_parameter_sets() extracts data from the user-defined model in any of the four sampling modes.

Reimplemented from Analyzer.

References Model::all_continuous_lower_bounds(), Model::all_continuous_upper_bounds(), Model::all_discrete_int_lower_bounds(), Model::all_discrete_int_upper_bounds(), Analyzer::allSamples, Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::current_variables(), Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Model::discrete_int_lower_bounds(), Model::discrete_int_upper_bounds(), Model::discrete_state_set_int_values(), Model::discrete_state_set_real_values(), Model::distribution_parameters(), NonDSampling::initialize_lhs(), NonDSampling::lhsDriver, NonD::numContAleatUncVars, NonD::numContDesVars, NonD::numContEpistUncVars, NonD::numContStateVars, NonD::numDiscIntAleatUncVars, NonD::numDiscIntDesVars, NonD::numDiscIntEpistUncVars, NonD::numDiscIntStateVars, NonDSampling::numSamples, NonDSampling::sampleRanks, NonDSampling::samplingVarsMode, and Variables::view().

Referenced by NonDLHSSampling::NonDLHSSampling(), NonDLHSSampling::pre_run(), NonDIncremLHSSampling::quantify_uncertainty(), and NonDAdaptImpSampling::quantify_uncertainty().

43.96.3.4 void get_parameter_sets (const RealVector & *lower_bnds*, const RealVector & *upper_bnds*) [protected]

Uses lhsDriver to generate a set of uniform samples over lower_bnds/upper_bnds. This version of get_parameter_sets() does not extract data from the user-defined model, but instead relies on the incoming bounded region definition. It only support a UNIFORM sampling mode, where the distinction of ACTIVE_UNIFORM vs. ALL_UNIFORM is handled elsewhere.

References Analyzer::allSamples, NonDSampling::initialize_lhs(), NonDSampling::lhsDriver, and NonDSampling::numSamples.

The documentation for this class was generated from the following files:

- NonDSampling.H
- NonDSampling.C

43.97 NonDSparseGrid Class Reference

Derived nondeterministic class that generates N-dimensional Smolyak sparse grids for numerical evaluation of expectation integrals over independent standard random variables. Inheritance diagram for NonDSparseGrid::

Public Member Functions

- NonDSparseGrid (Model &model, const UShortArray &ssg_level, const RealVector &dim_pref, short refine_control=Pecos::NO_CONTROL, bool track_uniq_prod_wts=true, short growth_rate=Pecos::MODERATE_RESTRICTED_GROWTH)
- void increment_grid ()

  increment ssgDriver::ssgLevel

- void increment_grid_weights (const RealVector &aniso_wts)

  update ssgDriver::ssgAnisoLevelWts and increment ssgDriver::ssgLevel based on specified anisotropic weighting

- void increment_refinement_sequence ()

  advance to next nevel in ssgLevelSpec sequence

- const std::set< UShortArray > & active_multi_index () const

  returns SparseGridDriver::active_multi_index()

- const std::set< UShortArray > & old_multi_index () const

  returns SparseGridDriver::old_multi_index()

- const UShort2DArray & smolyak_multi_index () const

  returns SparseGridDriver::smolyak_multi_index()

- const IntArray & smolyak_coefficients () const

  returns SparseGridDriver::smolyak_coefficients()

- void initialize_sets ()

  invokes SparseGridDriver::initialize_sets()

- void update_reference ()

  invokes SparseGridDriver::update_reference()

- void increment_set (const UShortArray &set)

  invokes SparseGridDriver::push_trial_set()

- int increment_size () const

  invokes SparseGridDriver::unique_trial_points()

- void restore_set ()

  invokes SparseGridDriver::restore_set()

- void evaluate_set ()

  invokes SparseGridDriver::compute_trial_grid()

- void decrement_set ()

  invokes SparseGridDriver::pop_trial_set()

- void update_sets (const UShortArray &set_star)

  invokes SparseGridDriver::update_sets()

- void print_final_sets (bool converged_within_tol)

  invokes SparseGridDriver::print_final_sets(bool)

- void finalize_sets ()

  invokes SparseGridDriver::finalize_sets()

- int num_samples () const

  get the current number of samples

Protected Member Functions

- NonDSparseGrid (Model &model)

  constructor

- ~NonDSparseGrid ()

  destructor

- void initialize_grid (const std::vector< Pecos::BasisPolynomial > &poly_basis)

  initialize integration grid by drawing from polynomial basis settings

- void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

- void reset ()

  restore initial state for repeated sub-iterator executions

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)

Private Attributes

- Pecos::SparseGridDriver * ssgDriver

  convenience pointer to the numIntDriver representation

- UShortArray ssgLevelSpec

  the user specification for the Smolyak sparse grid level, defining a sequence of refinement levels.

- unsigned short ssgLevelRef

  reference point (e.g., lower bound) for the Smolyak sparse grid level maintained within ssgDriver

43.97.1 Detailed Description

Derived nondeterministic class that generates N-dimensional Smolyak sparse grids for numerical evaluation of expectation integrals over independent standard random variables. This class is used by NonDPolynomialChaos and NonDStochCollocation, but could also be used for general numerical integration of moments. It employs 1-D Clenshaw-Curtis and Gaussian quadrature rules within Smolyak sparse grids.

43.97.2 Constructor & Destructor Documentation

43.97.2.1 NonDSparseGrid (Model & *model*, const UShortArray & *ssg_level*, const RealVector & *dim_pref*, short *refine_control* = `Pecos::NO_CONTROL`, bool *track_uniq_prod_wts* = `true`, short *growth_rate* = `Pecos::MODERATE_RESTRICTED_GROWTH`)

This alternate constructor is used for on-the-fly generation and evaluation of sparse grids within PCE and SC.

References NonDIntegration::numIntDriver, and NonDSparseGrid::ssgDriver.

43.97.2.2 NonDSparseGrid (Model & *model*) `[protected]`

constructor This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_-list_nodes has been called and probDescDB can be queried for settings from the method specification. It is not currently used, as there is not a separate sparse_grid method specification.

References NonDIntegration::check_variables(), NonDIntegration::dimPrefSpec, Model::distribution_-parameters(), ProblemDescDB::get_bool(), ProblemDescDB::get_short(), Iterator::iteratedModel, Iterator::maxConcurrency, NonD::natafTransform, NonDIntegration::numIntDriver, Iterator::probDescDB, NonDSparseGrid::ssgDriver, and NonDSparseGrid::ssgLevelRef.

43.97.3 Member Function Documentation

43.97.3.1 int num_samples () const [inline, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References NonDSparseGrid::ssgDriver.

43.97.3.2 void sampling_reset (int *min_samples*, bool *all_data_flag*, bool *stats_flag*) [protected, virtual]

used by DataFitSurrModel::build_global() to publish the minimum number of points needed from the sparse grid routine in order to build a particular global approximation.

Reimplemented from Iterator.

References NonDSparseGrid::ssgDriver, and NonDSparseGrid::ssgLevelRef.

The documentation for this class was generated from the following files:

- NonDSparseGrid.H
- NonDSparseGrid.C

43.98 NonDStochCollocation Class Reference

Nonintrusive stochastic collocation approaches to uncertainty quantification. Inheritance diagram for NonDStochCollocation::

Public Member Functions

- NonDStochCollocation (Model &model)

  standard constructor

- NonDStochCollocation (Model &model, short exp_coeffs_approach, unsigned short num_int_level, short u_space_type, bool piecewise_basis, bool use_derivs)

  alternate constructor

- ~NonDStochCollocation ()

  destructor

Protected Member Functions

- void resolve_inputs (short &u_space_type, short &data_order)

  perform error checks and mode overrides

- void initialize_u_space_model ()

  initialize uSpaceModel polynomial approximations with PCE/SC data

43.98.1 Detailed Description

Nonintrusive stochastic collocation approaches to uncertainty quantification. The NonDStochCollocation class uses a stochastic collocation (SC) approach to approximate the effect of parameter uncertainties on response functions of interest. It utilizes the InterpPolyApproximation class to manage multidimensional Lagrange polynomial interpolants.

43.98.2 Constructor & Destructor Documentation

43.98.2.1 NonDStochCollocation (Model & *model*)

standard constructor This constructor is called for a standard letter-envelope iterator instantiation using the ProblemDescDB.

References Model::assign_rep(), NonDExpansion::construct_expansion_sampler(), NonDExpansion::construct_quadrature(), NonDExpansion::construct_sparse_grid(), NonD::construct_u_space_model(), Model::derivative_concurrency(), NonDExpansion::expansionCoeffsApproach, ProblemDescDB::get_dusa(), ProblemDescDB::get_rdv(), ProblemDescDB::get_short(), Model::init_communicators(), NonDExpansion::initialize(), NonDStochCollocation::initialize_u_space_model(), Iterator::iteratedModel, NonD::numContDesVars, NonD::numContEpistUncVars, NonD::numContStateVars, NonDExpansion::numSamplesOnExpansion, Iterator::outputLevel, NonDExpansion::piecewiseBasis, Iterator::probDescDB, NonDStochCollocation::resolve_inputs(), and NonDExpansion::uSpaceModel.

43.98.2.2 NonDStochCollocation (Model & *model*, short *exp_coeffs_approach*, unsigned short *num_int_level*, short *u_space_type*, bool *piecewise_basis*, bool *use_derivs*)

alternate constructor This constructor is used for helper iterator instantiation on the fly.

References Model::assign_rep(), NonDExpansion::construct_quadrature(), NonDExpansion::construct_sparse_grid(), NonD::construct_u_space_model(), NonDExpansion::expansionCoeffsApproach, NonDExpansion::initialize(), NonDStochCollocation::initialize_u_space_model(), Iterator::iteratedModel, NonD::numContDesVars, NonD::numContEpistUncVars, NonD::numContStateVars, Iterator::outputLevel, NonDExpansion::piecewiseBasis, NonDExpansion::refineType, NonDStochCollocation::resolve_inputs(), and NonDExpansion::uSpaceModel.

The documentation for this class was generated from the following files:

- NonDStochCollocation.H
- NonDStochCollocation.C

43.99 NPSOLOptimizer Class Reference

Wrapper class for the NPSOL optimization library. Inheritance diagram for NPSOLOptimizer::

Public Member Functions

- NPSOLOptimizer (Model &model)

  standard constructor

- NPSOLOptimizer (NoDBBaseConstructor, Model &model)

  alternate constructor for Iterator instantiations by name

- NPSOLOptimizer (Model &model, const int &derivative_level, const Real &conv_tol)

  alternate constructor for instantiations "on the fly"

- NPSOLOptimizer (const RealVector &initial_point, const RealVector &var_lower_bnds, const RealVector &var_upper_bnds, const RealMatrix &lin_ineq_coeffs, const RealVector &lin_ineq_lower_bnds, const RealVector &lin_ineq_upper_bnds, const RealMatrix &lin_eq_coeffs, const RealVector &lin_eq_targets, const RealVector &nonlin_ineq_lower_bnds, const RealVector &nonlin_ineq_upper_bnds, const RealVector &nonlin_eq_targets, void(∗user_obj_eval)(int &, int &, double ∗, double &, double ∗, int &), void(∗user_con_eval)(int &, int &, int &, int &, int ∗, double ∗, double ∗, double ∗, int &), const int &derivative_level, const Real &conv_tol)

  alternate constructor for instantiations "on the fly"

- ∼NPSOLOptimizer ()

  destructor

- void find_optimum ()

  Used within the optimizer branch for computing the optimal solution. Redefines the run virtual function for the optimizer branch.

Private Member Functions

- void find_optimum_on_model ()

called by find_optimum for setUpType == "model"

- void find_optimum_on_user_functions ()

  called by find_optimum for setUpType == "user_functions"

Static Private Member Functions

- static void objective_eval (int &mode, int &n, double ∗x, double &f, double ∗gradf, int &nstate)

  OBJFUN in NPSOL manual: computes the value and first derivatives of the objective function (passed by function pointer to NPSOL).

Private Attributes

- String setUpType

  controls iteration mode: "model" (normal usage) or "user_functions" (user-supplied functions mode for "on the fly" instantiations). NonDReliability currently uses the user_functions mode.

- RealVector initialPoint

  holds initial point passed in for "user_functions" mode.

- RealVector lowerBounds

  holds variable lower bounds passed in for "user_functions" mode.

- RealVector upperBounds

  holds variable upper bounds passed in for "user_functions" mode.

- void(∗ userObjectiveEval)(int &, int &, double ∗, double &, double ∗, int &)

  holds function pointer for objective function evaluator passed in for "user_functions" mode.

- void(∗ userConstraintEval)(int &, int &, int &, int &, int ∗, double ∗, double ∗, double ∗, int &)

  holds function pointer for constraint function evaluator passed in for "user_functions" mode.

Static Private Attributes

- static NPSOLOptimizer ∗ npsolInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.99.1 Detailed Description

Wrapper class for the NPSOL optimization library. The NPSOLOptimizer class provides a wrapper for NPSOL, a Fortran 77 sequential quadratic programming library from Stanford University marketed by Stanford Business Associates. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or accessed through a static pointer.

The user input mappings are as follows: `max_function_evaluations` is implemented directly in NPSOLOptimizer's evaluator functions since there is no NPSOL parameter equivalent, and `max_iterations`, `convergence_tolerance`, `output` verbosity, `verify_level`, `function_precision`, and `linesearch_tolerance` are mapped into NPSOL's "Major Iteration Limit", "Optimality Tolerance", "Major Print Level" (`verbose:` Major Print Level = 20; `quiet:` Major Print Level = 10), "Verify Level", "Function Precision", and "Linesearch Tolerance" parameters, respectively, using NPSOL's npoptn() subroutine (as wrapped by npoptn2() from the npoptn_wrapper.f file). Refer to [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] for information on NPSOL's optional input parameters and the npoptn() subroutine.

43.99.2 Constructor & Destructor Documentation

43.99.2.1 NPSOLOptimizer (Model & *model*)

standard constructor This is the primary constructor. It accepts a Model reference.

References Minimizer::constraintTol, Iterator::convergenceTol, Iterator::fdGradStepSize, ProblemDescDB::get_int(), ProblemDescDB::get_real(), Iterator::gradientType, Iterator::maxIterations, Iterator::outputLevel, Iterator::probDescDB, SOLBase::set_options(), Minimizer::speculativeFlag, and Minimizer::vendorNumericalGradFlag.

43.99.2.2 NPSOLOptimizer (NoDBBaseConstructor, Model & *model*)

alternate constructor for Iterator instantiations by name This is an alternate constructor which accepts a Model but does not have a supporting method specification from the ProblemDescDB.

References Minimizer::constraintTol, Iterator::convergenceTol, Iterator::fdGradStepSize, Iterator::gradientType, Iterator::maxIterations, Iterator::outputLevel, SOLBase::set_options(), Minimizer::speculativeFlag, and Minimizer::vendorNumericalGradFlag.

43.99.2.3 NPSOLOptimizer (Model & *model*, const int & *derivative_level*, const Real & *conv_tol*)

alternate constructor for instantiations "on the fly" This is an alternate constructor for instantiations on the fly using a Model but no ProblemDescDB.

43.99.2.4 NPSOLOptimizer (const RealVector & *initial_point*, const RealVector & *var_lower_bnds*, const RealVector & *var_upper_bnds*, const RealMatrix & *lin_ineq_coeffs*, const RealVector & *lin_ineq_lower_bnds*, const RealVector & *lin_ineq_upper_bnds*, const RealMatrix & *lin_eq_coeffs*, const RealVector & *lin_eq_targets*, const RealVector & *nonlin_ineq_lower_bnds*, const RealVector & *nonlin_ineq_upper_bnds*, const RealVector & *nonlin_eq_targets*, void(∗)(int &, int &, double ∗, double &, double ∗, int &) *user_obj_eval*, void(∗)(int &, int &, int &, int &, int ∗, double ∗, double ∗, double ∗, int &) *user_con_eval*, const int & *derivative_level*, const Real & *conv_tol*)

alternate constructor for instantiations "on the fly" This is an alternate constructor for performing an optimization using the passed in objective function and constraint function pointers.

References SOLBase::allocate_arrays(), SOLBase::allocate_workspace(), SOLBase::augment_bounds(), NPSOLOptimizer::lowerBounds, Iterator::numContinuousVars, Minimizer::numLinearConstraints, Minimizer::numNonlinearConstraints, and NPSOLOptimizer::upperBounds.

The documentation for this class was generated from the following files:

- NPSOLOptimizer.H
- NPSOLOptimizer.C

43.100 Optimizer Class Reference

Base class for the optimizer branch of the iterator hierarchy. Inheritance diagram for Optimizer::

Static Public Member Functions

- static void not_available (const std::string &package_name)

  Static helper function: third-party opt packages which are not available.

Protected Member Functions

- Optimizer ()

  default constructor

- Optimizer (Model &model)

  standard constructor

- Optimizer (NoDBBaseConstructor, Model &model)

  alternate constructor for "on the fly" instantiations

- Optimizer (NoDBBaseConstructor, size_t num_cv, size_t num_div, size_t num_drv, size_t num_lin_ineq, size_t num_lin_eq, size_t num_nln_ineq, size_t num_nln_eq)

  alternate constructor for "on the fly" instantiations

- ~Optimizer ()

  destructor

- void initialize_run ()
- void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- void post_run (std::ostream &s)
- void finalize_run ()

  utility function to perform common operations following post_run(); deallocation and resetting of instance pointers

- void print_results (std::ostream &s)
- virtual void find_optimum ()=0

  Used within the optimizer branch for computing the optimal solution. Redefines the run virtual function for the optimizer branch.

Protected Attributes

- size_t numObjectiveFns

  number of objective functions (iterator view)

- bool localObjectiveRecast

  flag indicating whether local recasting to a single objective is used

- Optimizer * prevOptInstance

  pointer containing previous value of optimizerInstance

Static Protected Attributes

- static Optimizer * optimizerInstance

  pointer to Optimizer instance used in static member functions

Private Member Functions

- void objective_reduction (const Response &full_response, const RealVector &full_wts, Response &reduced_response) const

  forward mapping: maps multiple primary response functions to a single weighted objective for single-objective optimizers

- void local_objective_recast_retrieve (const Variables &vars, Response &response) const

  infers MOO/NLS solution from the solution of a single-objective optimizer

Static Private Member Functions

- static void primary_resp_recast (const Variables &native_vars, const Variables &scaled_vars, const Response &native_response, Response &scaled_response)

  primary response conversion map for RecastModel used in scaling and multiobjective: transform objectives (fns, grads, Hessians) from native (user) to iterator space

43.100.1 Detailed Description

Base class for the optimizer branch of the iterator hierarchy. The Optimizer class provides common data and functionality for DOTOptimizer, CONMINOptimizer, NPSOLOptimizer, SNLLOptimizer, NLPQLPOptimizer, COLINOptimizer, and JEGAOptimizer.

43.100.2 Constructor & Destructor Documentation

43.100.2.1 Optimizer (Model & *model*) `[protected]`

standard constructor This constructor extracts the inherited data for the optimizer branch and performs sanity checking on gradient and constraint settings.

References Dakota::abort_handler(), Iterator::activeSet, Model::assign_rep(), String::begins(), Iterator::bestVariablesArray, Minimizer::boundConstraintFlag, Variables::copy(), Model::current_response(), Model::current_variables(), Minimizer::cvScaleTypes, ProblemDescDB::get_sizet(), Minimizer::gnewton_set_recast(), Iterator::gradientType, Iterator::hessianType, Model::init_communicators(), RecastModel::initialize(), Minimizer::initialize_scaling(), Iterator::iteratedModel, Optimizer::localObjectiveRecast, Iterator::maxConcurrency, Iterator::methodName, Model::model_rep(), Model::model_type(), Iterator::numContinuousVars, Iterator::numFunctions, Minimizer::numIterPrimaryFns, Minimizer::numNonlinearConstraints, Minimizer::numNonlinearIneqConstraints, Optimizer::numObjectiveFns, Minimizer::numUserPrimaryFns, Minimizer::optimizationFlag, Optimizer::primary_resp_recast(), Minimizer::primaryRespScaleFlag, Iterator::probDescDB, ActiveSet::request_vector(), Response::reshape(), Minimizer::responseScaleTypes, Minimizer::scaleFlag, Minimizer::secondary_resp_recast(), Minimizer::secondaryRespScaleFlag, Minimizer::speculativeFlag, Minimizer::variables_recast(), and Minimizer::varsScaleFlag.

43.100.3 Member Function Documentation

43.100.3.1 void initialize_run () [protected, virtual]

Implements portions of initialize_run specific to Optimizers. This function should be invoked (or reimplemented) by any derived implementations of initialize_run() (which would otherwise hide it).

Reimplemented from Minimizer.

Reimplemented in CONMINOptimizer, DOTOptimizer, NLPQLPOptimizer, and SNLLOptimizer.

References Iterator::iteratedModel, Optimizer::localObjectiveRecast, Optimizer::optimizerInstance, Optimizer::prevOptInstance, Minimizer::scaleFlag, and Model::update_from_subordinate_model().

43.100.3.2 void run () [inline, protected, virtual]

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Iterator.

References Optimizer::find_optimum().

43.100.3.3 void post_run (std::ostream & *s*) [protected, virtual]

Implements portions of post_run specific to Optimizers. This function should be invoked (or reimplemented) by any derived implementations of post_run() (which would otherwise hide it).

Reimplemented from Iterator.

Reimplemented in COLINOptimizer, and SNLLOptimizer.

References Dakota::abort_handler(), Response::active_set_request_vector(), Iterator::bestResponseArray, Iterator::bestVariablesArray, Variables::continuous_variables(), Response::copy(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Optimizer::local_objective_recast_retrieve(), Optimizer::localObjectiveRecast, Minimizer::modify_s2n(), Minimizer::need_resp_trans_byvars(), Minimizer::numNonlinearConstraints, Minimizer::numUserPrimaryFns, Minimizer::primaryRespScaleFlag, Minimizer::response_modify_s2n(), Minimizer::secondaryRespScaleFlag, Response::update_partial(), and Minimizer::varsScaleFlag.

43.100.3.4 void finalize_run () [inline, protected, virtual]

utility function to perform common operations following post_run(); deallocation and resetting of instance pointers Optional: perform finalization phases of run sequence, like deallocating memory and resetting instance pointers. Commonly used in sub-iterator executions. This is a virtual function; when re-implementing, a derived class must call its nearest parent's finalize_run(), typically _after_ performing its own implementation steps.

Reimplemented from Minimizer.

Reimplemented in SNLLOptimizer.

References Optimizer::optimizerInstance, and Optimizer::prevOptInstance.

43.100.3.5 void print_results (std::ostream & *s*) [protected, virtual]

Redefines default iterator results printing to include optimization results (objective functions and constraints).

Reimplemented from Iterator.

References Dakota::abort_handler(), Iterator::bestResponseArray, Iterator::bestVariablesArray, Dakota::data_pairs, Model::interface_id(), Iterator::iteratedModel, Dakota::lookup_by_val(), Iterator::numContinuousVars, Iterator::numFunctions, Minimizer::numUserPrimaryFns, and Dakota::write_data_partial().

43.100.3.6 void primary_resp_recast (const Variables & *native_vars*, const Variables & *scaled_vars*, const Response & *native_response*, Response & *iterator_response*) [static, private]

primary response conversion map for RecastModel used in scaling and multiobjective: transform objectives (fns, grads, Hessians) from native (user) to iterator space Objective function map from user/native space to iterator/scaled/combined space using a RecastModel. If resizing the response, copies the constraint (secondary) data from native_response too

References Response::active_set_request_vector(), Response::copy(), Iterator::iteratedModel, Optimizer::localObjectiveRecast, Minimizer::need_resp_trans_byvars(), Minimizer::numUserPrimaryFns, Optimizer::objective_reduction(), Optimizer::optimizerInstance, Iterator::outputLevel, Model::primary_response_fn_weights(), Minimizer::primaryRespScaleFlag, Minimizer::response_modify_n2s(), Model::subordinate_model(), and Response::update_partial().

Referenced by Optimizer::Optimizer().

43.100.3.7 void objective_reduction (const Response & *full_response*, const RealVector & *full_wts*, Response & *reduced_response*) const [private]

forward mapping: maps multiple primary response functions to a single weighted objective for single-objective optimizers This function is responsible for the mapping of multiple objective functions into a single objective for publishing to single-objective optimizers. Used in DOTOptimizer, NPSOLOptimizer, SNLLOptimizer, and SGOPTApplication on every function evaluation. The simple weighting approach (using primaryRespFnWts) is the only technique supported currently. The weightings are used to scale function values, gradients, and Hessians as needed.

References Dakota::abort_handler(), Response::active_set_request_vector(), Response::function_gradient_view(), Response::function_gradients(), Response::function_hessian_view(), Response::function_hessians(), Response::function_value(), Response::function_values(), Minimizer::numUserPrimaryFns, Minimizer::objective(), Minimizer::objective_gradient(), Minimizer::objective_hessian(), Iterator::outputLevel, Dakota::write_col_vector_trans(), Dakota::write_data(), and Dakota::write_precision.

Referenced by Optimizer::primary_resp_recast().

43.100.3.8 void local_objective_recast_retrieve (const Variables & *vars*, Response & *response*) const [private]

infers MOO/NLS solution from the solution of a single-objective optimizer Retrieve a MOO/NLS response based on the data returned by a single objective optimizer by performing a data_pairs search.

References Response::active_set(), Dakota::data_pairs, Model::interface_id(), Iterator::iteratedModel, Dakota::lookup_by_val(), Minimizer::numUserPrimaryFns, and Response::update().

Referenced by Optimizer::post_run().

The documentation for this class was generated from the following files:

- DakotaOptimizer.H
- DakotaOptimizer.C

43.101 ParallelConfiguration Class Reference

Container class for a set of ParallelLevel list iterators that collectively identify a particular multilevel parallel configuration.

Public Member Functions

- ParallelConfiguration ()

  default constructor

- ParallelConfiguration (const ParallelConfiguration &pl)

  copy constructor

- ~ParallelConfiguration ()

  destructor

- ParallelConfiguration & operator= (const ParallelConfiguration &pl)

  assignment operator

- const ParallelLevel & w_parallel_level () const

  return the ParallelLevel corresponding to wPLIter

- const ParallelLevel & si_parallel_level () const

  return the ParallelLevel corresponding to siPLIter

- const ParallelLevel & ie_parallel_level () const

  return the ParallelLevel corresponding to iePLIter

- const ParallelLevel & ea_parallel_level () const

  return the ParallelLevel corresponding to eaPLIter

Private Member Functions

- void assign (const ParallelConfiguration &pl)

  assign the attributes of the incoming pl to this object

Private Attributes

- short numParallelLevels

  number of parallel levels

- ParLevLIter wPLIter

list iterator for MPI_COMM_WORLD (not strictly required, but improves modularity by avoiding explicit usage of MPI_COMM_WORLD)

- ParLevLIter siPLIter

  list iterator for concurrent iterator partitions (there may be more than one per parallel configuration instance)

- ParLevLIter iePLIter

  list iterator identifying the iterator-evaluation parallelLevel (there can only be one)

- ParLevLIter eaPLIter

  list iterator identifying the evaluation-analysis parallelLevel (there can only be one)

Friends

- class ParallelLibrary

  the ParallelLibrary class has special access priveleges in order to streamline implementation

43.101.1 Detailed Description

Container class for a set of ParallelLevel list iterators that collectively identify a particular multilevel parallel configuration. Rather than containing the multilevel parallel configuration directly, ParallelConfiguration instead provides a set of list iterators which point into a combined list of ParallelLevels. This approach allows different configurations to reuse ParallelLevels without copying them. A list of ParallelConfigurations is contained in ParallelLibrary (ParallelLibrary::parallelConfigurations).

The documentation for this class was generated from the following file:

- ParallelLibrary.H

43.102 ParallelDirectApplicInterface Class Reference

Sample derived interface class for testing parallel simulator plug-ins using assign_rep(). Inheritance diagram for ParallelDirectApplicInterface::

Public Member Functions

- ParallelDirectApplicInterface (const Dakota::ProblemDescDB &problem_db, const MPI_Comm &analysis_comm)

  constructor

- ~ParallelDirectApplicInterface ()

  destructor

Protected Member Functions

- int derived_map_ac (const Dakota::String &ac_name)

  execute an analysis code portion of a direct evaluation invocation

43.102.1 Detailed Description

Sample derived interface class for testing parallel simulator plug-ins using assign_rep(). The plug-in ParallelDirectApplicInterface resides in namespace SIM and uses a copy of textbook() to perform parallel parameter to response mappings. It may be activated by specifying the --with-plugin configure option, which activates the DAKOTA_PLUGIN macro in dakota_config.h used by main.C (which activates the plug-in code block within that file) and activates the PLUGIN_S declaration defined in Makefile.include and used in Makefile.source (which add this class to the build). Test input files should then use an analysis_driver of "plugin_textbook".

The documentation for this class was generated from the following files:

- PluginParallelDirectApplicInterface.H
- PluginParallelDirectApplicInterface.C

43.103 ParallelLevel Class Reference

Container class for the data associated with a single level of communicator partitioning.

Public Member Functions

- ParallelLevel ()

  default constructor

- ParallelLevel (const ParallelLevel &pl)

  copy constructor

- ~ParallelLevel ()

  destructor

- ParallelLevel & operator= (const ParallelLevel &pl)

  assignment operator

- bool dedicated_master_flag () const

  return dedicatedMasterFlag

- bool communicator_split_flag () const

  return commSplitFlag

- bool server_master_flag () const

  return serverMasterFlag

- bool message_pass () const

  return messagePass

- const int & num_servers () const

  return numServers

- const int & processors_per_server () const

  return procsPerServer

- const int & processor_remainder () const

  return procRemainder

- const MPI_Comm & server_intra_communicator () const

  return serverIntraComm

- const int & server_communicator_rank () const

  return serverCommRank

- const int & server_communicator_size () const

  return serverCommSize

- const MPI_Comm & hub_server_intra_communicator () const

  return hubServerIntraComm

- const int & hub_server_communicator_rank () const

  return hubServerCommRank

- const int & hub_server_communicator_size () const

  return hubServerCommSize

- const MPI_Comm & hub_server_inter_communicator () const

  return hubServerInterComm

- MPI_Comm ∗ hub_server_inter_communicators () const

  return hubServerInterComms

- const int & server_id () const

  return serverId

Private Member Functions

- void assign (const ParallelLevel &pl)

  assign the attributes of the incoming pl to this object

Private Attributes

- bool dedicatedMasterFlag

  signals dedicated master partitioning

- bool commSplitFlag

  signals a communicator split was used

- bool serverMasterFlag

  identifies master server processors

- bool messagePass

  flag for message passing at this level

- int numServers

  number of servers

- int procsPerServer
  processors per server
- int procRemainder
  proc remainder after equal distribution
- MPI_Comm serverIntraComm
  intracomm. for each server partition
- int serverCommRank
  rank in serverIntraComm
- int serverCommSize
  size of serverIntraComm
- MPI_Comm hubServerIntraComm
  intracomm for all serverCommRank==0 < w/i next higher level serverIntraComm
- int hubServerCommRank
  rank in hubServerIntraComm
- int hubServerCommSize
  size of hubServerIntraComm
- MPI_Comm hubServerInterComm
  intercomm. between a server & the hub < (on server partitions only)
- MPI_Comm * hubServerInterComms
  intercomm. array on hub processor
- int serverId
  server identifier

Friends

- class ParallelLibrary
  the ParallelLibrary class has special access priveleges in order to streamline implementation

43.103.1 Detailed Description

Container class for the data associated with a single level of communicator partitioning. A list of these levels is contained in ParallelLibrary (ParallelLibrary::parallelLevels), which defines all of the parallelism levels across one or more multilevel parallelism configurations.

The documentation for this class was generated from the following file:

- ParallelLibrary.H

43.104 ParallelLibrary Class Reference

Class for partitioning multiple levels of parallelism and managing message passing within these levels.

Public Member Functions

- ParallelLibrary (int &argc, char **&argv)

  stand-alone mode constructor

- ParallelLibrary ()

  default library mode constructor (assumes MPI_COMM_WORLD)

- ParallelLibrary (MPI_Comm dakota_mpi_comm)

  library mode constructor accepting communicator

- ParallelLibrary (const std::string &dummy)

  dummy constructor (used for dummy_lib)

- ~ParallelLibrary ()

  destructor

- const ParallelLevel & init_iterator_communicators (const int &iterator_servers, const int &procs_per_iterator, const int &max_iterator_concurrency, const std::string &default_config, const std::string &iterator_scheduling)

  split MPI_COMM_WORLD into iterator communicators

- const ParallelLevel & init_evaluation_communicators (const int &evaluation_servers, const int &procs_per_evaluation, const int &max_evaluation_concurrency, const int &asynch_local_evaluation_concurrency, const std::string &default_config, const std::string &evaluation_scheduling)

  split an iterator communicator into evaluation communicators

- const ParallelLevel & init_analysis_communicators (const int &analysis_servers, const int &procs_per_analysis, const int &max_analysis_concurrency, const int &asynch_local_analysis_concurrency, const std::string &default_config, const std::string &analysis_scheduling)

  split an evaluation communicator into analysis communicators

- void free_iterator_communicators ()

  deallocate iterator communicators

- void free_evaluation_communicators ()

  deallocate evaluation communicators

- void free_analysis_communicators ()

  deallocate analysis communicators

- void print_configuration ()

print the parallel level settings for a particular parallel configuration

- void specify_outputs_restart (CommandLineHandler &cmd_line_handler)

  specify output streams and restart file(s) using command line inputs (normal mode)

- void specify_outputs_restart (const char *clh_std_output_filename=NULL, const char *clh_std_error_filename=NULL, const char *clh_read_restart_filename=NULL, const char *clh_write_restart_filename=NULL, int stop_restart_evals=0, bool pre_run_flag=false)

  specify output streams and restart file(s) using external inputs (library mode).

- void manage_outputs_restart (const ParallelLevel &pl)

  manage output streams and restart file(s) (both modes)

- void close_streams ()

  close streams, files, and any other services

- void abort_helper (int code) const

  finalize MPI with correct communicator for abort

- void output_helper (const std::string &s, std::ostream &outfile=Cout) const

  perform stdout on rank 0 only

- bool command_line_check () const

  return checkFlag

- bool command_line_pre_run () const

  return preRunFlag

- bool command_line_run () const

  return runFlag

- bool command_line_post_run () const

  return postRunFlag

- bool command_line_user_modes () const

  return userModesFlag

- const std::string & command_line_pre_run_input () const

  preRunInput filename

- const std::string & command_line_pre_run_output () const

  preRunOutput filename

- const std::string & command_line_run_input () const

  runInput filename

- const std::string & command_line_run_output () const
  runOutput filename
- const std::string & command_line_post_run_input () const
  postRunInput filename
- const std::string & command_line_post_run_output () const
  postRunOutput fname
- void send_si (int &send_int, int dest, int tag)
  blocking send at the strategy-iterator communication level
- void recv_si (int &recv_int, int source, int tag, MPI_Status &status)
  blocking receive at the strategy-iterator communication level
- void send_si (MPIPackBuffer &send_buff, int dest, int tag)
  blocking send at the strategy-iterator communication level
- void isend_si (MPIPackBuffer &send_buff, int dest, int tag, MPI_Request &send_req)
  nonblocking send at the strategy-iterator communication level
- void recv_si (MPIUnpackBuffer &recv_buff, int source, int tag, MPI_Status &status)
  blocking receive at the strategy-iterator communication level
- void irecv_si (MPIUnpackBuffer &recv_buff, int source, int tag, MPI_Request &recv_req)
  nonblocking receive at the strategy-iterator communication level
- void send_ie (MPIPackBuffer &send_buff, int dest, int tag)
  blocking send at the iterator-evaluation communication level
- void isend_ie (MPIPackBuffer &send_buff, int dest, int tag, MPI_Request &send_req)
  nonblocking send at the iterator-evaluation communication level
- void recv_ie (MPIUnpackBuffer &recv_buff, int source, int tag, MPI_Status &status)
  blocking receive at the iterator-evaluation communication level
- void irecv_ie (MPIUnpackBuffer &recv_buff, int source, int tag, MPI_Request &recv_req)
  nonblocking receive at the iterator-evaluation communication level
- void send_ea (int &send_int, int dest, int tag)
  blocking send at the evaluation-analysis communication level
- void isend_ea (int &send_int, int dest, int tag, MPI_Request &send_req)
  nonblocking send at the evaluation-analysis communication level
- void recv_ea (int &recv_int, int source, int tag, MPI_Status &status)

blocking receive at the evaluation-analysis communication level

- void irecv_ea (int &recv_int, int source, int tag, MPI_Request &recv_req)

  nonblocking receive at the evaluation-analysis communication level

- void bcast_w (int &data)

  broadcast an integer across MPI_COMM_WORLD

- void bcast_i (int &data)

  broadcast an integer across an iterator communicator

- void bcast_i (short &data)

  broadcast a short integer across an iterator communicator

- void bcast_e (int &data)

  broadcast an integer across an evaluation communicator

- void bcast_a (int &data)

  broadcast an integer across an analysis communicator

- void bcast_si (int &data)

  broadcast an integer across a strategy-iterator intra communicator

- void bcast_w (MPIPackBuffer &send_buff)

  broadcast a packed buffer across MPI_COMM_WORLD

- void bcast_i (MPIPackBuffer &send_buff)

  broadcast a packed buffer across an iterator communicator

- void bcast_e (MPIPackBuffer &send_buff)

  broadcast a packed buffer across an evaluation communicator

- void bcast_a (MPIPackBuffer &send_buff)

  broadcast a packed buffer across an analysis communicator

- void bcast_si (MPIPackBuffer &send_buff)

  broadcast a packed buffer across a strategy-iterator intra communicator

- void bcast_w (MPIUnpackBuffer &recv_buff)

  matching receive for packed buffer broadcast across MPI_COMM_WORLD

- void bcast_i (MPIUnpackBuffer &recv_buff)

  matching receive for packed buffer bcast across an iterator communicator

- void bcast_e (MPIUnpackBuffer &recv_buff)

  matching receive for packed buffer bcast across an evaluation communicator

- void bcast_a (MPIUnpackBuffer &recv_buff)

  matching receive for packed buffer bcast across an analysis communicator

- void bcast_si (MPIUnpackBuffer &recv_buff)

  matching recv for packed buffer bcast across a strat-iterator intra comm

- void barrier_w ()

  enforce MPI_Barrier on MPI_COMM_WORLD

- void barrier_i ()

  enforce MPI_Barrier on an iterator communicator

- void barrier_e ()

  enforce MPI_Barrier on an evaluation communicator

- void barrier_a ()

  enforce MPI_Barrier on an analysis communicator

- void reduce_sum_ea (double ∗local_vals, double ∗sum_vals, const int &num_vals)

  compute a sum over an eval-analysis intra-communicator using MPI_Reduce

- void reduce_sum_a (double ∗local_vals, double ∗sum_vals, const int &num_vals)

  compute a sum over an analysis communicator using MPI_Reduce

- void test (MPI_Request &request, int &test_flag, MPI_Status &status)

  test a nonblocking send/receive request for completion

- void wait (MPI_Request &request, MPI_Status &status)

  wait for a nonblocking send/receive request to complete

- void waitall (const int &num_recvs, MPI_Request ∗&recv_reqs)

  wait for all messages from a series of nonblocking receives

- void waitsome (const int &num_sends, MPI_Request ∗&recv_requests, int &num_recvs, int ∗&index_-array, MPI_Status ∗&status_array)

  wait for at least one message from a series of nonblocking receives but complete all that are available

- void free (MPI_Request &request)

  free an MPI_Request

- const int & world_size () const

  return worldSize

- const int & world_rank () const

  return worldRank

- bool mpirun_flag () const

  return mpirunFlag

- bool is_null () const

  return dummyFlag

- Real parallel_time () const

  returns current MPI wall clock time

- void parallel_configuration_iterator (const ParConfigLIter &pc_iter)

  set the current ParallelConfiguration node

- const ParConfigLIter & parallel_configuration_iterator () const

  return the current ParallelConfiguration node

- const ParallelConfiguration & parallel_configuration () const

  return the current ParallelConfiguration instance

- size_t num_parallel_configurations () const

  returns the number of entries in parallelConfigurations

- bool parallel_configuration_is_complete ()

  identifies if the current ParallelConfiguration has been fully populated

- void increment_parallel_configuration ()

  add a new node to parallelConfigurations and increment currPCIter

- bool w_parallel_level_defined () const

  test current parallel configuration for definition of world parallel level

- bool si_parallel_level_defined () const

  test current parallel configuration for definition of strategy-iterator parallel level

- bool ie_parallel_level_defined () const

  test current parallel configuration for definition of iterator-evaluation parallel level

- bool ea_parallel_level_defined () const

  test current parallel configuration for definition of evaluation-analysis parallel level

- std::vector< MPI_Comm > analysis_intra_communicators ()

  return the set of analysis intra communicators for all parallel configurations (used for setting up direct simulation interfaces prior to execution time).

Static Public Member Functions

- static bool detect_parallel_launch (int &argc, char **&argv)

  detect parallel launch of DAKOTA using mpirun/mpiexec/poe/etc. based on command line arguments and environment variables

Private Member Functions

- void init_mpi_comm (MPI_Comm dakota_mpi_comm)

  convenience function for initializing from specific comm

- void init_communicators (const ParallelLevel &parent_pl, const int &num_servers, const int &procs_per_server, const int &max_concurrency, const int &asynch_local_concurrency, const std::string &default_config, const std::string &scheduling_override)

  split a parent communicator into child server communicators

- void free_communicators (ParallelLevel &pl)

  deallocate intra/inter communicators for a particular ParallelLevel

- bool split_communicator_dedicated_master (const ParallelLevel &parent_pl, ParallelLevel &child_pl)

  split a parent communicator into a dedicated master processor and num_servers child communicators

- bool split_communicator_peer_partition (const ParallelLevel &parent_pl, ParallelLevel &child_pl)

  split a parent communicator into num_servers peer child communicators (no dedicated master processor)

- bool resolve_inputs (int &num_servers, int &procs_per_server, const int &avail_procs, int &proc_remainder, const int &max_concurrency, const int &capacity_multiplier, const std::string &default_config, const std::string &scheduling_override, bool print_rank)

  resolve user inputs into a sensible partitioning scheme

- void send (MPIPackBuffer &send_buff, const int &dest, const int &tag, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  blocking buffer send at the current communication level

- void send (int &send_int, const int &dest, const int &tag, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  blocking integer send at the current communication level

- void isend (MPIPackBuffer &send_buff, const int &dest, const int &tag, MPI_Request &send_req, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  nonblocking buffer send at the current communication level

- void isend (int &send_int, const int &dest, const int &tag, MPI_Request &send_req, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  nonblocking integer send at the current communication level

- void recv (MPIUnpackBuffer &recv_buff, const int &source, const int &tag, MPI_Status &status, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  blocking buffer receive at the current communication level

- void recv (int &recv_int, const int &source, const int &tag, MPI_Status &status, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  blocking integer receive at the current communication level

- void irecv (MPIUnpackBuffer &recv_buff, const int &source, const int &tag, MPI_Request &recv_req, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  nonblocking buffer receive at the current communication level

- void irecv (int &recv_int, const int &source, const int &tag, MPI_Request &recv_req, ParallelLevel &parent_pl, ParallelLevel &child_pl)

  nonblocking integer receive at the current communication level

- void bcast (int &data, const MPI_Comm &comm)

  broadcast an integer across a communicator

- void bcast (short &data, const MPI_Comm &comm)

  broadcast a short integer across a communicator

- void bcast (MPIPackBuffer &send_buff, const MPI_Comm &comm)

  send a packed buffer across a communicator using a broadcast

- void bcast (MPIUnpackBuffer &recv_buff, const MPI_Comm &comm)

  matching receive for a packed buffer broadcast

- void barrier (const MPI_Comm &comm)

  enforce MPI_Barrier on comm

- void reduce_sum (double *local_vals, double *sum_vals, const int &num_vals, const MPI_Comm &comm)

  compute a sum over comm using MPI_Reduce

- void check_error (const std::string &err_source, const int &err_code)

  check the MPI return code and abort if error

- void manage_run_modes (CommandLineHandler &cmd_line_handler)

  manage run mode information from command-line handler

- void split_filenames (const char *filenames, std::string &input_filename, std::string &output_filename)

  split a double colon separated pair of filenames (possibly empty) into input and output filename strings

Private Attributes

- std::ofstream output_ofstream

  tagged file redirection of stdout

- std::ofstream error_ofstream

  tagged file redirection of stderr

- MPI_Comm dakotaMPIComm

  MPI_Comm on which DAKOTA is running.

- int worldRank

  rank in MPI_Comm in which DAKOTA is running

- int worldSize

  size of MPI_Comm in which DAKOTA is running

- bool mpirunFlag

  flag for a parallel mpirun/yod launch

- bool ownMPIFlag

  flag for ownership of MPI_Init/MPI_Finalize

- bool dummyFlag

  prevents multiple MPI_Finalize calls due to dummy_lib

- bool stdOutputToFile

  flags redirection of DAKOTA std output to a file

- bool stdErrorToFile

  flags redirection of DAKOTA std error to a file

- bool checkFlag

  flags invocation with command line option -check

- bool preRunFlag

  flags invocation with command line option -pre_run

- bool runFlag

  flags invocation with command line option -run

- bool postRunFlag

  flags invocation with command line option -post_run

- bool userModesFlag

  whether user run mdoes are active

- std::string preRunInput

  filename for pre_run input

- std::string preRunOutput

  filename for pre_run output

- std::string runInput

  filename for run input

- std::string runOutput

  filename for run output

- std::string postRunInput

  filename for post_run input

- std::string postRunOutput

  filename for post_run output

- Real startCPUTime

  start reference for UTILIB CPU timer

- Real startWCTime

  start reference for UTILIB wall clock timer

- Real startMPITime

  start reference for MPI wall clock timer

- long startClock

  start reference for local clock() timer measuring < parent+child CPU

- std::string stdOutputFilename

  filename for redirection of stdout

- std::string stdErrorFilename

  filename for redirection of stderr

- std::string readRestartFilename

  input filename for restart

- std::string writeRestartFilename

  output filename for restart

- int stopRestartEvals

  number of evals at which to stop restart processing

- std::list< ParallelLevel > parallelLevels

  the complete set of parallelism levels for managing multilevel parallelism among one or more configurations

- std::list< ParallelConfiguration > parallelConfigurations

  the set of parallel configurations which manage list iterators for indexing into parallelLevels

- ParLevLIter currPLIter

  list iterator identifying the current node in parallelLevels

- ParConfigLIter currPCIter

  list iterator identifying the current node in parallelConfigurations

43.104.1 Detailed Description

Class for partitioning multiple levels of parallelism and managing message passing within these levels. The ParallelLibrary class encapsulates all of the details of performing message passing within multiple levels of parallelism. It provides functions for partitioning of levels according to user configuration input and functions for passing messages within and across MPI communicators for each of the parallelism levels. If support for other message-passing libraries beyond MPI becomes needed (PVM, ...), then ParallelLibrary would be promoted to a base class with virtual functions to encapsulate the library-specific syntax.

43.104.2 Constructor & Destructor Documentation

43.104.2.1 ParallelLibrary (int & *argc*, char ∗∗& *argv*)

stand-alone mode constructor This constructor is the one used by main.C. It calls MPI_Init conditionally based on whether a parallel launch is detected.

References Dakota::abort_handler(), ParallelLibrary::currPLIter, Dakota::Dak_pl, ParallelLibrary::detect_parallel_launch(), ParallelLibrary::increment_parallel_configuration(), ParallelLibrary::mpirunFlag, ParallelLibrary::ownMPIFlag, ParallelLibrary::parallelLevels, ParallelLevel::serverCommRank, ParallelLevel::serverCommSize, ParallelLevel::serverIntraComm, Dakota::start_dakota_heartbeat(), ParallelLibrary::startClock, ParallelLibrary::startCPUTime, ParallelLibrary::startMPITime, ParallelLibrary::startWCTime, ParallelLibrary::worldRank, and ParallelLibrary::worldSize.

43.104.2.2 ParallelLibrary ()

default library mode constructor (assumes MPI_COMM_WORLD) This constructor provides a library mode default ParallelLibrary. It does not call MPI_Init, but rather gathers data from MPI_COMM_WORLD if MPI_Init has been called elsewhere.

References ParallelLibrary::dakotaMPIComm, and ParallelLibrary::init_mpi_comm().

43.104.2.3 ParallelLibrary (MPI_Comm *dakota_mpi_comm*)

library mode constructor accepting communicator This constructor provides a library mode ParallelLibrary, accepting an MPI communicator that might not be MPI_COMM_WORLD. It does not call MPI_Init, but rather gathers data from dakota_mpi_comm if MPI_Init has been called elsewhere.

References ParallelLibrary::dakotaMPIComm, and ParallelLibrary::init_mpi_comm().

43.104.2.4 ParallelLibrary (const std::string & *dummy*)

dummy constructor (used for dummy_lib) This constructor is used for creation of the global dummy_lib object, which is used to satisfy initialization requirements when the real ParallelLibrary object is not available.

43.104.3 Member Function Documentation

43.104.3.1 void specify_outputs_restart (CommandLineHandler & *cmd_line_handler*)

specify output streams and restart file(s) using command line inputs (normal mode) On the rank 0 processor, get the -output, -error, -read_restart, and -write_restart filenames and the -stop_restart limit from the command line. Defaults for the filenames from the command line handler are NULL for the filenames except write which defaults to dakota.rst and 0 for read_restart_evals if no user specification. This information is Bcast from rank 0 to all iterator masters in manage_outputs_restart().

References ParallelLibrary::manage_run_modes(), CommandLineHandler::read_restart_evals(), ParallelLibrary::readRestartFilename, GetLongOpt::retrieve(), ParallelLibrary::stdErrorFilename, ParallelLibrary::stdOutputFilename, ParallelLibrary::stopRestartEvals, ParallelLibrary::worldRank, and ParallelLibrary::writeRestartFilename.

Referenced by main(), and run_dakota().

43.104.3.2 void specify_outputs_restart (const char * *clh_std_output_filename* = `NULL`, const char * *clh_std_error_filename* = `NULL`, const char * *clh_read_restart_filename* = `NULL`, const char * *clh_write_restart_filename* = `NULL`, int *stop_restart_evals* = `0`, bool *pre_run_flag* = `false`)

specify output streams and restart file(s) using external inputs (library mode). Rather than extracting from the command line, pass the std output, std error, read restart, and write restart filenames and the stop restart limit directly. This function only needs to be invoked to specify non-default values [defaults for the filenames are NULL (resulting in no output redirection, no restart read, and default restart write) and 0 for the stop restart limit (resulting in no restart read limit)].

References ParallelLibrary::postRunFlag, ParallelLibrary::preRunFlag, ParallelLibrary::readRestartFilename, ParallelLibrary::runFlag, ParallelLibrary::stdErrorFilename, ParallelLibrary::stdOutputFilename, ParallelLibrary::stopRestartEvals, ParallelLibrary::userModesFlag, and ParallelLibrary::writeRestartFilename.

43.104.3.3 void manage_outputs_restart (const ParallelLevel & *pl*)

manage output streams and restart file(s) (both modes) If the user has specified the use of files for DAKOTA standard output and/or standard error, then bind these filenames to the Cout/Cerr macros. In addition, if concurrent

iterators are to be used, create and tag multiple output streams in order to prevent jumbled output. Manage restart file(s) by processing any incoming evaluations from an old restart file and by setting up the binary output stream for new evaluations. Only master iterator processor(s) read & write restart information. This function must follow init_iterator_communicators so that restart can be managed properly for concurrent iterator strategies. In the case of concurrent iterators, each iterator has its own restart file tagged with iterator number.

References Dakota::abort_handler(), ParallelLibrary::bcast(), ParallelLibrary::checkFlag, Dakota::dakota_cerr, Dakota::dakota_cout, Dakota::data_pairs, ParallelLevel::dedicatedMasterFlag, ParallelLibrary::error_ofstream, ParallelLevel::hubServerCommSize, ParallelLevel::hubServerIntraComm, ParallelLevel::numServers, ParallelLibrary::output_ofstream, ParallelLibrary::postRunFlag, ParallelLibrary::postRunInput, ParallelLibrary::postRunOutput, ParallelLibrary::preRunFlag, ParallelLibrary::preRunInput, ParallelLibrary::preRunOutput, ParallelLibrary::readRestartFilename, ParallelLibrary::runFlag, ParallelLibrary::runInput, ParallelLibrary::runOutput, ParallelLevel::serverCommRank, ParallelLevel::serverId, ParallelLevel::serverMasterFlag, MPIPackBuffer::size(), ParallelLibrary::stdErrorFilename, ParallelLibrary::stdErrorToFile, ParallelLibrary::stdOutputFilename, ParallelLibrary::stdOutputToFile, ParallelLibrary::stopRestartEvals, ParallelLibrary::userModesFlag, ParallelLibrary::worldRank, Dakota::write_restart, and ParallelLibrary::writeRestartFilename.

Referenced by Strategy::init_iterator_parallelism().

43.104.3.4 void close_streams ()

close streams, files, and any other services Close streams associated with manage_outputs and manage_restart and terminate any additional services that may be active.

References Dakota::abort_handler(), ParallelLibrary::currPCIter, Dakota::dakota_cerr, Dakota::dakota_cout, Dakota::dc_ptr_int, ParallelLibrary::error_ofstream, Dakota::mc_ptr_int, ParallelLibrary::output_ofstream, ParallelLevel::serverMasterFlag, ParallelLibrary::stdErrorToFile, ParallelLibrary::stdOutputToFile, and Dakota::write_restart.

Referenced by ParallelLibrary::∼ParallelLibrary().

43.104.3.5 void increment_parallel_configuration () `[inline]`

add a new node to parallelConfigurations and increment currPCIter Called from the ParallelLibrary ctor and from Model::init_communicators(). An increment is performed for each Model initialization except the first (which inherits the world and strategy-iterator parallel levels from the first partial configuration).

References ParallelLibrary::currPCIter, ParallelConfiguration::eaPLIter, ParallelConfiguration::iePLIter, ParallelConfiguration::numParallelLevels, ParallelLibrary::parallelConfigurations, ParallelLibrary::parallelLevels, ParallelConfiguration::siPLIter, ParallelLibrary::worldSize, and ParallelConfiguration::wPLIter.

Referenced by Model::init_communicators(), ParallelLibrary::init_mpi_comm(), and ParallelLibrary::ParallelLibrary().

43.104.3.6 void init_mpi_comm (MPI_Comm *dakota_mpi_comm*) `[private]`

convenience function for initializing from specific comm shared function for initializing based on passed MPI_Comm

References ParallelLibrary::currPLIter, Dakota::Dak_pl, ParallelLibrary::increment_parallel_configuration(),

ParallelLibrary::mpirunFlag, ParallelLibrary::parallelLevels, ParallelLevel::serverCommRank, ParallelLevel::serverCommSize, ParallelLevel::serverIntraComm, Dakota::start_dakota_heartbeat(), ParallelLibrary::startClock, ParallelLibrary::startCPUTime, ParallelLibrary::startMPITime, ParallelLibrary::startWCTime, ParallelLibrary::worldRank, and ParallelLibrary::worldSize.

Referenced by ParallelLibrary::ParallelLibrary().

43.104.3.7 void init_communicators (const ParallelLevel & *parent_pl*, const int & *num_servers*, const int & *procs_per_server*, const int & *max_concurrency*, const int & *asynch_local_concurrency*, const std::string & *default_config*, const std::string & *scheduling_override*) `[private]`

split a parent communicator into child server communicators Split parent communicator into concurrent child server partitions as specified by the passed parameters. This constructs new child intra-communicators and parent-child inter-communicators. This function is called from the Strategy constructor for the concurrent iterator level and from ApplicationInterface::init_communicators() for the concurrent evaluation and concurrent analysis levels.

References ParallelLevel::commSplitFlag, ParallelLibrary::currPCIter, ParallelLibrary::currPLIter, ParallelLevel::dedicatedMasterFlag, ParallelLevel::numServers, ParallelLibrary::parallelLevels, ParallelLevel::procRemainder, ParallelLevel::procsPerServer, ParallelLibrary::resolve_inputs(), ParallelLevel::serverCommRank, ParallelLevel::serverCommSize, ParallelLibrary::split_communicator_dedicated_master(), and ParallelLibrary::split_communicator_peer_partition().

Referenced by ParallelLibrary::init_analysis_communicators(), ParallelLibrary::init_evaluation_communicators(), and ParallelLibrary::init_iterator_communicators().

43.104.3.8 bool resolve_inputs (int & *num_servers*, int & *procs_per_server*, const int & *avail_procs*, int & *proc_remainder*, const int & *max_concurrency*, const int & *capacity_multiplier*, const std::string & *default_config*, const std::string & *scheduling_override*, bool *print_rank*) `[private]`

resolve user inputs into a sensible partitioning scheme This function is responsible for the "auto-configure" intelligence of DAKOTA. It resolves a variety of inputs and overrides into a sensible partitioning configuration for a particular parallelism level. It also handles the general case in which a user's specification request does not divide out evenly with the number of available processors for the level. If num_servers & procs_per_server are both nondefault, then the former takes precedence.

Referenced by ParallelLibrary::init_communicators().

43.104.3.9 void split_filenames (const char * *filenames*, std::string & *input_filename*, std::string & *output_filename*) `[private]`

split a double colon separated pair of filenames (possibly empty) into input and output filename strings Tokenize colon-delimited input and output filenames, returns unchanged strings if tokens not found.

Referenced by ParallelLibrary::manage_run_modes().

The documentation for this class was generated from the following files:

- ParallelLibrary.H
- ParallelLibrary.C

43.105 ParamResponsePair Class Reference

Container class for a variables object, a response object, and an evaluation id.

Public Member Functions

- ParamResponsePair ()

  default constructor

- ParamResponsePair (const Variables &vars, const String &interface_id, const Response &response, bool deep_copy=false)

  alternate constructor for temporaries

- ParamResponsePair (const Variables &vars, const String &interface_id, const Response &response, const int eval_id, bool deep_copy=true)

  standard constructor for history uses

- ParamResponsePair (const ParamResponsePair &pair)

  copy constructor

- ~ParamResponsePair ()

  destructor

- ParamResponsePair & operator= (const ParamResponsePair &pair)

  assignment operator

- void read (std::istream &s)

  read a ParamResponsePair object from an std::istream

- void write (std::ostream &s) const

  write a ParamResponsePair object to an std::ostream

- void read_annotated (std::istream &s)

  read a ParamResponsePair object in annotated format from an std::istream

- void write_annotated (std::ostream &s) const

  write a ParamResponsePair object in annotated format to an std::ostream

- void write_tabular (std::ostream &s) const

  write a ParamResponsePair object in tabular format to an std::ostream

- void read (BiStream &s)

  read a ParamResponsePair object from the binary restart stream

- void write (BoStream &s) const

write a ParamResponsePair object to the binary restart stream

- void read (MPIUnpackBuffer &s)

  read a ParamResponsePair object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a ParamResponsePair object to a packed MPI buffer

- int eval_id () const

  return the evaluation identifier

- const String & interface_id () const

  return the interface identifier from the response object

- const IntStringPair & eval_interface_ids () const

  return the aggregate eval/interface identifier from the response object

- const Variables & prp_parameters () const

  return the parameters object

- const Response & prp_response () const

  return the response object

- void prp_response (const Response &response)

  set the response object

- const ActiveSet & active_set () const

  return the active set object from the response object

- void active_set (const ActiveSet &set)

  set the active set object within the response object

Private Attributes

- Variables prPairParameters

  the set of parameters for the function evaluation

- Response prPairResponse

  the response set for the function evaluation

- IntStringPair evalInterfaceIds

  the evalInterfaceIds aggregate

Friends

- bool operator== (const ParamResponsePair &pair1, const ParamResponsePair &pair2)

  equality operator

- bool operator!= (const ParamResponsePair &pair1, const ParamResponsePair &pair2)

  inequality operator

43.105.1 Detailed Description

Container class for a variables object, a response object, and an evaluation id. ParamResponsePair provides a container class for association of the input for a particular function evaluation (a variables object) with the output from this function evaluation (a response object), along with an evaluation identifier. This container defines the basic unit used in the data_pairs cache, in restart file operations, and in a variety of scheduling algorithm queues. With the advent of STL, replacement of arrays of this class with map<> and pair<> template constructs may be possible (using map<pair<int,String>, pair<Variables,Response> >, for example), assuming that deep copies, I/O, alternate constructors, etc., can be adequately addressed. Boost tuple<> may also be a candidate.

43.105.2 Constructor & Destructor Documentation

43.105.2.1 ParamResponsePair (const Variables & *vars*, const String & *interface_id*, const Response & *response*, bool *deep_copy* = `false`) `[inline]`

alternate constructor for temporaries Uses of this constructor often employ the standard Variables and Response copy constructors to share representations since this constructor is commonly used for search_pairs (which are local instantiations that go out of scope prior to any changes to values; i.e., they are not used for history).

43.105.2.2 ParamResponsePair (const Variables & *vars*, const String & *interface_id*, const Response & *response*, const int *eval_id*, bool *deep_copy* = `true`) `[inline]`

standard constructor for history uses Uses of this constructor often do not share representations since deep copies are used when history mechanisms (e.g., data_pairs and beforeSynchCorePRPQueue) are involved.

43.105.3 Member Function Documentation

43.105.3.1 void read (MPIUnpackBuffer & *s*) `[inline]`

read a ParamResponsePair object from a packed MPI buffer interfaceId is omitted since master processor retains interface ids and communicates asv and response data only with slaves.

References ParamResponsePair::evalInterfaceIds, ParamResponsePair::prPairParameters, and ParamResponsePair::prPairResponse.

43.105.3.2 void write (MPIPackBuffer & *s*) const [inline]

write a ParamResponsePair object to a packed MPI buffer interfaceId is omitted since master processor retains interface ids and communicates asv and response data only with slaves.

References ParamResponsePair::evalInterfaceIds, ParamResponsePair::prPairParameters, and ParamResponsePair::prPairResponse.

43.105.4 Member Data Documentation

43.105.4.1 IntStringPair evalInterfaceIds [private]

the evalInterfaceIds aggregate the function evaluation identifier (assigned from Interface::evalIdCntr) is paired with the interface used to generate the response object. Used in PRPCache id_vars_set_compare to prevent duplicate detection on results from different interfaces. evalInterfaceIds belongs here rather than in Response since some Response objects involve consolidation of several fn evals (e.g., Model::synchronize_derivatives()) that are not, in total, generated by a single interface. The prPair, on the other hand, is used for storage of all low level fn evals that get evaluated in ApplicationInterface::map().

Referenced by ParamResponsePair::eval_id(), ParamResponsePair::eval_interface_ids(), ParamResponsePair::interface_id(), ParamResponsePair::operator=(), Dakota::operator==(), ParamResponsePair::read(), ParamResponsePair::read_annotated(), ParamResponsePair::write(), ParamResponsePair::write_annotated(), and ParamResponsePair::write_tabular().

The documentation for this class was generated from the following files:

- ParamResponsePair.H
- ParamResponsePair.C

43.106 ParamStudy Class Reference

Class for vector, list, centered, and multidimensional parameter studies. Inheritance diagram for ParamStudy::

Public Member Functions

- ParamStudy (Model &model)

  constructor

- ~ParamStudy ()

  destructor

- void pre_run ()

  pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori

- void extract_trends ()

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

- void post_input ()

  read tabular data for post-run mode

- void post_run (std::ostream &s)

  post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way

Private Member Functions

- void sample ()

  performs the parameter study by sampling from a list of points

- void vector_loop ()

  performs the parameter study by sampling along a vector, starting from an initial point followed by numSteps increments along continous/discrete step vectors

- void centered_loop ()

  performs a number of plus and minus offsets for each parameter centered about an initial point

- void multidim_loop ()

  performs a full factorial combination for all intersections defined by a set of multidimensional partitions

- bool distribute_list_of_points (const RealVector &list_of_pts)

  distributes incoming list_of_pts among listCVPoints, listDIVPoints, and listDRVPoints

- bool distribute_step_vector (const RealVector &step_vector)

  distributes incoming step_vector among contStepVector and discStepVector

- void final_point_to_step_vector ()

  compute step vectors from finalPoint, initial points, and numSteps

- void distribute_partitions ()

  compute step vectors from variablePartitions and global bounds

- bool check_num_steps (int num_steps)

  perform error checks on numSteps

- bool check_final_point (const RealVector &final_pt)

  perform error checks on finalPoint

- bool check_steps_per_variable (const IntVector &steps_per_var)

  perform error checks on stepsPerVariable

- bool check_variable_partitions (const UShortArray &partitions)

  perform error checks on variablePartitions

- bool check_finite_bounds ()

  check for finite variable bounds within iteratedModel, as required for computing partitions of finite ranges

- bool check_ranges_sets (int num_steps)

  sanity check for vector parameter study

- bool check_ranges_sets (const IntVector &steps)

  sanity check for centered parameter study

- bool check_sets (const IntVector &steps)

  sanity check for increments along int/real set dimensions

- int truncate (const Real &value) const

  cast Real to int and ensure no resulting change in value

- int integer_step (int range, int num_steps) const

  check for integer remainder and return step

- int index_step (size_t start, size_t end, int num_steps) const

  check for out of bounds and index remainder and return step

- void write_ordered (std::ostream &s, const RealVector &c_vector, const IntVector &di_vector, const RealVector &dr_vector)

  reorder CV/DIV/DRV into standard output order

- void write_ordered (std::ostream &s, const RealVector &c_vector, const IntVector &d_vector)

  reorder CV/DV into standard output order

- void c_step (size_t c_index, int increment, Variables &vars)

  helper function for performing a continuous step in one variable

- void dri_step (size_t d_index, size_t di_index, int increment, Variables &vars)

  helper function for performing a discrete step in an integer range variable

- void dsi_step (size_t d_index, size_t di_index, int increment, const IntSet &values, Variables &vars)

  helper function for performing a discrete step in an integer set variable

- void dsr_step (size_t d_index, size_t dr_index, int increment, const RealSet &values, Variables &vars)

  helper function for performing a discrete step in a real set variable

Private Attributes

- short pStudyType

  internal code for parameter study type: LIST, VECTOR_SV, VECTOR_FP, CENTERED, or MULTIDIM

- size_t numEvals

  total number of parameter study evaluations computed from specification

- RealVectorArray listCVPoints

  array of continuous evaluation points for the list_parameter_study

- IntVectorArray listDIVPoints

  array of discrete int evaluation points for the list_parameter_study

- RealVectorArray listDRVPoints

  array of discrete real evaluation points for the list_parameter_study

- RealVector initialCVPoint

  the continuous starting point for vector and centered parameter studies

- IntVector initialDIVPoint

  the continuous starting point for vector and centered parameter studies

- RealVector initialDRVPoint

  the continuous starting point for vector and centered parameter studies

- RealVector finalPoint

  the ending point for vector_parameter_study (a specification option)

- RealVector contStepVector

  the n-dimensional continuous increment in vector_parameter_study

- IntVector discStepVector

  the n-dimensional discrete increment in vector_parameter_study

- int numSteps

  the number of times stepVector is applied in vector_parameter_study

- IntVector stepsPerVariable

  number of offsets in the plus and the minus direction for each variable in a centered_parameter_study

- UShortArray variablePartitions

  number of partitions for each variable in a multidim_parameter_study

43.106.1 Detailed Description

Class for vector, list, centered, and multidimensional parameter studies. The ParamStudy class contains several algorithms for performing parameter studies of different types. The vector parameter study steps along an n-dimensional vector from an arbitrary initial point to an arbitrary final point in a specified number of steps. The centered parameter study performs a number of plus and minus offsets in each coordinate direction around a center point. A multidimensional parameter study fills an n-dimensional hypercube based on bounds and a specified number of partitions for each dimension. And the list parameter study provides for a user specification of a list of points to evaluate, which allows general parameter investigations not fitting the structure of vector, centered, or multidim parameter studies.

43.106.2 Member Function Documentation

43.106.2.1 void pre_run () `[virtual]`

pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori pre-run phase, which a derived iterator may optionally reimplement; when not present, pre-run is likely integrated into the derived run function. This is a virtual function; when re-implementing, a derived class must call its nearest parent's pre_run(), if implemented, typically _before_ performing its own implementation steps.

Reimplemented from Iterator.

References Dakota::abort_handler(), Analyzer::allHeaders, Analyzer::allVariables, ParamStudy::centered_-loop(), Model::continuous_variables(), ParamStudy::contStepVector, Dakota::copy_data(), Model::current_-variables(), Model::discrete_int_variables(), Model::discrete_real_variables(), ParamStudy::discStepVector, ParamStudy::distribute_partitions(), ParamStudy::final_point_to_step_vector(), ParamStudy::finalPoint, ParamStudy::initialCVPoint, ParamStudy::initialDIVPoint, ParamStudy::initialDRVPoint, Iterator::iteratedModel, ParamStudy::multidim_loop(), ParamStudy::numEvals, ParamStudy::numSteps, Iterator::outputLevel, ParamStudy::pStudyType, ParamStudy::sample(), Variables::shared_data(), ParamStudy::stepsPerVariable, ParamStudy::variablePartitions, ParamStudy::vector_loop(), Dakota::write_data(), and ParamStudy::write_ordered().

43.106.2.2 void post_run (std::ostream & *s*) `[virtual]`

post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way Post-run phase, which a derived iterator may optionally reimplement; when not present, post-run is likely integrated into run. This is a virtual function; when re-implementing, a derived class must call its nearest parent's post_run(), typically _after_ performing its own implementation steps.

Reimplemented from Iterator.

References Analyzer::allResponses, Analyzer::allVariables, SensAnalysisGlobal::compute_correlations(), PStudyDACE::pStudyDACESensGlobal, ParamStudy::pStudyType, and Iterator::subIteratorFlag.

The documentation for this class was generated from the following files:

- ParamStudy.H
- ParamStudy.C

43.107 partial_prp_equality Struct Reference

predicate for comparing ONLY the interfaceId and Vars attributes of PRPair

Public Member Functions

- bool operator() (const ParamResponsePair &database_pr, const ParamResponsePair &search_pr) const
  access operator

43.107.1 Detailed Description

predicate for comparing ONLY the interfaceId and Vars attributes of PRPair

The documentation for this struct was generated from the following file:

- PRPMultiIndex.H

43.108 partial_prp_hash Struct Reference

wrapper to delegate to the ParamResponsePair hash_value function

Public Member Functions

- std::size_t operator() (const ParamResponsePair &prp) const
  access operator

43.108.1 Detailed Description

wrapper to delegate to the ParamResponsePair hash_value function

The documentation for this struct was generated from the following file:

- PRPMultiIndex.H

43.109 PecosApproximation Class Reference

Derived approximation class for global basis polynomials. Inheritance diagram for PecosApproximation::

Public Member Functions

- PecosApproximation ()

  default constructor

- PecosApproximation (const String &approx_type, const UShortArray &approx_order, size_t num_vars, short data_order)

  alternate constructor

- PecosApproximation (ProblemDescDB &problem_db, size_t num_vars)

  standard ProblemDescDB-driven constructor

- ~PecosApproximation ()

  destructor

- void increment_order ()

  increment OrthogPolyApproximation::approxOrder uniformly

- void solution_approach (short soln_approach)

  set pecosBasisApprox.configOptions.expCoeffsSolnApproach

- short solution_approach () const

  get pecosBasisApprox.configOptions.expCoeffsSolnApproach

- void expansion_coefficient_flag (bool coeff_flag)

  set pecosBasisApprox.configOptions.expansionCoeffFlag

- bool expansion_coefficient_flag () const

  get pecosBasisApprox.configOptions.expansionCoeffFlag

- void expansion_gradient_flag (bool grad_flag)

  set pecosBasisApprox.configOptions.expansionGradFlag

- bool expansion_gradient_flag () const

  get pecosBasisApprox.configOptions.expansionGradFlag

- void refinement_control (short refine_cntl)

  set pecosBasisApprox.configOptions.refinementControl

- short refinement_control () const

  get pecosBasisApprox.configOptions.refinementControl

- void vbd_control (short vbd_cntl)

  set pecosBasisApprox.configOptions.vbdControl

- short vbd_control () const

  get pecosBasisApprox.configOptions.vbdControl

- void compute_component_effects ()

  Performs global sensitivity analysis using Sobol' Indices by computing component (main and interaction) effects.

- void compute_total_effects ()

  Performs global sensitivity analysis using Sobol' Indices by computing total effects.

- const Pecos::IntIntMap & sobol_index_map () const

  return polyApproxRep->sobolIndexMap

- const Pecos::RealVector & sobol_indices () const

  return polyApproxRep->sobolIndices

- const Pecos::RealVector & total_sobol_indices () const

  return polyApproxRep->totalSobolIndices

- const Pecos::RealVector & dimension_decay_rates () const

  return OrthogPolyApproximation::decayRates

- void random_variables_key (const Pecos::BoolDeque &random_vars_key)

  set pecosBasisApprox.randomVarsKey

- void integration_iterator (const Iterator &iterator)

  set pecosBasisApprox.driverRep

- void construct_basis (const Pecos::ShortArray &u_types, const Pecos::DistributionParams &dp, const Pecos::BasisConfigOptions &bc_options)

  invoke Pecos::OrthogPolyApproximation::construct_basis()

- void basis_types (const Pecos::ShortArray &basis_types)

  set Pecos::OrthogPolyApproximation::basisTypes

- const Pecos::ShortArray & basis_types () const

  get Pecos::OrthogPolyApproximation::basisTypes

- void polynomial_basis (const std::vector< Pecos::BasisPolynomial > &poly_basis)

  set Pecos::OrthogPolyApproximation::polynomialBasis

- const std::vector< Pecos::BasisPolynomial > & polynomial_basis () const

  get Pecos::OrthogPolyApproximation::polynomialBasis

- void coefficients_norms_flag (bool flag)

  invoke Pecos::OrthogPolyApproximation::coefficients_norms_flag()

- void expansion_terms (size_t terms)

  invoke Pecos::OrthogPolyApproximation::expansion_terms()

- size_t expansion_terms () const

  return Pecos::OrthogPolyApproximation::expansion_terms()

- void allocate_arrays ()

  invoke Pecos::PolynomialApproximation::allocate_arrays()

- Real mean ()

  return the mean of the expansion, treating all variables as random

- Real mean (const Pecos::RealVector &x)

  return the mean of the expansion for a given parameter vector, treating a subset of the variables as random

- const Pecos::RealVector & mean_gradient ()

  return the gradient of the expansion mean for a given parameter vector, treating all variables as random

- const Pecos::RealVector & mean_gradient (const Pecos::RealVector &x, const Pecos::SizetArray &dvv)

  return the gradient of the expansion mean for a given parameter vector and given DVV, treating a subset of the variables as random

- Real variance ()

  return the variance of the expansion, treating all variables as random

- Real variance (const Pecos::RealVector &x)

  return the variance of the expansion for a given parameter vector, treating a subset of the variables as random

- const Pecos::RealVector & variance_gradient ()

  return the gradient of the expansion variance for a given parameter vector, treating all variables as random

- const Pecos::RealVector & variance_gradient (const Pecos::RealVector &x, const Pecos::SizetArray &dvv)

  return the gradient of the expansion variance for a given parameter vector and given DVV, treating a subset of the variables as random

- Real covariance (PecosApproximation *pecos_approx_2)

  return the variance of the expansion, treating all variables as random

- Real covariance (const Pecos::RealVector &x, PecosApproximation *pecos_approx_2)

  return the variance of the expansion, treating a subset of the variables as random

- void compute_moments ()

  compute moments up to the order supported by the Pecos polynomial approximation

- void compute_moments (const Pecos::RealVector &x)

  compute moments in all-variables mode up to the order supported by the Pecos polynomial approximation

- const RealVector & moments () const

  return virtual Pecos::PolynomialApproximation::moments()

- const RealVector & expansion_moments () const

  return Pecos::PolynomialApproximation::expansionMoments

- const RealVector & numerical_moments () const

  return Pecos::PolynomialApproximation::numericalMoments

- Pecos::BasisApproximation & pecos_basis_approximation ()

  return pecosBasisApprox

Protected Member Functions

- Real get_value (const Pecos::RealVector &x)

  retrieve the approximate function value for a given parameter vector

- const Pecos::RealVector & get_gradient (const Pecos::RealVector &x)

  retrieve the approximate function gradient for a given parameter vector

- const Pecos::RealSymMatrix & get_hessian (const Pecos::RealVector &x)

  retrieve the approximate function Hessian for a given parameter vector

- int min_coefficients () const

  return the minimum number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- void build ()

  builds the approximation from scratch

- void rebuild ()

  rebuilds the approximation incrementally

- void pop (bool save_data)

  removes entries from end of SurrogateData::{vars,resp}Data (last points appended)

- void restore ()

  restores state prior to previous append()

- bool restore_available ()

  queries availability of restoration for trial set

- size_t restoration_index ()

  return index of trial set within restorable bookkeeping sets

- void finalize ()

  finalize approximation by applying all remaining trial sets

- size_t finalization_index (size_t i)

  return index of i-th trailing trial set within restorable bookkeeping sets

- void store ()

  store current approximation for later combination

- void combine (short corr_type)

  combine current approximation with previously stored approximation

- void print_coefficients (std::ostream &s) const

  print the coefficient array computed in build()/rebuild()

- const RealVector & approximation_coefficients () const

  return the coefficient array computed by build()/rebuild()

- void approximation_coefficients (const RealVector &approx_coeffs)

  set the coefficient array from external sources, rather than computing with build()/rebuild()

Private Member Functions

- void approx_type_to_basis_type (const String &approx_type, short &basis_type)

  utility to convert Dakota type string to Pecos type enumeration

Private Attributes

- Pecos::BasisApproximation pecosBasisApprox

  the Pecos basis approximation, encompassing OrthogPolyApproximation and InterpPolyApproximation

- Pecos::PolynomialApproximation * polyApproxRep

  convenience pointer to representation

43.109.1 Detailed Description

Derived approximation class for global basis polynomials. The PecosApproximation class provides a global approximation based on basis polynomials. This includes orthogonal polynomials used for polynomial chaos expansions and interpolation polynomials used for stochastic collocation.

43.109.2 Member Function Documentation

43.109.2.1 void build () [inline, protected, virtual]

builds the approximation from scratch This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References PecosApproximation::pecosBasisApprox.

43.109.2.2 void rebuild () [inline, protected, virtual]

rebuilds the approximation incrementally This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References PecosApproximation::pecosBasisApprox.

43.109.2.3 void pop (bool *save_data*) [inline, protected, virtual]

removes entries from end of SurrogateData::{vars,resp}Data (last points appended) This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References PecosApproximation::pecosBasisApprox.

43.109.2.4 void restore () [inline, protected, virtual]

restores state prior to previous append() This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References PecosApproximation::pecosBasisApprox.

43.109.2.5 void finalize () [inline, protected, virtual]

finalize approximation by applying all remaining trial sets This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References PecosApproximation::pecosBasisApprox.

The documentation for this class was generated from the following files:

- PecosApproximation.H
- PecosApproximation.C

43.110 ProblemDescDB Class Reference

The database containing information parsed from the DAKOTA input file. Inheritance diagram for ProblemDescDB::

Public Member Functions

- ProblemDescDB ()

  default constructor

- ProblemDescDB (ParallelLibrary &parallel_lib)

  standard constructor

- ProblemDescDB (const ProblemDescDB &db)

  copy constructor

- ~ProblemDescDB ()

  destructor

- ProblemDescDB operator= (const ProblemDescDB &db)

  assignment operator

- void manage_inputs (CommandLineHandler &cmd_line_handler)

  invokes manage_inputs(const char, ...) using the dakota input filename passed with the "-input" option on the DAKOTA command line. This is the normal API employed in main.C.*

- void manage_inputs (const char *dakota_input_file, const char *parser_options=NULL, void(*callback)(void *)=NULL, void *callback_data=NULL)

  invokes parse_inputs() to populate the problem description database and execute any callback function, broadcast() to propagate DB data to all processors, and post_process() to construct default variables/response vectors. This is an alternate API used by the file parsing mode in library_mode.C.

- void parse_inputs (const char *dakota_input_file, const char *parser_options=NULL, void(*callback)(void *)=NULL, void *callback_data=NULL)

  parses the input file and populates the problem description database. This function reads from the dakota input filename passed in and allows subsequent modifications to be done by a callback function. This API is used by the mixed mode option in library_mode.C since it allows broadcast() and post_process() to be deferred until all inputs have been provided.

- void check_input ()

verifies that there is at least one of each of the required keywords in the dakota input file. Used by parse_inputs().

- void broadcast ()

  invokes send_db_buffer() and receive_db_buffer() to broadcast DB data across the processor allocation. Used by manage_inputs().

- void post_process ()

  post-processes the (minimal) input specification to assign default variables/responses specification arrays. Used by manage_inputs().

- void lock ()

  Locks the database in order to prevent data access when the list nodes may not be set properly. Unlocked by a set nodes operation.

- void unlock ()

  Explicitly unlocks the database. Use with care.

- void set_db_list_nodes (const String &method_tag)

  set dataMethodIter based on a method identifier string to activate a particular method specification in dataMethodList and use pointers from this method specification to set all other list iterators.

- void set_db_list_nodes (const size_t &method_index)

  set dataMethodIter based on an index within dataMethodList to activate a particular method specification and use pointers from this method specification to set all other list iterators.

- void resolve_top_method ()

  For a (default) strategy lacking a method pointer, this function is used to determine which of several potential method specifications corresponds to the top method and then sets the list nodes accordingly.

- void set_db_method_node (const String &method_tag)

  set dataMethodIter based on a method identifier string to activate a particular method specification (only).

- void set_db_method_node (const size_t &method_index)

  set dataMethodIter based on an index within dataMethodList to activate a particular method specification (only).

- size_t get_db_method_node ()

  return the index of the active node in dataMethodList

- void set_db_model_nodes (const String &model_tag)

  set the model list iterators (dataModelIter, dataVariablesIter, dataInterfaceIter, and dataResponsesIter) based on the model identifier string

- void set_db_model_nodes (const size_t &model_index)

  set the model list iterators (dataModelIter, dataVariablesIter, dataInterfaceIter, and dataResponsesIter) based on an index within dataModelList

- size_t get_db_model_node ()

return the index of the active node in dataModelList

- void set_db_variables_node (const String &variables_tag)
  set dataVariablesIter based on the variables identifier string
- void set_db_interface_node (const String &interface_tag)
  set dataInterfaceIter based on the interface identifier string
- void set_db_responses_node (const String &responses_tag)
  set dataResponsesIter based on the responses identifier string
- ParallelLibrary & parallel_library () const
  return the parallelLib reference
- IteratorList & iterator_list ()
  return a list of all Iterator objects that have been instantiated
- ModelList & model_list ()
  return a list of all Model objects that have been instantiated
- VariablesList & variables_list ()
  return a list of all Variables objects that have been instantiated
- InterfaceList & interface_list ()
  return a list of all Interface objects that have been instantiated
- ResponseList & response_list ()
  return a list of all Response objects that have been instantiated
- const RealVector & get_rdv (const String &entry_name) const
  get a RealVector out of the database based on an identifier string
- const IntVector & get_idv (const String &entry_name) const
  get an IntVector out of the database based on an identifier string
- const UShortArray & get_dusa (const String &entry_name) const
  get an UShortArray out of the database based on an identifier string
- const RealSymMatrix & get_rsdm (const String &entry_name) const
  get a RealSymMatrix out of the database based on an identifier string
- const RealVectorArray & get_rdva (const String &entry_name) const
  get a RealVectorArray out of the database based on an identifier string
- const IntList & get_dil (const String &entry_name) const
  get an IntList out of the database based on an identifier string

- const IntSet & get_dis (const String &entry_name) const
  get an IntSet out of the database based on an identifier string
- const IntSetArray & get_disa (const String &entry_name) const
  get an IntSetArray out of the database based on an identifier string
- const RealSetArray & get_drsa (const String &entry_name) const
  get a RealSetArray out of the database based on an identifier string
- const StringArray & get_dsa (const String &entry_name) const
  get a StringArray out of the database based on an identifier string
- const String2DArray & get_ds2a (const String &entry_name) const
  get a String2DArray out of the database based on an identifier string
- const String & get_string (const String &entry_name) const
  get a String out of the database based on an identifier string
- const Real & get_real (const String &entry_name) const
  get a Real out of the database based on an identifier string
- int get_int (const String &entry_name) const
  get an int out of the database based on an identifier string
- short get_short (const String &entry_name) const
  get a short out of the database based on an identifier string
- unsigned short get_ushort (const String &entry_name) const
  get an unsigned short out of the database based on an identifier string
- size_t get_sizet (const String &entry_name) const
  get a size_t out of the database based on an identifier string
- bool get_bool (const String &entry_name) const
  get a bool out of the database based on an identifier string
- void ** get_voidss (const String &entry_name) const
  *for getting a void**, e.g., &dlLib*
- void insert_node (const DataStrategy &data_strategy)
  set the DataStrategy object
- void insert_node (const DataMethod &data_method)
  add a DataMethod object to the dataMethodList

- void insert_node (const DataModel &data_model)

  add a DataModel object to the dataModelList

- void insert_node (DataVariables &data_variables)

  add a DataVariables object to the dataVariablesList

- void insert_node (const DataInterface &data_interface)

  add a DataInterface object to the dataInterfaceList

- void insert_node (const DataResponses &data_responses)

  add a DataResponses object to the dataResponsesList

- void set (const String &entry_name, const RealVector &rdv)

  set a RealVector within the database based on an identifier string

- void set (const String &entry_name, const IntVector &idv)

  set an IntVector within the database based on an identifier string

- void set (const String &entry_name, const RealSymMatrix &rsdm)

  set a RealMatrix within the database based on an identifier string

- void set (const String &entry_name, const RealVectorArray &rdva)

  set a RealVectorArray within the database based on an identifier string

- void set (const String &entry_name, const StringArray &dsa)

  set a StringArray within the database based on an identifier string

- bool is_null () const

  function to check dbRep (does this envelope contain a letter)

Protected Member Functions

- ProblemDescDB (BaseConstructor, ParallelLibrary &parallel_lib)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- virtual void derived_parse_inputs (const char ∗dakota_input_file, const char ∗parser_options)

  derived class specifics within parse_inputs()

- virtual void derived_broadcast ()

  derived class specifics within broadcast()

- virtual void derived_post_process ()

  derived class specifics within post_process()

Protected Attributes

- DataStrategy strategySpec

  the strategy specification (only one allowed) resulting from a call to strategy_kwhandler() or insert_node()

- std::list< DataMethod > dataMethodList

  list of method specifications, one for each call to method_kwhandler() or insert_node()

- std::list< DataModel > dataModelList

  list of model specifications, one for each call to model_kwhandler() or insert_node()

- std::list< DataVariables > dataVariablesList

  list of variables specifications, one for each call to variables_kwhandler() or insert_node()

- std::list< DataInterface > dataInterfaceList

  list of interface specifications, one for each call to interface_kwhandler() or insert_node()

- std::list< DataResponses > dataResponsesList

  list of responses specifications, one for each call to responses_kwhandler() or insert_node()

- size_t strategyCntr

  counter for strategy specifications used in check_input

Private Member Functions

- const Iterator & get_iterator (Model &model)

  retrieve an existing Iterator, if it exists, or instantiate a new one

- const Model & get_model ()

  retrieve an existing Model, if it exists, or instantiate a new one

- const Variables & get_variables ()

  retrieve an existing Variables, if it exists, or instantiate a new one

- const Interface & get_interface ()

  retrieve an existing Interface, if it exists, or instantiate a new one

- const Response & get_response (const Variables &vars)

  retrieve an existing Response, if it exists, or instantiate a new one

- ProblemDescDB * get_db (ParallelLibrary &parallel_lib)

  Used by the envelope constructor to instantiate the correct letter class.

- void send_db_buffer ()

MPI send of a large buffer containing strategySpec and all objects in dataMethodList, dataModelList, dataVariablesList, dataInterfaceList, and dataResponsesList. Used by manage_inputs().

- void receive_db_buffer ()

  MPI receive of a large buffer containing strategySpec and all objects in dataMethodList, dataModelList, dataVariablesList, dataInterfaceList, and dataResponsesList. Used by manage_inputs().

Private Attributes

- ParallelLibrary & parallelLib

  reference to the parallel_lib object passed from main

- std::list< DataMethod >::iterator dataMethodIter

  iterator identifying the active list node in dataMethodList

- std::list< DataModel >::iterator dataModelIter

  iterator identifying the active list node in dataModelList

- std::list< DataVariables >::iterator dataVariablesIter

  iterator identifying the active list node in dataVariablesList

- std::list< DataInterface >::iterator dataInterfaceIter

  iterator identifying the active list node in dataInterfaceList

- std::list< DataResponses >::iterator dataResponsesIter

  iterator identifying the active list node in dataResponsesList

- IteratorList iteratorList

  list of iterator objects, one for each method specification

- ModelList modelList

  list of model objects, one for each model specification

- VariablesList variablesList

  list of variables objects, one for each variables specification

- InterfaceList interfaceList

  list of interface objects, one for each interface specification

- ResponseList responseList

  list of response objects, one for each responses specification

- bool methodDBLocked

  prevents use of get_<type> retrieval and set_<type> update functions prior to setting the list node for the active method specification

- bool modelDBLocked

  prevents use of get_<type> retrieval and set_<type> update functions prior to setting the list node for the active model specification

- bool variablesDBLocked

  prevents use of get_<type> retrieval and set_<type> update functions prior to setting the list node for the active variables specification

- bool interfaceDBLocked

  prevents use of get_<type> retrieval and set_<type> update functions prior to setting the list node for the active interface specification

- bool responsesDBLocked

  prevents use of get_<type> retrieval and set_<type> update functions prior to setting the list node for the active responses specification

- ProblemDescDB * dbRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing dbRep

Friends

- class Model

  Model requires access to get_variables() and get_response().

- class SingleModel

  SingleModel requires access to get_interface().

- class HierarchSurrModel

  HierarchSurrModel requires access to get_model().

- class DataFitSurrModel

  DataFitSurrModel requires access to get_iterator() and get_model().

- class NestedModel

  NestedModel requires access to get_interface(), get_response(), get_iterator(), and get_model().

- class Strategy

  Strategy requires access to get_iterator().

- class SingleMethodStrategy

  SingleMethodStrategy requires access to get_model().

- class HybridStrategy

  HybridStrategy requires access to get_model().

- class SequentialHybridStrategy

  SequentialStrategy requires access to get_iterator().

- class ConcurrentStrategy

  ConcurrentStrategy requires access to get_model().

- class SurrBasedLocalMinimizer

  SurrBasedLocalMinimizer requires access to get_iterator().

- class SurrBasedGlobalMinimizer

  SurrBasedGlobalMinimizer requires access to get_iterator().

43.110.1 Detailed Description

The database containing information parsed from the DAKOTA input file. The ProblemDescDB class is a database for DAKOTA input file data that is populated by a parser defined in a derived class. When the parser reads a complete keyword, it populates a data class object (DataStrategy, DataMethod, DataVariables, DataInterface, or DataResponses) and, for all cases except strategy, appends the object to a linked list (dataMethodList, dataVariablesList, dataInterfaceList, or dataResponsesList). No strategy linked list is used since only one strategy specification is allowed.

43.110.2 Constructor & Destructor Documentation

43.110.2.1 ProblemDescDB ()

default constructor The default constructor: dbRep is NULL in this case. This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

43.110.2.2 ProblemDescDB (ParallelLibrary & *parallel_lib*)

standard constructor This is the envelope constructor which uses problem_db to build a fully populated db object. It only needs to extract enough data to properly execute get_db(problem_db), since the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived classes.

References Dakota::abort_handler(), ProblemDescDB::dbRep, and ProblemDescDB::get_db().

43.110.2.3 ProblemDescDB (const ProblemDescDB & *db*)

copy constructor Copy constructor manages sharing of dbRep and incrementing of referenceCount.

References ProblemDescDB::dbRep, and ProblemDescDB::referenceCount.

43.110.2.4 ~ProblemDescDB ()

destructor Destructor decrements referenceCount and only deletes dbRep when referenceCount reaches zero.

References Dakota::Dak_pddb, ProblemDescDB::dbRep, and ProblemDescDB::referenceCount.

43.110.2.5 ProblemDescDB (BaseConstructor, ParallelLibrary & *parallel_lib*) `[protected]`

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_db() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_db() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~ProblemDescDB).

43.110.3 Member Function Documentation

43.110.3.1 ProblemDescDB operator= (const ProblemDescDB & *db*)

assignment operator Assignment operator decrements referenceCount for old dbRep, assigns new dbRep, and increments referenceCount for new dbRep.

References ProblemDescDB::dbRep, and ProblemDescDB::referenceCount.

43.110.3.2 void manage_inputs (CommandLineHandler & *cmd_line_handler*)

invokes manage_inputs(const char*, ...) using the dakota input filename passed with the "-input" option on the DAKOTA command line. This is the normal API employed in main.C. Manage command line inputs using the CommandLineHandler class and parse the input file.

References ProblemDescDB::dbRep, ProblemDescDB::manage_inputs(), ProblemDescDB::parallelLib, GetLongOpt::retrieve(), and ParallelLibrary::world_rank().

Referenced by main(), ProblemDescDB::manage_inputs(), run_dakota(), and run_dakota_parse().

43.110.3.3 void manage_inputs (const char * *dakota_input_file*, const char * *parser_options* = `NULL`, void(*)(void *) *callback* = `NULL`, void * *callback_data* = `NULL`)

invokes parse_inputs() to populate the problem description database and execute any callback function, broadcast() to propagate DB data to all processors, and post_process() to construct default variables/response vectors. This is an alternate API used by the file parsing mode in library_mode.C. Parse the input file, broadcast it to all processors, and post-process the data on all processors.

References ProblemDescDB::broadcast(), ProblemDescDB::dbRep, ProblemDescDB::manage_inputs(), ProblemDescDB::parse_inputs(), and ProblemDescDB::post_process().

43.110.3.4 void parse_inputs (const char * *dakota_input_file*, const char * *parser_options* = NULL, void(*)(void *) *callback* = NULL, void * *callback_data* = NULL)

parses the input file and populates the problem description database. This function reads from the dakota input filename passed in and allows subsequent modifications to be done by a callback function. This API is used by the mixed mode option in library_mode.C since it allows broadcast() and post_process() to be deferred until all inputs have been provided. Parse the input file, execute the callback function (if present), and perform basic checks on keyword counts.

References ProblemDescDB::check_input(), ProblemDescDB::dbRep, ProblemDescDB::derived_parse_inputs(), ProblemDescDB::parallelLib, ProblemDescDB::parse_inputs(), and ParallelLibrary::world_rank().

Referenced by ProblemDescDB::manage_inputs(), ProblemDescDB::parse_inputs(), and run_dakota_mixed().

43.110.3.5 void post_process ()

post-processes the (minimal) input specification to assign default variables/responses specification arrays. Used by manage_inputs(). When using library mode in a parallel application, post_process() should be called on all processors following broadcast() of a minimal problem specification.

References ProblemDescDB::dbRep, and ProblemDescDB::derived_post_process().

Referenced by ProblemDescDB::manage_inputs(), Dakota::run_dakota_data(), and run_dakota_mixed().

43.110.3.6 ProblemDescDB * get_db (ParallelLibrary & *parallel_lib*) [private]

Used by the envelope constructor to instantiate the correct letter class. Initializes dbRep to the appropriate derived type. The standard derived class constructors are invoked.

References Dakota::Dak_pddb.

Referenced by ProblemDescDB::ProblemDescDB().

The documentation for this class was generated from the following files:

- ProblemDescDB.H
- ProblemDescDB.C

43.111 PStudyDACE Class Reference

Base class for managing common aspects of parameter studies and design of experiments methods. Inheritance diagram for PStudyDACE::

Protected Member Functions

- PStudyDACE (Model &model)

  constructor

- PStudyDACE (NoDBBaseConstructor, Model &model)

  alternate constructor for instantiations "on the fly"

- ~PStudyDACE ()

  destructor

- void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- void print_results (std::ostream &s)

  print the final iterator results

- virtual void extract_trends ()=0

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

- void volumetric_quality (int ndim, int num_samples, double *sample_points)

  Calculation of volumetric quality measures.

Protected Attributes

- SensAnalysisGlobal pStudyDACESensGlobal

  initialize statistical post processing

- bool volQualityFlag

flag which specifies evaluation of volumetric quality measures

- bool varBasedDecompFlag

  flag which specifies calculating variance based decomposition sensitivity analysis metrics

Private Attributes

- double chiMeas

  quality measure

- double dMeas

  quality measure

- double hMeas

  quality measure

- double tauMeas

  quality measure

43.111.1 Detailed Description

Base class for managing common aspects of parameter studies and design of experiments methods. The PStudyDACE base class manages common data and functions, such as those involving the best solutions located during the parameter set evaluations or the printing of final results.

43.111.2 Member Function Documentation

43.111.2.1 void run () `[inline, protected, virtual]`

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Iterator.

References Analyzer::bestVarsRespMap, and PStudyDACE::extract_trends().

43.111.2.2 void print_results (std::ostream & *s*) `[protected, virtual]`

print the final iterator results This virtual function provides additional iterator-specific final results outputs beyond the function evaluation summary printed in finalize_run().

Reimplemented from Analyzer.

References PStudyDACE::chiMeas, Analyzer::compactMode, Model::continuous_variable_labels(), SensAnalysisGlobal::correlations_computed(), Model::discrete_int_variable_labels(), Model::discrete_real_variable_labels(), PStudyDACE::dMeas, PStudyDACE::hMeas, Iterator::iteratedModel, Analyzer::numLSqTerms, Analyzer::numObjFns, SensAnalysisGlobal::print_correlations(), Analyzer::print_sobol_indices(), PStudyDACE::pStudyDACESensGlobal, Model::response_labels(), PStudyDACE::tauMeas, PStudyDACE::varBasedDecompFlag, and PStudyDACE::volQualityFlag.

43.111.2.3 void volumetric_quality (int *ndim*, int *num_samples*, double * *sample_points*) [protected]

Calculation of volumetric quality measures. Calculation of volumetric quality measures developed by FSU.

References PStudyDACE::chiMeas, PStudyDACE::dMeas, PStudyDACE::hMeas, and PStudyDACE::tauMeas.

Referenced by FSUDesignCompExp::get_parameter_sets(), and DDACEDesignCompExp::get_parameter_sets().

The documentation for this class was generated from the following files:

- DakotaPStudyDACE.H
- DakotaPStudyDACE.C

43.112 PSUADEDesignCompExp Class Reference

Wrapper class for the PSUADE library. Inheritance diagram for PSUADEDesignCompExp::

Public Member Functions

- PSUADEDesignCompExp (Model &model)

  primary constructor for building a standard DACE iterator

- ~PSUADEDesignCompExp ()

  destructor

- void pre_run ()

  pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori

- void post_input ()

  read tabular data for post-run mode

- void extract_trends ()

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

- void post_run (std::ostream &s)

  post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way

- int num_samples () const

  get the current number of samples

- void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)

  reset sampling iterator to use at least min_samples

- const String & sampling_scheme () const

  return sampling name

- void vary_pattern (bool pattern_flag)

  sets varyPattern in derived classes that support it

- void get_parameter_sets (Model &model)

  Returns one block of samples (ndim ∗ num_samples).

Private Member Functions

- void enforce_input_rules ()

  enforce sanity checks/modifications for the user input specification

Private Attributes

- int samplesSpec

  initial specification of number of samples

- int numSamples

  current number of samples to be evaluated

- const UShortArray & varPartitionsSpec

  number of partitions in each variable direction

- int numPartitions

  number of partitions to pass to PSUADE (levels = partitions + 1)

- bool allDataFlag

  flag which triggers the update of allVars/allResponses for use by Iterator::all_variables() and Iterator::all_responses()

- size_t numDACERuns

  counter for number of run() executions for this object

- bool varyPattern

  flag for generating a sequence of seed values within multiple get_parameter_sets() calls so that the sample sets are not repeated, but are still repeatable

- const int seedSpec

  the user seed specification for the random number generator (allows repeatable results)

- int randomSeed

  current seed for the random number generator

43.112.1 Detailed Description

Wrapper class for the PSUADE library. The PSUADEDesignCompExp class provides a wrapper for PSUADE, a C++ design of experiments library from Lawrence Livermore National Laboratory. Currently this class only includes the PSUADE Morris One-at-a-time (MOAT) method to uniformly sample the parameter space spanned by the active bounds of the current Model. It returns all generated samples and their corresponding responses as well as the best sample found.

43.112.2 Constructor & Destructor Documentation

43.112.2.1 PSUADEDesignCompExp (Model & *model*)

primary constructor for building a standard DACE iterator This constructor is called for a standard iterator built with data from probDescDB.

References Dakota::abort_handler(), Iterator::maxConcurrency, Iterator::methodName, and PSUADEDesignCompExp::numSamples.

43.112.3 Member Function Documentation

43.112.3.1 void pre_run () `[virtual]`

pre-run portion of run_iterator (optional); re-implemented by Iterators which can generate all Variables (parameter sets) a priori pre-run phase, which a derived iterator may optionally reimplement; when not present, pre-run is likely integrated into the derived run function. This is a virtual function; when re-implementing, a derived class must call its nearest parent's pre_run(), if implemented, typically _before_ performing its own implementation steps.

Reimplemented from Iterator.

References PSUADEDesignCompExp::get_parameter_sets(), and Iterator::iteratedModel.

43.112.3.2 void post_run (std::ostream & *s*) `[virtual]`

post-run portion of run_iterator (optional); verbose to print results; re-implemented by Iterators that can read all Variables/Responses and perform final analysis phase in a standalone way Post-run phase, which a derived iterator may optionally reimplement; when not present, post-run is likely integrated into run. This is a virtual function; when re-implementing, a derived class must call its nearest parent's post_run(), typically _after_ performing its own implementation steps.

Reimplemented from Iterator.

References Dakota::abort_handler(), Analyzer::allResponses, Analyzer::allSamples, Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Iterator::iteratedModel, Iterator::numContinuousVars, Iterator::numFunctions, and PSUADEDesignCompExp::numSamples.

43.112.3.3 int num_samples () const [inline, virtual]

get the current number of samples Return current number of evaluation points. Since the calculation of samples, collocation points, etc. might be costly, provide a default implementation here that backs out from the maxConcurrency. May be (is) overridden by derived classes.

Reimplemented from Iterator.

References PSUADEDesignCompExp::numSamples.

43.112.3.4 void enforce_input_rules () [private]

enforce sanity checks/modifications for the user input specification Users may input a variety of quantities, but this function must enforce any restrictions imposed by the sampling algorithms.

References Dakota::abort_handler(), Iterator::methodName, Iterator::numContinuousVars, PSUADEDesignCompExp::numPartitions, PSUADEDesignCompExp::numSamples, and PSUADEDesignCompExp::varPartitionsSpec.

Referenced by PSUADEDesignCompExp::get_parameter_sets().

The documentation for this class was generated from the following files:

- PSUADEDesignCompExp.H
- PSUADEDesignCompExp.C

43.113 RecastBaseConstructor Struct Reference

Dummy struct for overloading constructors used in on-the-fly Model instantiations.

Public Member Functions

- RecastBaseConstructor (int=0)

  C++ structs can have constructors.

43.113.1 Detailed Description

Dummy struct for overloading constructors used in on-the-fly Model instantiations. RecastBaseConstructor is used to overload the constructor used for on-the-fly Model instantiations. Putting this struct here avoids circular dependencies.

The documentation for this struct was generated from the following file:

- global_defs.h

43.114 RecastModel Class Reference

Derived model class which provides a thin wrapper around a sub-model in order to recast the form of its inputs and/or outputs. Inheritance diagram for RecastModel::

Public Member Functions

- RecastModel (const Model &sub_model, const Sizet2DArray &vars_map_indices, const SizetArray &vars_comps_total, bool nonlinear_vars_mapping, void(∗variables_map)(const Variables &recast_vars, Variables &sub_model_vars), void(∗set_map)(const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set), const Sizet2DArray &primary_resp_map_indices, const Sizet2DArray &secondary_resp_map_indices, size_t recast_secondary_offset, const BoolDequeArray &nonlinear_resp_mapping, void(∗primary_resp_map)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response), void(∗secondary_resp_map)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response))

  standard constructor

- RecastModel (const Model &sub_model, const SizetArray &vars_comps_totals, size_t num_recast_primary_fns, size_t num_recast_secondary_fns, size_t recast_secondary_offset)

  alternate constructor

- ∼RecastModel ()

  destructor

- void initialize (const Sizet2DArray &vars_map_indices, bool nonlinear_vars_mapping, void(∗variables_map)(const Variables &recast_vars, Variables &sub_model_vars), void(∗set_map)(const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set), const Sizet2DArray &primary_resp_map_indices, const Sizet2DArray &secondary_resp_map_indices, const BoolDequeArray &nonlinear_resp_mapping, void(∗primary_resp_map)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response), void(∗secondary_resp_map)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response))

  completes initialization of the RecastModel after alternate construction

- void submodel_supports_derivative_estimation (bool sed_flag)

  override the submodel's derivative estimation behavior

Protected Member Functions

- void derived_compute_response (const ActiveSet &set)

  portion of compute_response() specific to RecastModel (forward to subModel.compute_response())

- void derived_asynch_compute_response (const ActiveSet &set)

  portion of asynch_compute_response() specific to RecastModel (forward to subModel.asynch_compute_response())

- const IntResponseMap & derived_synchronize ()

  portion of synchronize() specific to RecastModel (forward to subModel.synchronize())

- const IntResponseMap & derived_synchronize_nowait ()

  portion of synchronize_nowait() specific to RecastModel (forward to subModel.synchronize_nowait())

- Iterator & subordinate_iterator ()

  return sub-iterator, if present, within subModel

- Model & subordinate_model ()

  return subModel

- Model & surrogate_model ()

  return surrogate model, if present, within subModel

- Model & truth_model ()

  return truth model, if present, within subModel

- void derived_subordinate_models (ModelList &ml, bool recurse_flag)

  add subModel to list and recurse into subModel

- void update_from_subordinate_model (bool recurse_flag=true)

  pass request to subModel if recursing and then update from it

- Interface & interface ()

  return subModel interface

- void primary_response_fn_weights (const RealVector &wts, bool recurse_flag=true)

  set the relative weightings for multiple objective functions or least squares terms and optionally recurses into subModel

- void surrogate_function_indices (const IntSet &surr_fn_indices)

  update the subModel's surrogate response function indices (DataFitSurrModel::surrogateFnIndices)

- void surrogate_response_mode (bool mode)

  update the subModel's surrogate response mode (SurrogateModel::responseMode)

- void build_approximation ()

builds the subModel approximation

- bool build_approximation (const Variables &vars, const IntResponsePair &response_pr)

  builds the subModel approximation

- void update_approximation (bool rebuild_flag)

  replaces data in the subModel approximation

- void update_approximation (const Variables &vars, const IntResponsePair &response_pr, bool rebuild_-flag)

  replaces data in the subModel approximation

- void update_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map, bool rebuild_flag)

  replaces data in the subModel approximation

- void append_approximation (bool rebuild_flag)

  appends data to the subModel approximation

- void append_approximation (const Variables &vars, const IntResponsePair &response_pr, bool rebuild_-flag)

  appends data to the subModel approximation

- void append_approximation (const VariablesArray &vars_array, const IntResponseMap &resp_map, bool rebuild_flag)

  appends data to the subModel approximation

- void pop_approximation (bool save_surr_data)

  remove the previous data set addition to a surrogate (e.g., due to a previous append_approximation() call); flag manages storing of surrogate data for use in a subsequent restore_approximation()

- void restore_approximation ()

  restore a previous approximation data state within a surrogate

- bool restore_available ()

  query for whether a trial increment is restorable within a surrogate

- void finalize_approximation ()

  finalize an approximation by applying all previous trial increments

- void store_approximation ()

  move the current approximation into storage for later combination

- void combine_approximation (short corr_type)

  combine the current approximation with one previously stored

- std::vector< Approximation > & approximations ()

retrieve the set of Approximations from the subModel

- const RealVectorArray & approximation_coefficients ()

  retrieve the approximation coefficients from the subModel

- void approximation_coefficients (const RealVectorArray &approx_coeffs)

  set the approximation coefficients within the subModel

- const RealVector & approximation_variances (const RealVector &c_vars)

  retrieve the approximation variances from the subModel

- const Pecos::SurrogateData & approximation_data (size_t index)

  retrieve the approximation data from the subModel

- void component_parallel_mode (short mode)

  RecastModel only supports parallelism in subModel, so this virtual function redefinition is simply a sanity check.

- String local_eval_synchronization ()

  return subModel local synchronization setting

- int local_eval_concurrency ()

  return subModel local evaluation concurrency

- bool derived_master_overload () const

  flag which prevents overloading the master with a multiprocessor evaluation (request forwarded to subModel)

- void derived_init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set up RecastModel for parallel operations (request forwarded to subModel)

- void derived_init_serial ()

  set up RecastModel for serial operations (request forwarded to subModel).

- void derived_set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set active parallel configuration within subModel

- void derived_free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  deallocate communicator partitions for the RecastModel (request forwarded to subModel)

- void serve ()

  Service subModel job requests received from the master. Completes when a termination message is received from stop_servers().

- void stop_servers ()

  executed by the master to terminate subModel server operations when RecastModel iteration is complete.

- void inactive_view (short view, bool recurse_flag=true)

update the Model's inactive view based on higher level (nested) context and optionally recurse into subModel

- const String & interface_id () const

  return the subModel interface identifier

- int evaluation_id () const

  return the current evaluation id for the RecastModel (request forwarded to subModel)

- void set_evaluation_reference ()

  set the evaluation counter reference points for the RecastModel (request forwarded to subModel)

- void fine_grained_evaluation_counters ()

  request fine-grained evaluation reporting within subModel

- void print_evaluation_summary (std::ostream &s, bool minimal_header=false, bool relative_count=true) const

  print the evaluation summary for the RecastModel (request forwarded to subModel)

Private Member Functions

- void initialize_data_from_submodel ()

  code shared among constructors to initialize base class data from submodel

- void set_mapping (const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set)

  Uses recastFnMap to convert incoming recast_set from iterator into sub_model_set for use with subModel.

- void update_from_sub_model ()

  update current variables/labels/bounds/targets from subModel

Private Attributes

- Model subModel

  the sub-model underlying the function pointers

- Sizet2DArray varsMapIndices

  For each subModel variable, identifies the indices of the recast variables used to define it (maps RecastModel variables to subModel variables; data is packed with only the variable indices employed rather than a sparsely filled N_sm x N_r matrix).

- bool nonlinearVarsMapping

  boolean set to true if the variables mapping involves a nonlinear transformation. Used in set_mapping() to manage the requirement for gradients within the Hessian transformations. This does not require a BoolDeque for each individual variable, since response gradients and Hessians are managed per function, not per variable.

- bool respMapping

  set to true if non-NULL primaryRespMapping or secondaryRespMapping are supplied

- Sizet2DArray primaryRespMapIndices

  For each recast primary function, identifies the indices of the subModel functions used to define it (maps subModel response to RecastModel Response).

- Sizet2DArray secondaryRespMapIndices

  For each recast secondary function, identifies the indices of the subModel functions used to define it (maps subModel response to RecastModel response).

- BoolDequeArray nonlinearRespMapping

  array of BoolDeques, one for each recast response function. Each BoolDeque defines which subModel response functions contribute to the recast function using a nonlinear mapping. Used in set_mapping() to augment the subModel function value/gradient requirements.

- IntActiveSetMap recastSetMap

  map of recast active set passed to derived_asynch_compute_response(). Needed for currentResponse update in synchronization routines.

- IntVariablesMap recastVarsMap

  map of recast variables used by derived_asynch_compute_response(). Needed for primaryRespMapping() and secondaryRespMapping() in synchronization routines.

- IntVariablesMap subModelVarsMap

  map of subModel variables used by derived_asynch_compute_response(). Needed for primaryRespMapping() and secondaryRespMapping() in synchronization routines.

- IntResponseMap recastResponseMap

  map of recast responses used by RecastModel::derived_synchronize() and RecastModel::derived_synchronize_-nowait()

- void(∗ variablesMapping)(const Variables &recast_vars, Variables &sub_model_vars)

  holds pointer for variables mapping function passed in ctor/initialize

- void(∗ setMapping)(const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_-set)

  holds pointer for set mapping function passed in ctor/initialize

- void(∗ primaryRespMapping)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  holds pointer for primary response mapping function passed in ctor/initialize

- void(∗ secondaryRespMapping)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response)

  holds pointer for secondary response mapping function passed in ctor/initialize

43.114.1 Detailed Description

Derived model class which provides a thin wrapper around a sub-model in order to recast the form of its inputs and/or outputs. The RecastModel class uses function pointers to allow recasting of the subModel input/output into new problem forms. This is currently used to recast SBO approximate subproblems, but can be used for multiobjective, input/output scaling, and other problem modifications in the future.

43.114.2 Constructor & Destructor Documentation

43.114.2.1 RecastModel (const Model & *sub_model*, const Sizet2DArray & *vars_map_indices*, const SizetArray & *vars_comps_totals*, bool *nonlinear_vars_mapping*, void(∗)(const Variables &recast_vars, Variables &sub_model_vars) *variables_map*, void(∗)(const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set) *set_map*, const Sizet2DArray & *primary_resp_map_indices*, const Sizet2DArray & *secondary_resp_map_indices*, size_t *recast_secondary_offset*, const BoolDequeArray & *nonlinear_resp_mapping*, void(∗)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response) *primary_resp_map*, void(∗)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response) *secondary_resp_map*)

standard constructor Default recast model constructor. Requires full definition of the transformation. Parameter vars_comps_totals indicates the number of each type of variable {4 types} x {3 domains} in the recast variable space

References Dakota::abort_handler(), Constraints::copy(), Response::copy(), Variables::copy(), Model::current_response(), Model::current_variables(), Model::currentResponse, Model::currentVariables, Variables::cv(), Response::function_gradients(), Response::function_hessians(), RecastModel::initialize_data_from_submodel(), RecastModel::nonlinearRespMapping, Response::num_functions(), Model::num_functions(), Constraints::num_linear_eq_constraints(), Constraints::num_linear_ineq_constraints(), Constraints::num_nonlinear_eq_constraints(), Constraints::num_nonlinear_ineq_constraints(), Model::numDerivVars, Model::numFns, RecastModel::primaryRespMapIndices, RecastModel::primaryRespMapping, Constraints::reshape(), Response::reshape(), RecastModel::respMapping, RecastModel::secondaryRespMapIndices, RecastModel::secondaryRespMapping, RecastModel::subModel, Model::user_defined_constraints(), Model::userDefinedConstraints, Variables::variables_components_totals(), RecastModel::variablesMapping, and Variables::view().

43.114.2.2 RecastModel (const Model & *sub_model*, const SizetArray & *vars_comps_totals*, size_t *num_recast_primary_fns*, size_t *num_recast_secondary_fns*, size_t *recast_secondary_offset*)

alternate constructor This alternate constructor defers initialization of the function pointers until a separate call to initialize(), and accepts the minimum information needed to construct currentVariables, currentResponse, and userDefinedConstraints. The resulting model is sufficiently complete for passing to an Iterator. Parameter vars_comps_totals indicates the number of each type of variable {4 types} x {3 domains} in the recast variable space

References Constraints::copy(), Response::copy(), Variables::copy(), Model::current_response(), Model::current_variables(), Model::currentResponse, Model::currentVariables, Variables::cv(), Response::function_gradients(), Response::function_hessians(), RecastModel::initialize_data_from_submodel(), Model::num_functions(), Constraints::num_linear_eq_constraints(), Constraints::num_linear_ineq_constraints(), Constraints::num_nonlinear_eq_constraints(), Constraints::num_nonlinear_ineq_constraints(),

Model::numDerivVars, Model::numFns, Constraints::reshape(), Response::reshape(), RecastModel::subModel, Model::user_defined_constraints(), Model::userDefinedConstraints, Variables::variables_components_totals(), and Variables::view().

43.114.3 Member Function Documentation

43.114.3.1 void initialize (const Sizet2DArray & *vars_map_indices*, bool *nonlinear_vars_mapping*, void(∗)(const Variables &recast_vars, Variables &sub_model_vars) *variables_map*, void(∗)(const Variables &recast_vars, const ActiveSet &recast_set, ActiveSet &sub_model_set) *set_map*, const Sizet2DArray & *primary_resp_map_indices*, const Sizet2DArray & *secondary_resp_map_indices*, const BoolDequeArray & *nonlinear_resp_mapping*, void(∗)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response) *primary_resp_map*, void(∗)(const Variables &sub_model_vars, const Variables &recast_vars, const Response &sub_model_response, Response &recast_response) *secondary_resp_map*)

completes initialization of the RecastModel after alternate construction This function is used for late initialization of the recasting functions. It is used in concert with the alternate constructor.

References Dakota::abort_handler(), RecastModel::nonlinearRespMapping, RecastModel::nonlinearVarsMapping, RecastModel::primaryRespMapIndices, RecastModel::primaryRespMapping, RecastModel::respMapping, RecastModel::secondaryRespMapIndices, RecastModel::secondaryRespMapping, RecastModel::setMapping, RecastModel::variablesMapping, and RecastModel::varsMapIndices.

Referenced by LeastSq::LeastSq(), EffGlobalMinimizer::minimize_surrogates_on_model(), NonDLocalReliability::mpp_search(), NonDGlobalReliability::optimize_gaussian_process(), Optimizer::Optimizer(), NonDLocalInterval::quantify_uncertainty(), and NonDGlobalInterval::quantify_uncertainty().

43.114.3.2 void update_from_sub_model () [private]

update current variables/labels/bounds/targets from subModel Update inactive values and labels in currentVariables and inactive bound constraints in userDefinedConstraints from variables and constraints data within subModel.

References Model::continuous_lower_bounds(), Constraints::continuous_lower_bounds(), Model::continuous_upper_bounds(), Constraints::continuous_upper_bounds(), Model::continuous_variable_labels(), Variables::continuous_variable_labels(), Model::continuous_variables(), Variables::continuous_variables(), Model::currentResponse, Model::currentVariables, Model::discrete_design_set_int_values(), Model::discrete_design_set_real_values(), Model::discrete_int_lower_bounds(), Constraints::discrete_int_lower_bounds(), Model::discrete_int_upper_bounds(), Constraints::discrete_int_upper_bounds(), Model::discrete_int_variable_labels(), Variables::discrete_int_variable_labels(), Model::discrete_int_variables(), Variables::discrete_int_variables(), Model::discrete_real_lower_bounds(), Constraints::discrete_real_lower_bounds(), Model::discrete_real_upper_bounds(), Constraints::discrete_real_upper_bounds(), Model::discrete_real_variable_labels(), Variables::discrete_real_variable_labels(), Model::discrete_real_variables(), Variables::discrete_real_variables(), Model::discrete_state_set_int_values(), Model::discrete_state_set_real_values(), Model::discreteDesignSetIntValues, Model::discreteDesignSetRealValues, Model::discreteStateSetIntValues, Model::discreteStateSetRealValues, Model::distParams, Model::distribution_parameters(), Response::function_label(), Model::inactive_continuous_lower_bounds(), Constraints::inactive_continuous_lower_-

bounds(), Model::inactive_continuous_upper_bounds(), Constraints::inactive_continuous_upper_-bounds(), Model::inactive_continuous_variable_labels(), Variables::inactive_continuous_variable_labels(), Model::inactive_continuous_variables(), Variables::inactive_continuous_variables(), Model::inactive_discrete_-int_lower_bounds(), Constraints::inactive_discrete_int_lower_bounds(), Model::inactive_discrete_int_upper_-bounds(), Constraints::inactive_discrete_int_upper_bounds(), Model::inactive_discrete_int_variable_labels(), Variables::inactive_discrete_int_variable_labels(), Model::inactive_discrete_int_variables(), Variables::inactive_-discrete_int_variables(), Model::inactive_discrete_real_lower_bounds(), Constraints::inactive_discrete_real_-lower_bounds(), Model::inactive_discrete_real_upper_bounds(), Constraints::inactive_discrete_real_upper_-bounds(), Model::inactive_discrete_real_variable_labels(), Variables::inactive_discrete_real_variable_labels(), Model::inactive_discrete_real_variables(), Variables::inactive_discrete_real_variables(), Model::linear_-eq_constraint_coeffs(), Constraints::linear_eq_constraint_coeffs(), Model::linear_eq_constraint_targets(), Constraints::linear_eq_constraint_targets(), Model::linear_ineq_constraint_coeffs(), Constraints::linear_ineq_-constraint_coeffs(), Model::linear_ineq_constraint_lower_bounds(), Constraints::linear_ineq_constraint_lower_-bounds(), Model::linear_ineq_constraint_upper_bounds(), Constraints::linear_ineq_constraint_upper_bounds(), Model::nonlinear_eq_constraint_targets(), Constraints::nonlinear_eq_constraint_targets(), Model::nonlinear_-ineq_constraint_lower_bounds(), Constraints::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_-ineq_constraint_upper_bounds(), Constraints::nonlinear_ineq_constraint_upper_bounds(), Model::num_-functions(), Model::num_linear_eq_constraints(), Model::num_linear_ineq_constraints(), Model::num_-nonlinear_eq_constraints(), Constraints::num_nonlinear_eq_constraints(), Model::num_nonlinear_ineq_-constraints(), Constraints::num_nonlinear_ineq_constraints(), Model::numFns, Model::primary_response_-fn_weights(), Model::primaryRespFnWts, RecastModel::primaryRespMapping, Model::response_labels(), RecastModel::secondaryRespMapping, RecastModel::subModel, Model::userDefinedConstraints, and Recast-Model::variablesMapping.

Referenced by RecastModel::update_from_subordinate_model().

The documentation for this class was generated from the following files:

- RecastModel.H
- RecastModel.C

43.115 Response Class Reference

Container class for response functions and their derivatives. Response provides the handle class.

Public Member Functions

- Response ()
  default constructor
- Response (const Variables &vars, const ProblemDescDB &problem_db)
  standard constructor built from problem description database
- Response (const ActiveSet &set)
  alternate constructor using limited data
- Response (const Response &response)
  copy constructor
- ~Response ()
  destructor
- Response operator= (const Response &response)
  assignment operator
- size_t num_functions () const
  return the number of response functions
- const ActiveSet & active_set () const
  return the active set
- void active_set (const ActiveSet &set)
  set the active set
- const ShortArray & active_set_request_vector () const
  return the active set request vector
- void active_set_request_vector (const ShortArray &asrv)
  set the active set request vector
- const SizetArray & active_set_derivative_vector () const
  return the active set derivative vector
- void active_set_derivative_vector (const SizetArray &asdv)
  set the active set derivative vector

- const String & responses_id () const

  return the response identifier

- const String & function_label (const size_t &i) const

  return a response function identifier string

- const StringArray & function_labels () const

  return the response function identifier strings

- void function_label (const String &label, const size_t &i)

  set a response function identifier string

- void function_labels (const StringArray &labels)

  set the response function identifier strings

- const Real & function_value (const size_t &i) const

  return a function value

- Real & function_value_view (const size_t &i)

  return a "view" of a function value for updating in place

- const RealVector & function_values () const

  return all function values

- RealVector function_values_view ()

  return all function values as a view for updating in place

- void function_value (const Real &function_val, const size_t &i)

  set a function value

- void function_values (const RealVector &function_vals)

  set all function values

- const Real * function_gradient (const int &i) const

  *return the i-th function gradient as a const Real**

- RealVector function_gradient_view (const int &i) const

  return the i-th function gradient as a SerialDenseVector Teuchos::View (shallow copy) for updating in place

- RealVector function_gradient_copy (const int &i) const

  return the i-th function gradient as a SerialDenseVector Teuchos::Copy (deep copy)

- const RealMatrix & function_gradients () const

  return all function gradients

- RealMatrix function_gradients_view ()

return all function gradients as a view for updating in place

- void function_gradient (const RealVector &function_grad, const int &i)

  set a function gradient

- void function_gradients (const RealMatrix &function_grads)

  set all function gradients

- const RealSymMatrix & function_hessian (const size_t &i) const

  return the i-th function Hessian

- RealSymMatrix function_hessian_view (const size_t &i)

  return the i-th function Hessian as a Teuchos::View (shallow copy) for updating in place

- const RealSymMatrixArray & function_hessians () const

  return all function Hessians

- RealSymMatrixArray function_hessians_view ()

  return all function Hessians as Teuchos::Views (shallow copies) for updating in place

- void function_hessian (const RealSymMatrix &function_hessian, const size_t &i)

  set a function Hessian

- void function_hessians (const RealSymMatrixArray &function_hessians)

  set all function Hessians

- void read (std::istream &s)

  read a response object from an std::istream

- void write (std::ostream &s) const

  write a response object to an std::ostream

- void read_annotated (std::istream &s)

  read a response object in annotated format from an std::istream

- void write_annotated (std::ostream &s) const

  write a response object in annotated format to an std::ostream

- void read_tabular (std::istream &s)

  read responseRep::functionValues in tabular format from an std::istream

- void write_tabular (std::ostream &s) const

  write responseRep::functionValues in tabular format to an std::ostream

- void read (BiStream &s)

  read a response object from the binary restart stream

- void write (BoStream &s) const

  write a response object to the binary restart stream

- void read (MPIUnpackBuffer &s)

  read a response object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a response object to a packed MPI buffer

- Response copy () const

  a deep copy for use in history mechanisms

- int data_size ()

  handle class forward to corresponding body class member function

- void read_data (double *response_data)

  handle class forward to corresponding body class member function

- void write_data (double *response_data)

  handle class forward to corresponding body class member function

- void overlay (const Response &response)

  handle class forward to corresponding body class member function

- void update (const Response &response)

  Used in place of operator= when only results data updates are desired (functionValues/functionGradients/functionHessians are updated, ASV/labels/id's/etc. are not). Care is taken to allow different derivative array sizing between the two response objects.

- void update (const RealVector &source_fn_vals, const RealMatrix &source_fn_grads, const RealSymMatrixArray &source_fn_hessians, const ActiveSet &source_set)

  Overloaded form which allows update from components of a response object. Care is taken to allow different derivative array sizing.

- void update_partial (size_t start_index_target, size_t num_items, const Response &response, size_t start_index_source)

  partial update of this response object from another response object. The response objects may have different numbers of response functions.

- void update_partial (size_t start_index_target, size_t num_items, const RealVector &source_fn_vals, const RealMatrix &source_fn_grads, const RealSymMatrixArray &source_fn_hessians, const ActiveSet &source_set, size_t start_index_source)

  Overloaded form which allows partial update from components of a response object. The response objects may have different numbers of response functions.

- void reshape (const size_t &num_fns, const size_t &num_params, bool grad_flag, bool hess_flag)

rehapes response data arrays

- void reset ()

  handle class forward to corresponding body class member function

- void reset_inactive ()

  handle class forward to corresponding body class member function

- bool is_null () const

  function to check responseRep (does this handle contain a body)

Private Attributes

- ResponseRep * responseRep

  pointer to the body (handle-body idiom)

Friends

- bool operator== (const Response &resp1, const Response &resp2)

  equality operator

- bool operator!= (const Response &resp1, const Response &resp2)

  inequality operator

43.115.1 Detailed Description

Container class for response functions and their derivatives. Response provides the handle class. The Response class is a container class for an abstract set of functions (functionValues) and their first (functionGradients) and second (functionHessians) derivatives. The functions may involve objective and constraint functions (optimization data set), least squares terms (parameter estimation data set), or generic response functions (uncertainty quantification data set). It is not currently part of a class hierarchy, since the abstraction has been sufficiently general and has not required specialization. For memory efficiency, it employs the "handle-body idiom" approach to reference counting and representation sharing (see Coplien "Advanced C++", p. 58), for which Response serves as the handle and ResponseRep serves as the body.

43.115.2 Constructor & Destructor Documentation

43.115.2.1 Response ()

default constructor Need a populated problem description database to build a meaningful Response object, so set the responseRep=NULL in default constructor for efficiency. This then requires a check on NULL in the copy constructor, assignment operator, and destructor.

The documentation for this class was generated from the following files:

- DakotaResponse.H
- DakotaResponse.C

43.116 ResponseRep Class Reference

Container class for response functions and their derivatives. ResponseRep provides the body class.

Private Member Functions

- ResponseRep ()

  default constructor

- ResponseRep (const Variables &vars, const ProblemDescDB &problem_db)

  standard constructor built from problem description database

- ResponseRep (const ActiveSet &set)

  alternate constructor using limited data

- ~ResponseRep ()

  destructor

- void read (std::istream &s)

  read a responseRep object from an std::istream

- void write (std::ostream &s) const

  write a responseRep object to an std::ostream

- void read_annotated (std::istream &s)

  read a responseRep object from an std::istream (annotated format)

- void write_annotated (std::ostream &s) const

  write a responseRep object to an std::ostream (annotated format)

- void read_tabular (std::istream &s)

  read functionValues from an std::istream (tabular format)

- void write_tabular (std::ostream &s) const

  write functionValues to an std::ostream (tabular format)

- void read (BiStream &s)

  read a responseRep object from a binary stream

- void write (BoStream &s) const

  write a responseRep object to a binary stream

- void read (MPIUnpackBuffer &s)

  read a responseRep object from a packed MPI buffer

- void write (MPIPackBuffer &s) const

  write a responseRep object to a packed MPI buffer

- int data_size ()

  return the number of doubles active in response. Used for sizing double∗ response_data arrays passed into read_data and write_data.

- void read_data (double ∗response_data)

  read from an incoming double∗ array

- void write_data (double ∗response_data)

  write to an incoming double∗ array

- void overlay (const Response &response)

  add incoming response to functionValues/Gradients/Hessians

- void update (const RealVector &source_fn_vals, const RealMatrix &source_fn_grads, const RealSymMatrixArray &source_fn_hessians, const ActiveSet &source_set)

  update this response object from components of another response object

- void update_partial (size_t start_index_target, size_t num_items, const RealVector &source_fn_vals, const RealMatrix &source_fn_grads, const RealSymMatrixArray &source_fn_hessians, const ActiveSet &source_set, size_t start_index_source)

  partially update this response object partial components of another response object

- void reshape (const size_t &num_fns, const size_t &num_params, bool grad_flag, bool hess_flag)

  rehapes response data arrays

- void reset ()

  resets all response data to zero

- void reset_inactive ()

  resets all inactive response data to zero

- void active_set_request_vector (const ShortArray &asrv)

  set the active set request vector and verify consistent number of response functions

- void active_set_derivative_vector (const SizetArray &asdv)

  set the active set derivative vector and reshape functionGradients/functionHessians if needed

Private Attributes

- int referenceCount

  number of handle objects sharing responseRep

- RealVector functionValues

  abstract set of response functions

- RealMatrix functionGradients

  first derivatives of the response functions

- RealSymMatrixArray functionHessians

  second derivatives of the response functions

- ActiveSet responseActiveSet

  copy of the ActiveSet used by the Model to generate a Response instance

- StringArray functionLabels

  response function identifiers used to improve output readability

- String responsesId

  response identifier string from the input file

Friends

- class Response

  the handle class can access attributes of the body class directly

- bool operator== (const ResponseRep &rep1, const ResponseRep &rep2)

  equality operator

43.116.1 Detailed Description

Container class for response functions and their derivatives. ResponseRep provides the body class. The ResponseRep class is the "representation" of the response container class. It is the "body" portion of the "handle-body idiom" (see Coplien "Advanced C++", p. 58). The handle class (Response) provides for memory efficiency in management of multiple response objects through reference counting and representation sharing. The body class (ResponseRep) actually contains the response data (functionValues, functionGradients, functionHessians, etc.). The representation is hidden in that an instance of ResponseRep may only be created by Response. Therefore, programmers create instances of the Response handle class, and only need to be aware of the handle/body mechanisms when it comes to managing shallow copies (shared representation) versus deep copies (separate representation used for history mechanisms).

43.116.2 Constructor & Destructor Documentation

43.116.2.1 ResponseRep (const Variables & *vars*, const ProblemDescDB & *problem_db*) `[private]`

standard constructor built from problem description database The standard constructor used by Dakota::ModelRep.

References Dakota::abort_handler(), Variables::continuous_variable_ids(), Variables::cv(), ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, ProblemDescDB::get_sizet(), ProblemDescDB::get_string(), ActiveSet::request_vector(), and ResponseRep::responseActiveSet.

43.116.2.2 ResponseRep (const ActiveSet & *set*) [private]

alternate constructor using limited data Used for building a response object of the correct size on the fly (e.g., by slave analysis servers performing execute() on a local_response). functionLabels is not needed for this purpose since it's not passed in the MPI send/recv buffers. However, NPSOLOptimizer's user-defined functions option uses this constructor to build bestResponseArray.front() and bestResponseArray.front() needs functionLabels for I/O, so construction of functionLabels has been added.

References Dakota::build_labels(), ResponseRep::functionGradients, ResponseRep::functionHessians, and ResponseRep::functionLabels.

43.116.3 Member Function Documentation

43.116.3.1 void read (std::istream & *s*) [private]

read a responseRep object from an std::istream ASCII version of read needs capabilities for capturing data omissions or formatting errors (resulting from user error or asynch race condition) and analysis failures (resulting from nonconvergence, instability, etc.).

References ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, Dakota::re_match(), Dakota::read_col_vector_trans(), ResponseRep::read_data(), ActiveSet::request_vector(), ResponseRep::reset(), and ResponseRep::responseActiveSet.

Referenced by Response::read().

43.116.3.2 void write (std::ostream & *s*) const [private]

write a responseRep object to an std::ostream ASCII version of write.

References Dakota::abort_handler(), Dakota::array_write_annotated(), ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionLabels, ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::responseActiveSet, Dakota::write_col_vector_trans(), ResponseRep::write_data(), and Dakota::write_precision.

Referenced by Response::write().

43.116.3.3 void read_annotated (std::istream & *s*) [private]

read a responseRep object from an std::istream (annotated format) read_annotated() is used for neutral file translation of restart files. Since objects are built solely from this data, annotations are used. This version closely mirrors the BiStream version.

References ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionLabels, ResponseRep::functionValues, Dakota::read_col_vector_trans(), ResponseRep::read_data(), ActiveSet::request_-

vector(), ResponseRep::reset(), ResponseRep::reshape(), ActiveSet::reshape(), and ResponseRep::responseActiveSet.

Referenced by Response::read_annotated().

43.116.3.4 void write_annotated (std::ostream & *s*) const [private]

write a responseRep object to an std::ostream (annotated format) write_annotated() is used for neutral file translation of restart files. Since objects need to be build solely from this data, annotations are used. This version closely mirrors the BoStream version, with the exception of the use of white space between fields.

References Dakota::array_write_annotated(), ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionLabels, ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::responseActiveSet, ActiveSet::write_annotated(), Dakota::write_col_vector_trans(), and ResponseRep::write_data().

Referenced by Response::write_annotated().

43.116.3.5 void read_tabular (std::istream & *s*) [private]

read functionValues from an std::istream (tabular format) read_tabular is used to read functionValues in tabular format. It is currently only used by ApproximationInterfaces in reading samples from a file. There is insufficient data in a tabular file to build complete response objects; rather, the response object must be constructed a priori and then its functionValues can be set.

References ResponseRep::functionValues.

Referenced by Response::read_tabular().

43.116.3.6 void write_tabular (std::ostream & *s*) const [private]

write functionValues to an std::ostream (tabular format) write_tabular is used for output of functionValues in a tabular format for convenience in post-processing/plotting of DAKOTA results.

References ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::responseActiveSet, and Dakota::write_precision.

Referenced by Response::write_tabular().

43.116.3.7 void read (BiStream & *s*) [private]

read a responseRep object from a binary stream Binary version differs from ASCII version in 2 primary ways: (1) it lacks formatting. (2) the Response has not been sized a priori. In reading data from the binary restart file, a ParamResponsePair was constructed with its default constructor which called the Response default constructor. Therefore, we must first read sizing data and resize all of the arrays.

References ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionLabels, ResponseRep::functionValues, Dakota::read_col_vector_trans(), ResponseRep::read_data(), ActiveSet::request_vector(), ResponseRep::reset(), ResponseRep::reshape(), and ResponseRep::responseActiveSet.

43.116.3.8 void write (BoStream & *s*) const [private]

write a responseRep object to a binary stream Binary version differs from ASCII version in 2 primary ways: (1) It lacks formatting. (2) In reading data from the binary restart file, ParamResponsePairs are constructed with their default constructor which calls the Response default constructor. Therefore, we must first write sizing data so that ResponseRep::read(BoStream& s) can resize the arrays.

References ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionLabels, ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::responseActiveSet, Dakota::write_col_vector_trans(), and ResponseRep::write_data().

43.116.3.9 void read (MPIUnpackBuffer & *s*) [private]

read a responseRep object from a packed MPI buffer UnpackBuffer version differs from BiStream version in the omission of functionLabels. Master processor retains labels and interface ids and communicates asv and response data only with slaves.

References ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, Dakota::read_col_vector_trans(), ResponseRep::read_data(), ResponseRep::reset(), ResponseRep::reshape(), and ResponseRep::responseActiveSet.

43.116.3.10 void write (MPIPackBuffer & *s*) const [private]

write a responseRep object to a packed MPI buffer MPIPackBuffer version differs from BoStream version only in the omission of functionLabels. The master processor retains labels and ids and communicates asv and response data only with slaves.

References ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::responseActiveSet, Dakota::write_col_vector_trans(), and ResponseRep::write_data().

43.116.3.11 void update (const RealVector & *source_fn_vals*, const RealMatrix & *source_fn_grads*, const RealSymMatrixArray & *source_fn_hessians*, const ActiveSet & *source_set*) [private]

update this response object from components of another response object Copy function values/gradients/Hessians data _only_. Prevents unwanted overwriting of responseActiveSet, functionLabels, etc. Also, care is taken to account for differences in derivative variable matrix sizing.

References Dakota::abort_handler(), ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::reset_inactive(), and ResponseRep::responseActiveSet.

Referenced by Response::update().

43.116.3.12 void update_partial (size_t *start_index_target*, size_t *num_items*, const RealVector & *source_fn_vals*, const RealMatrix & *source_fn_grads*, const RealSymMatrixArray & *source_fn_hessians*, const ActiveSet & *source_set*, size_t *start_index_source*) [private]

partially update this response object partial components of another response object Copy function values/gradients/Hessians data _only_. Prevents unwanted overwriting of responseActiveSet, functionLabels, etc. Also, care is taken to account for differences in derivative variable matrix sizing.

References Dakota::abort_handler(), ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, ActiveSet::request_vector(), ResponseRep::reset_inactive(), and ResponseRep::responseActiveSet.

Referenced by Response::update_partial().

43.116.3.13 void reshape (const size_t & *num_fns*, const size_t & *num_params*, bool *grad_flag*, bool *hess_flag*) [private]

rehapes response data arrays Reshape functionValues, functionGradients, and functionHessians according to num_fns, num_params, grad_flag, and hess_flag.

References Dakota::build_labels(), ActiveSet::derivative_vector(), ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionLabels, ResponseRep::functionValues, ActiveSet::request_vector(), ActiveSet::reshape(), and ResponseRep::responseActiveSet.

Referenced by ResponseRep::active_set_derivative_vector(), ResponseRep::read(), ResponseRep::read_annotated(), and Response::reshape().

43.116.3.14 void reset () [private]

resets all response data to zero Reset all numerical response data (not labels, ids, or active set) to zero.

References ResponseRep::functionGradients, ResponseRep::functionHessians, and ResponseRep::functionValues.

Referenced by ResponseRep::read(), ResponseRep::read_annotated(), and Response::reset().

43.116.3.15 void reset_inactive () [private]

resets all inactive response data to zero Used to clear out any inactive data left over from previous evaluations.

References ResponseRep::functionGradients, ResponseRep::functionHessians, ResponseRep::functionValues, ActiveSet::request_vector(), and ResponseRep::responseActiveSet.

Referenced by Response::reset_inactive(), ResponseRep::update(), and ResponseRep::update_partial().

43.116.4 Member Data Documentation

43.116.4.1 RealMatrix functionGradients [private]

first derivatives of the response functions the gradient vectors (plural) are column vectors in the matrix (singular) with (row, col) = (variable index, response fn index).

Referenced by ResponseRep::active_set_derivative_vector(), Response::copy(), Response::function_gradient(), Response::function_gradient_copy(), Response::function_gradient_view(), Response::function_gradients(), Response::function_gradients_view(), Dakota::operator==(), ResponseRep::overlay(), ResponseRep::read(), ResponseRep::read_annotated(), ResponseRep::read_data(), ResponseRep::reset(), ResponseRep::reset_inactive(), ResponseRep::reshape(), ResponseRep::ResponseRep(), ResponseRep::update(), ResponseRep::update_partial(), ResponseRep::write(), ResponseRep::write_annotated(), and ResponseRep::write_data().

The documentation for this class was generated from the following files:

- DakotaResponse.H
- DakotaResponse.C

43.117 RichExtrapVerification Class Reference

Class for Richardson extrapolation for code and solution verification. Inheritance diagram for RichExtrapVerification::

Public Member Functions

- RichExtrapVerification (Model &model)

  constructor

- ~RichExtrapVerification ()

  destructor

- void perform_verification ()

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

- void print_results (std::ostream &s)

  print the final iterator results

Private Member Functions

- void estimate_order ()

  perform a single estimation of convOrder using extrapolation()

- void converge_order ()

  iterate using extrapolation() until convOrder stabilizes

- void converge_qoi ()

  iterate using extrapolation() until QOIs stabilize

- void extrapolation (const RealVector &refine_triple, RealMatrix &qoi_triples)

  estimate convOrder from refinement and quantity of interest (QOI) triples

- void extrapolate_result (const RealVector &refine_triple, const RealMatrix &qoi_triples)

predict the converged value based on the convergence rate and the value of Phi

Private Attributes

- short studyType

  internal code for extrapolation study type: ESTIMATE_ORDER, CONVERGE_ORDER, or CONVERGE_QOI

- size_t numFactors

  number of refinement factors defined from active state variables

- RealVector initialCVars

  initial reference values for refinement factors

- size_t factorIndex

  the index of the active factor

- Real refinementRate

  rate of mesh refinement (default = 2.)

- RealMatrix convOrder

  the orders of convergence of the QOIs (numFunctions by numFactors)

- RealMatrix extrapQOI

  the extrapolated value of the QOI (numFunctions by numFactors)

- RealMatrix numErrorQOI

  the numerical uncertainty associated with level of refinement (numFunctions by numFactors)

- RealVector refinementRefPt

  This is a reference point reported for the converged extrapQOI and numErrorQOI. It currently corresponds to the coarsest mesh in the final refinement triple.

43.117.1 Detailed Description

Class for Richardson extrapolation for code and solution verification. The RichExtrapVerification class contains several algorithms for performing Richardson extrapolation.

43.117.2 Member Function Documentation

43.117.2.1 void print_results (std::ostream & *s*) `[virtual]`

print the final iterator results This virtual function provides additional iterator-specific final results outputs beyond the function evaluation summary printed in finalize_run().

Reimplemented from Verification.

References Model::continuous_variable_labels(), RichExtrapVerification::convOrder, Dakota::copy_data(), RichExtrapVerification::extrapQOI, Iterator::iteratedModel, RichExtrapVerification::numErrorQOI, RichExtrapVerification::refinementRate, RichExtrapVerification::refinementRefPt, Model::response_labels(), and Dakota::write_data().

43.117.2.2 void estimate_order () [private]

perform a single estimation of convOrder using extrapolation() This algorithm executes a single refinement triple and returns convergence order estimates.

References RichExtrapVerification::extrapolate_result(), RichExtrapVerification::extrapolation(), RichExtrapVerification::extrapQOI, RichExtrapVerification::factorIndex, RichExtrapVerification::initialCVars, RichExtrapVerification::numErrorQOI, RichExtrapVerification::numFactors, Iterator::numFunctions, RichExtrapVerification::refinementRate, and RichExtrapVerification::refinementRefPt.

Referenced by RichExtrapVerification::perform_verification().

43.117.2.3 void converge_order () [private]

iterate using extrapolation() until convOrder stabilizes This algorithm continues to refine until the convergence order estimate converges.

References Iterator::convergenceTol, RichExtrapVerification::convOrder, Dakota::copy_data(), RichExtrapVerification::extrapolate_result(), RichExtrapVerification::extrapolation(), RichExtrapVerification::extrapQOI, RichExtrapVerification::factorIndex, RichExtrapVerification::initialCVars, Iterator::maxIterations, RichExtrapVerification::numErrorQOI, RichExtrapVerification::numFactors, Iterator::numFunctions, Iterator::outputLevel, RichExtrapVerification::refinementRate, RichExtrapVerification::refinementRefPt, and Dakota::write_data().

Referenced by RichExtrapVerification::perform_verification().

43.117.2.4 void converge_qoi () [private]

iterate using extrapolation() until QOIs stabilize This algorithm continues to refine until the discretization error lies within a prescribed tolerance.

References Iterator::convergenceTol, RichExtrapVerification::extrapolate_result(), RichExtrapVerification::extrapolation(), RichExtrapVerification::extrapQOI, RichExtrapVerification::factorIndex, RichExtrapVerification::initialCVars, Iterator::maxIterations, RichExtrapVerification::numErrorQOI, RichExtrapVerification::numFactors, Iterator::numFunctions, Iterator::outputLevel, RichExtrapVerification::refinementRate, RichExtrapVerification::refinementRefPt, and Dakota::write_data().

Referenced by RichExtrapVerification::perform_verification().

The documentation for this class was generated from the following files:

- RichExtrapVerification.H
- RichExtrapVerification.C

43.118 SensAnalysisGlobal Class Reference

Class for a utility class containing correlation calculations and variance-based decomposition.

Public Member Functions

- SensAnalysisGlobal ()

  constructor

- ~SensAnalysisGlobal ()

  destructor

- void compute_correlations (const VariablesArray &vars_samples, const IntResponseMap &resp_samples)

  computes four correlation matrices for input and output data simple, partial, simple rank, and partial rank

- void compute_correlations (const RealMatrix &vars_samples, const IntResponseMap &resp_samples)

  computes four correlation matrices for input and output data simple, partial, simple rank, and partial rank

- bool correlations_computed () const

  returns corrComputed to indicate whether compute_correlations() has been invoked

- void print_correlations (std::ostream &s, StringMultiArrayConstView cv_labels, StringMultiArrayConstView div_labels, StringMultiArrayConstView drv_labels, const StringArray &resp_labels) const

  prints the correlations computed in compute_correlations()

Private Member Functions

- void simple_corr (RealMatrix &total_data, bool rank_on, const int &num_in)

  computes simple correlations

- void partial_corr (RealMatrix &total_data, bool rank_on, const int &num_in)

  computes partial correlations

Static Private Member Functions

- static bool rank_sort (const int &x, const int &y)

  sort algorithm to compute ranks for rank correlations

Private Attributes

- RealMatrix simpleCorr
  matrix to hold simple raw correlations
- RealMatrix simpleRankCorr
  matrix to hold simple rank correlations
- RealMatrix partialCorr
  matrix to hold partial raw correlations
- RealMatrix partialRankCorr
  matrix to hold partial rank correlations
- size_t numFns
  number of responses
- size_t numVars
  number of inputs
- bool numericalIssuesRaw
  flag indicating numerical issues in partial raw correlation calculations
- bool numericalIssuesRank
  flag indicating numerical issues in partial rank correlation calculations
- bool corrComputed
  flag indictaing whether correlations have been computed

Static Private Attributes

- static RealArray rawData = RealArray()
  array to hold temporary data before sort

43.118.1 Detailed Description

Class for a utility class containing correlation calculations and variance-based decomposition. This class provides code for several of the sampling methods both in the NonD branch and in the PStudyDACE branch. Currently, the utility functions provide global sensitivity analysis through correlation calculations (e.g. simple, partial, rank, raw) as well as variance-based decomposition.

The documentation for this class was generated from the following files:

- SensAnalysisGlobal.H
- SensAnalysisGlobal.C

43.119 SequentialHybridStrategy Class Reference

Strategy for sequential hybrid minimization using multiple optimization and nonlinear least squares methods on multiple models of varying fidelity. Inheritance diagram for SequentialHybridStrategy::

Public Member Functions

- SequentialHybridStrategy (ProblemDescDB &problem_db)

  constructor

- ~SequentialHybridStrategy ()

  destructor

Protected Member Functions

- void run_strategy ()

  Performs the hybrid minimization strategy by executing multiple iterators on different models of varying fidelity.

- const Variables & variables_results () const

  return the final solution from selectedIterators (variables)

- const Response & response_results () const

  return the final solution from selectedIterators (response)

- void initialize_iterator (int job_index)

  initialize the iterator about to be executed within a parallel iterator scheduling function (serve_iterators() or static_-schedule_iterators())

- void pack_parameters_buffer (MPIPackBuffer &send_buffer, int job_index)

  pack a send_buffer for assigning an iterator job to a server

- void unpack_parameters_buffer (MPIUnpackBuffer &recv_buffer)

  unpack a recv_buffer for accepting an iterator job from the scheduler

- void pack_results_buffer (MPIPackBuffer &send_buffer, int job_index)

  pack a send_buffer for returning iterator results from a server

- void unpack_results_buffer (MPIUnpackBuffer &recv_buffer, int job_index)

  unpack a recv_buffer for accepting iterator results from a server

- void update_local_results (int job_index)

  update local PRP results arrays with current iteration results

Private Member Functions

- void run_sequential ()

  run a sequential hybrid

- void run_sequential_adaptive ()

  run a sequential adaptive hybrid

- void partition_sets (size_t num_sets, int job_index, size_t &start_index, size_t &job_size)

  convert num_sets and job_index into a start_index and job_size for extraction from parameterSets

- void extract_parameter_sets (int job_index, VariablesArray &partial_param_sets)

  extract partial_param_sets from parameterSets based on job_index

- void update_local_results (PRPArray &prp_results, int job_id)

  update the partial set of final results from the local iterator execution

- void initialize_iterator (const VariablesArray &param_sets)

  called by unpack_parameters_buffer(MPIUnpackBuffer) and initialize_iterator(int) to update the active Model and Iterator

Private Attributes

- String hybridType

  sequential or sequential_adaptive

- size_t seqCount

  hybrid sequence counter: 0 to numIterators-1

- Real progressMetric

  the amount of progress made in a single iterator++ cycle within a sequential adaptive hybrid

- Real progressThreshold

  when the progress metric falls below this threshold, the sequential adaptive hybrid switches to the next method

- PRP2DArray prpResults

2-D array of results corresponding to numIteratorJobs, one set of results per job (iterators may return multiple final solutions)

- VariablesArray parameterSets

  1-D array of variable starting points for the iterator jobs

43.119.1 Detailed Description

Strategy for sequential hybrid minimization using multiple optimization and nonlinear least squares methods on multiple models of varying fidelity. The sequential hybrid minimization strategy has two approaches: (1) the non-adaptive sequential hybrid runs one method to completion, passes its best results as the starting point for a subsequent method, and continues this succession until all methods have been executed (the stopping rules are controlled internally by each minimizer), and (2) the adaptive sequential hybrid uses adaptive stopping rules for the minimizers that are controlled externally by the strategy. Note that while the strategy is targeted at minimizers, any iterator may be used so long as it defines the notion of a final solution which can be passed as the starting point for subsequent iterators.

43.119.2 Member Function Documentation

43.119.2.1 void pack_parameters_buffer (MPIPackBuffer & *send_buffer*, int *job_index*) [inline, protected, virtual]

pack a send_buffer for assigning an iterator job to a server This virtual function redefinition is executed on the dedicated master processor for self scheduling. It is not used for peer partitions.

Reimplemented from Strategy.

References SequentialHybridStrategy::extract_parameter_sets(), and SequentialHybridStrategy::seqCount.

Referenced by SequentialHybridStrategy::run_sequential().

43.119.2.2 void unpack_parameters_buffer (MPIUnpackBuffer & *recv_buffer*) [inline, protected, virtual]

unpack a recv_buffer for accepting an iterator job from the scheduler This virtual function redefinition is executed on an iterator server for dedicated master self scheduling. It is not used for peer partitions.

Reimplemented from Strategy.

References SequentialHybridStrategy::initialize_iterator(), and SequentialHybridStrategy::seqCount.

43.119.2.3 void pack_results_buffer (MPIPackBuffer & *send_buffer*, int *job_index*) [inline, protected, virtual]

pack a send_buffer for returning iterator results from a server This virtual function redefinition is executed either on an iterator server for dedicated master self scheduling or on peers 2 through n for static scheduling.

Reimplemented from Strategy.

References SequentialHybridStrategy::prpResults.

43.119.2.4 void unpack_results_buffer (MPIUnpackBuffer & *recv_buffer*, int *job_index*) [inline, protected, virtual]

unpack a recv_buffer for accepting iterator results from a server This virtual function redefinition is executed on an strategy master (either the dedicated master processor for self scheduling or peer 1 for static scheduling).

Reimplemented from Strategy.

References SequentialHybridStrategy::prpResults.

43.119.2.5 void run_sequential () [private]

run a sequential hybrid In the sequential nonadaptive case, there is no interference with the iterators. Each runs until its own convergence criteria is satisfied. Status: fully operational.

References Iterator::accepts_multiple_points(), ParallelLibrary::bcast_i(), ParallelLibrary::bcast_si(), Response::function_values(), Strategy::graph2DFlag, Iterator::initialize_graphics(), Model::interface_id(), Response::is_null(), Variables::is_null(), Strategy::iteratorCommRank, Strategy::iteratorCommSize, Strategy::iteratorServerId, HybridStrategy::methodList, Iterator::num_final_solutions(), Strategy::numIteratorJobs, HybridStrategy::numIterators, Strategy::numIteratorServers, SequentialHybridStrategy::pack_parameters_buffer(), Strategy::parallelLib, SequentialHybridStrategy::parameterSets, Strategy::paramsMsgLen, SequentialHybridStrategy::prpResults, ParallelLibrary::recv_si(), Iterator::response_results(), Strategy::resultsMsgLen, Iterator::returns_multiple_points(), Strategy::schedule_iterators(), HybridStrategy::selectedIterators, ParallelLibrary::send_si(), SequentialHybridStrategy::seqCount, MPIPackBuffer::size(), Strategy::stratIterDedMaster, Strategy::stratIterMessagePass, Strategy::tabularDataFile, Strategy::tabularDataFlag, HybridStrategy::userDefinedModels, Iterator::variables_results(), Strategy::worldRank, and Dakota::write_data().

Referenced by SequentialHybridStrategy::run_strategy().

43.119.2.6 void run_sequential_adaptive () [private]

run a sequential adaptive hybrid In the sequential adaptive case, there is interference with the iterators through the use of the ++ overloaded operator. iterator++ runs the iterator for one cycle, after which a progress_metric is computed. This progress metric is used to dictate method switching instead of each iterator's internal convergence criteria. Status: incomplete.

References Strategy::graph2DFlag, HybridStrategy::methodList, HybridStrategy::numIterators, SequentialHybridStrategy::progressMetric, SequentialHybridStrategy::progressThreshold, Strategy::run_iterator(), HybridStrategy::selectedIterators, SequentialHybridStrategy::seqCount, Strategy::tabularDataFile, Strategy::tabularDataFlag, HybridStrategy::userDefinedModels, and Strategy::worldRank.

Referenced by SequentialHybridStrategy::run_strategy().

43.119.2.7 void extract_parameter_sets (int *job_index*, VariablesArray & *partial_param_sets*) [inline, private]

extract partial_param_sets from parameterSets based on job_index This convenience function is executed on an iterator master (static scheduling) or a strategy master (self scheduling) at run initialization time and has access to the full parameterSets array (this is All-Reduced for all peers at the completion of each cycle in run_sequential()).

References SequentialHybridStrategy::parameterSets, and SequentialHybridStrategy::partition_sets().

Referenced by SequentialHybridStrategy::initialize_iterator(), and SequentialHybridStrategy::pack_parameters_-buffer().

The documentation for this class was generated from the following files:

- SequentialHybridStrategy.H
- SequentialHybridStrategy.C

43.120 SerialDirectApplicInterface Class Reference

Sample derived interface class for testing serial simulator plug-ins using assign_rep(). Inheritance diagram for SerialDirectApplicInterface::

Public Member Functions

- SerialDirectApplicInterface (const Dakota::ProblemDescDB &problem_db)

  constructor

- ~SerialDirectApplicInterface ()

  destructor

Protected Member Functions

- int derived_map_ac (const Dakota::String &ac_name)

  execute an analysis code portion of a direct evaluation invocation

43.120.1 Detailed Description

Sample derived interface class for testing serial simulator plug-ins using assign_rep(). The plug-in SerialDirectApplicInterface resides in namespace SIM and uses a copy of rosenbrock() to perform serial parameter to response mappings. It may be activated by specifying the --with-plugin configure option, which activates the DAKOTA_PLUGIN macro in dakota_config.h used by main.C (which activates the plug-in code block within that file) and activates the PLUGIN_S declaration defined in Makefile.include and used in Makefile.source (which add this class to the build). Test input files should then use an analysis_driver of "plugin_rosenbrock".

The documentation for this class was generated from the following files:

- PluginSerialDirectApplicInterface.H
- PluginSerialDirectApplicInterface.C

43.121 SharedVariablesData Class Reference

Container class encapsulating variables data that can be shared among a set of Variables instances.

Public Member Functions

- SharedVariablesData ()

  default constructor

- SharedVariablesData (const ProblemDescDB &problem_db, const std::pair< short, short > &view)

  standard constructor

- SharedVariablesData (const std::pair< short, short > &view, const SizetArray &vars_comps_totals)

  lightweight constructor

- SharedVariablesData (const SharedVariablesData &svd)

  copy constructor

- ~SharedVariablesData ()

  destructor

- SharedVariablesData & operator= (const SharedVariablesData &svd)

  assignment operator

- void size_all_continuous_labels (bool relax)

  size labels for all of the continuous variables, with or without discrete relaxation

- void initialize_all_continuous_types (bool relax)

  initialize types for all of the continuous variables, with or without discrete relaxation

- void initialize_all_continuous_ids (bool relax)

  initialize ids for all of the continuous variables, with or without discrete relaxation

- void size_all_discrete_int_labels ()

  size labels for all of the discrete integer variables

- void initialize_all_discrete_int_types ()

  initialize types for all of the discrete integer variables

- void size_all_discrete_real_labels ()

  size labels for all of the discrete real variables

- void initialize_all_discrete_real_types ()

  initialize types for all of the discrete real variables

- StringMultiArrayView all_continuous_labels (size_t start, size_t num_items) const

  get num_items continuous labels beginning at index start

- void all_continuous_labels (StringMultiArrayConstView cv_labels, size_t start, size_t num_items)

  set num_items continuous labels beginning at index start

- void all_continuous_label (const String &cv_label, size_t index)

  set continuous label at index start

- StringMultiArrayView all_discrete_int_labels (size_t start, size_t num_items) const

  get num_items discrete integer labels beginning at index start

- void all_discrete_int_labels (StringMultiArrayConstView div_labels, size_t start, size_t num_items)

  set num_items discrete integer labels beginning at index start

- void all_discrete_int_label (const String &div_label, size_t index)

  set discrete integer label at index start

- StringMultiArrayView all_discrete_real_labels (size_t start, size_t num_items) const

  get num_items discrete real labels beginning at index start

- void all_discrete_real_labels (StringMultiArrayConstView drv_labels, size_t start, size_t num_items)

  set num_items discrete real labels beginning at index start

- void all_discrete_real_label (const String &drv_label, size_t index)

  set discrete real label at index start

- UShortMultiArrayConstView all_continuous_types (size_t start, size_t num_items) const

  get num_items continuous types beginning at index start

- UShortMultiArrayConstView all_discrete_int_types (size_t start, size_t num_items) const

  get num_items discrete integer types beginning at index start

- UShortMultiArrayConstView all_discrete_real_types (size_t start, size_t num_items) const

  get num_items discrete real types beginning at index start

- SizetMultiArrayConstView all_continuous_ids (size_t start, size_t num_items) const

  get num_items continuous ids beginning at index start

- const SizetArray & merged_discrete_ids () const

  get ids of discrete variables that have been merged into continuous arrays

- const std::pair< short, short > & view () const

  retreive the Variables view

- void inactive_view (short view2)

set the inactive Variables view

- const String & id () const

  return the user-provided or default Variables identifier

- const SizetArray & components_totals () const

  return variable type counts for {continuous,discrete integer,discrete real} {design,aleatory uncertain,epistemic uncertain,state}

- size_t vc_lookup (unsigned short key) const

  retrieve the variables type count within svdRep->variablesComponents corresponding to (a fine-grain variables type) key

Private Attributes

- SharedVariablesDataRep * svdRep

  pointer to the body (handle-body idiom)

43.121.1 Detailed Description

Container class encapsulating variables data that can be shared among a set of Variables instances. An array of Variables objects (e.g., Analyzer::allVariables) contains repeated configuration data (id's, labels, counts). SharedVariablesData employs a handle-body idiom to allow this shared data to be managed in a single object with many references to it, one per Variables object in the array. This allows scaling to larger sample sets.

The documentation for this class was generated from the following file:

- SharedVariablesData.H

43.122 SharedVariablesDataRep Class Reference

The representation of a SharedVariablesData instance. This representation, or body, may be shared by multiple SharedVariablesData handle instances.

Private Member Functions

- SharedVariablesDataRep (const ProblemDescDB &problem_db, const std::pair< short, short > &view)
  standard constructor
- SharedVariablesDataRep (const std::pair< short, short > &view, const SizetArray &vars_comps_totals)
  lightweight constructor
- ~SharedVariablesDataRep ()
  destructor
- void size_all_continuous_labels (bool relax)
  size allContinuousLabels, with or without discrete relaxation
- void initialize_all_continuous_types (bool relax)
  initialize allContinuousTypes, with or without discrete relaxation
- void initialize_all_continuous_ids (bool relax)
  initialize allContinuousIds, with or without discrete relaxation
- void size_all_discrete_int_labels ()
  size allDiscreteIntLabels
- void initialize_all_discrete_int_types ()
  initialize allDiscreteIntTypes
- void size_all_discrete_real_labels ()
  size allDiscreteRealLabels
- void initialize_all_discrete_real_types ()
  initialize allDiscreteRealTypes
- size_t vc_lookup (unsigned short key) const
  retrieve the count within variablesComponents corresponding to key

Private Attributes

- String variablesId
  variables identifier string from the input file

- std::pair< short, short > variablesView

  the variables view pair containing active (first) and inactive (second) view enumerations

- std::map< unsigned short, size_t > variablesComponents

  map linking variable types to counts

- SizetArray variablesCompsTotals

  totals for variable type counts for {continuous,discrete integer,discrete real} {design,aleatory uncertain,epistemic uncertain,state}

- StringMultiArray allContinuousLabels

  array of variable labels for all of the continuous variables

- StringMultiArray allDiscreteIntLabels

  array of variable labels for all of the discrete integer variables

- StringMultiArray allDiscreteRealLabels

  array of variable labels for all of the discrete real variables

- UShortMultiArray allContinuousTypes

  array of variable types for all of the continuous variables

- UShortMultiArray allDiscreteIntTypes

  array of variable types for all of the discrete integer variables

- UShortMultiArray allDiscreteRealTypes

  array of variable types for all of the discrete real variables

- SizetMultiArray allContinuousIds

  array of 1-based position identifiers for the all continuous variables array

- SizetArray mergedDiscreteIds

  array of discrete variable identifiers for which the discrete requirement is relaxed by merging them into a continuous array

- int referenceCount

  number of handle objects sharing svdRep

Friends

- class **SharedVariablesData**

43.122.1 Detailed Description

The representation of a SharedVariablesData instance. This representation, or body, may be shared by multiple SharedVariablesData handle instances. The SharedVariablesData/SharedVariablesDataRep pairs utilize a handle-body idiom (Coplien, Advanced C++).

43.122.2 Constructor & Destructor Documentation

43.122.2.1 SharedVariablesDataRep (const ProblemDescDB & *problem_db*, const std::pair< short, short > & *view*) [private]

standard constructor This constructor is the one which must build the base class data for all derived classes. get_-variables() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_variables() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~Variables).

References SharedVariablesDataRep::allContinuousLabels, SharedVariablesDataRep::allDiscreteIntLabels, SharedVariablesDataRep::allDiscreteRealLabels, Dakota::copy_data_partial(), ProblemDescDB::get_-dsa(), ProblemDescDB::get_sizet(), SharedVariablesDataRep::initialize_all_continuous_ids(), SharedVariablesDataRep::initialize_all_continuous_types(), SharedVariablesDataRep::initialize_all_-discrete_int_types(), SharedVariablesDataRep::initialize_all_discrete_real_types(), SharedVariables-DataRep::variablesComponents, SharedVariablesDataRep::variablesCompsTotals, and SharedVariables-DataRep::variablesView.

43.122.3 Member Data Documentation

43.122.3.1 SizetMultiArray allContinuousIds [private]

array of 1-based position identifiers for the all continuous variables array These identifiers define positions of the all continuous variables array within the total variable sequence.

Referenced by SharedVariablesDataRep::initialize_all_continuous_ids().

The documentation for this class was generated from the following files:

- SharedVariablesData.H
- SharedVariablesData.C

43.123 SingleMethodStrategy Class Reference

Simple fall-through strategy for running a single iterator on a single model. Inheritance diagram for SingleMethodStrategy::

Public Member Functions

- SingleMethodStrategy (ProblemDescDB &problem_db)

  constructor

- ~SingleMethodStrategy ()

  destructor

- void run_strategy ()

  Perform the strategy by executing selectedIterator on userDefinedModel.

- const Variables & variables_results () const

  return the final solution from selectedIterator (variables)

- const Response & response_results () const

  return the final solution from selectedIterator (response)

Private Attributes

- Model userDefinedModel

  the model to be iterated

- Iterator selectedIterator

  the iterator

43.123.1 Detailed Description

Simple fall-through strategy for running a single iterator on a single model. This strategy executes a single iterator on a single model. Since it does not provide coordination for multiple iterators and models, it can considered to be a "fall-through" strategy in that it allows control to fall through immediately to the iterator.

The documentation for this class was generated from the following files:

- SingleMethodStrategy.H
- SingleMethodStrategy.C

43.124 SingleModel Class Reference

Derived model class which utilizes a single interface to map variables into responses. Inheritance diagram for SingleModel::

Public Member Functions

- SingleModel (ProblemDescDB &problem_db)

  constructor

- ~SingleModel ()

  destructor

Protected Member Functions

- Interface & interface ()

  return userDefinedInterface

- void derived_compute_response (const ActiveSet &set)

  portion of compute_response() specific to SingleModel (invokes a synchronous map() on userDefinedInterface)

- void derived_asynch_compute_response (const ActiveSet &set)

  portion of asynch_compute_response() specific to SingleModel (invokes an asynchronous map() on userDefinedInterface)

- const IntResponseMap & derived_synchronize ()

  portion of synchronize() specific to SingleModel (invokes synch() on userDefinedInterface)

- const IntResponseMap & derived_synchronize_nowait ()

  portion of synchronize_nowait() specific to SingleModel (invokes synch_nowait() on userDefinedInterface)

- void component_parallel_mode (short mode)

  SingleModel only supports parallelism in userDefinedInterface, so this virtual function redefinition is simply a sanity check.

- String local_eval_synchronization ()

  return userDefinedInterface synchronization setting

- int local_eval_concurrency ()

  return userDefinedInterface asynchronous evaluation concurrency

- bool derived_master_overload () const

  flag which prevents overloading the master with a multiprocessor evaluation (request forwarded to userDefinedInterface)

- void derived_init_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set up SingleModel for parallel operations (request forwarded to userDefinedInterface)

- void derived_init_serial ()

  set up SingleModel for serial operations (request forwarded to userDefinedInterface).

- void derived_set_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  set active parallel configuration for the SingleModel (request forwarded to userDefinedInterface)

- void derived_free_communicators (const int &max_iterator_concurrency, bool recurse_flag=true)

  deallocate communicator partitions for the SingleModel (request forwarded to userDefinedInterface)

- void serve ()

  Service userDefinedInterface job requests received from the master. Completes when a termination message is received from stop_servers().

- void stop_servers ()

  executed by the master to terminate userDefinedInterface server operations when SingleModel iteration is complete.

- const String & interface_id () const

  return the userDefinedInterface identifier

- int evaluation_id () const

  return the current evaluation id for the SingleModel (request forwarded to userDefinedInterface)

- bool evaluation_cache () const

  return flag indicated usage of an evaluation cache by the SingleModel (request forwarded to userDefinedInterface)

- void set_evaluation_reference ()

  set the evaluation counter reference points for the SingleModel (request forwarded to userDefinedInterface)

- void fine_grained_evaluation_counters ()

  request fine-grained evaluation reporting within the userDefinedInterface

- void print_evaluation_summary (std::ostream &s, bool minimal_header=false, bool relative_count=true) const

  print the evaluation summary for the SingleModel (request forwarded to userDefinedInterface)

Private Attributes

- Interface userDefinedInterface

  the interface used for mapping variables to responses

43.124.1 Detailed Description

Derived model class which utilizes a single interface to map variables into responses. The SingleModel class is the simplest of the derived model classes. It provides the capabilities of the original Model class, prior to the development of surrogate and nested model extensions. The derived response computation and synchronization functions utilize a single interface to perform the function evaluations.

The documentation for this class was generated from the following files:

- SingleModel.H
- SingleModel.C

43.125 SNLLBase Class Reference

Base class for OPT++ optimization and least squares methods. Inheritance diagram for SNLLBase::

Public Member Functions

- SNLLBase ()

  default constructor

- SNLLBase (Model &model)

  standard constructor

- ~SNLLBase ()

  destructor

Protected Member Functions

- void copy_con_vals_dak_to_optpp (const RealVector &local_fn_vals, RealVector &g, const size_t &offset)

  convenience function for copying local_fn_vals to g; used by constraint evaluator functions

- void copy_con_vals_optpp_to_dak (const RealVector &g, RealVector &local_fn_vals, const size_t &offset)

  convenience function for copying g to local_fn_vals; used in final solution logging

- void copy_con_grad (const RealMatrix &local_fn_grads, RealMatrix &grad_g, const size_t &offset)

  convenience function for copying local_fn_grads to grad_g; used by constraint evaluator functions

- void copy_con_hess (const RealSymMatrixArray &local_fn_hessians, OPTPP::OptppArray< RealSymMatrix > &hess_g, const size_t &offset)

  convenience function for copying local_fn_hessians to hess_g; used by constraint evaluator functions

- void snll_pre_instantiate (const String &merit_fn, bool bound_constr_flag, const int &num_constr)

  convenience function for setting OPT++ options prior to the method instantiation

- void snll_post_instantiate (const int &num_cv, bool vendor_num_grad_flag, const String &finite_diff_type, const Real &fdss, const int &max_iter, const int &max_fn_evals, const Real &conv_tol, const

Real &grad_tol, const Real &max_step, bool bound_constr_flag, const int &num_constr, short output_-lev, OPTPP::OptimizeClass *the_optimizer, OPTPP::NLP0 *nlf_objective, OPTPP::FDNLF1 *fd_nlf1, OPTPP::FDNLF1 *fd_nlf1_con)

convenience function for setting OPT++ options after the method instantiation

- void snll_initialize_run (OPTPP::NLP0 *nlf_objective, OPTPP::NLP *nlp_constraint, const RealVector &init_pt, bool bound_constr_flag, const RealVector &lower_bnds, const RealVector &upper_bnds, const RealMatrix &lin_ineq_coeffs, const RealVector &lin_ineq_l_bnds, const RealVector &lin_ineq_u_bnds, const RealMatrix &lin_eq_coeffs, const RealVector &lin_eq_targets, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_targets)

  convenience function for OPT++ configuration prior to the method invocation

- void snll_post_run (OPTPP::NLP0 *nlf_objective)

  convenience function for setting OPT++ options after the method instantiations

Static Protected Member Functions

- static void init_fn (int n, RealVector &x)

  An initialization mechanism provided by OPT++ (not currently used).

Protected Attributes

- String searchMethod

  value_based_line_search, gradient_based_line_search, trust_region, or tr_pds

- OPTPP::SearchStrategy searchStrat

  enum: LineSearch, TrustRegion, or TrustPDS

- OPTPP::MeritFcn meritFn

  enum: NormFmu, ArgaezTapia, or VanShanno

- bool constantASVFlag

  flags a user selection of active_set_vector == constant. By mapping this into mode override, reliance on duplicate detection can be avoided.

Static Protected Attributes

- static Minimizer * optLSqInstance

  pointer to the active base class object instance used within the static evaluator functions in order to avoid the need for static data

- static bool modeOverrideFlag

flags OPT++ mode override (for combining value, gradient, and Hessian requests)

- static EvalType lastFnEvalLocn

  an enum used to track whether an nlf evaluator or a constraint evaluator was the last location of a function evaluation

- static int lastEvalMode

  copy of mode from constraint evaluators

- static RealVector lastEvalVars

  copy of variables from constraint evaluators

43.125.1 Detailed Description

Base class for OPT++ optimization and least squares methods. The SNLLBase class provides a common base class for SNLLOptimizer and SNLLLeastSq, both of which are wrappers for OPT++, a C++ optimization library from the Computational Sciences and Mathematics Research (CSMR) department at Sandia's Livermore CA site.

The documentation for this class was generated from the following files:

- SNLLBase.H
- SNLLBase.C

43.126 SNLLLeastSq Class Reference

Wrapper class for the OPT++ optimization library. Inheritance diagram for SNLLLeastSq::

Public Member Functions

- SNLLLeastSq (Model &model)
  standard constructor
- SNLLLeastSq (const String &method_name, Model &model)
  alternate constructor for instantiations without ProblemDescDB support
- ~SNLLLeastSq ()
  destructor
- void minimize_residuals ()
  Performs the iterations to determine the least squares solution.

Protected Member Functions

- void initialize_run ()
  invokes LeastSq::initialize_run(), SNLLBase::snll_initialize_run(), and performs other set-up
- void post_run (std::ostream &s)
  invokes snll_post_run and re-implements post_run (does not call parent) and performs other solution processing
- void finalize_run ()
  restores instances

Static Private Member Functions

- static void nlf2_evaluator_gn (int mode, int n, const RealVector &x, double &f, RealVector &grad_f, RealSymMatrix &hess_f, int &result_mode)

objective function evaluator function which obtains values and gradients for least square terms and computes objective function value, gradient, and Hessian using the Gauss-Newton approximation.

- static void constraint1_evaluator_gn (int mode, int n, const RealVector &x, RealVector &g, RealMatrix &grad_g, int &result_mode)

  constraint evaluator function which provides constraint values and gradients to OPT++ Gauss-Newton methods.

- static void constraint2_evaluator_gn (int mode, int n, const RealVector &x, RealVector &g, RealMatrix &grad_g, OPTPP::OptppArray< RealSymMatrix > &hess_g, int &result_mode)

  constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ Gauss-Newton methods.

Private Attributes

- SNLLLeastSq * prevSnllLSqInstance

  pointer to the previously active object instance used for restoration in the case of iterator/model recursion

- OPTPP::NLP0 * nlfObjective

  objective NLF base class pointer

- OPTPP::NLP0 * nlfConstraint

  constraint NLF base class pointer

- OPTPP::NLP * nlpConstraint

  constraint NLP pointer

- OPTPP::NLF2 * nlf2

  pointer to objective NLF for full Newton optimizers

- OPTPP::NLF2 * nlf2Con

  pointer to constraint NLF for full Newton optimizers

- OPTPP::NLF1 * nlf1Con

  pointer to constraint NLF for Quasi Newton optimizers

- OPTPP::OptimizeClass * theOptimizer

  optimizer base class pointer

- OPTPP::OptNewton * optnewton

  Newton optimizer pointer.

- OPTPP::OptBCNewton * optbcnewton

  Bound constrained Newton optimizer ptr.

- OPTPP::OptDHNIPS * optdhnips

  Disaggregated Hessian NIPS optimizer ptr.

Static Private Attributes

- static SNLLLeastSq * snllLSqInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.126.1 Detailed Description

Wrapper class for the OPT++ optimization library. The SNLLLeastSq class provides a wrapper for OPT++, a C++ optimization library of nonlinear programming and pattern search techniques from the Computational Sciences and Mathematics Research (CSMR) department at Sandia's Livermore CA site. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function, a static member, or accessed by static pointer.

The user input mappings are as follows: `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, `max_step`, `gradient_tolerance`, `search_method`, and `search_scheme_size` are set using OPT++'s setMaxIter(), setMaxFeval(), setFcnTol(), setMaxStep(), setGradTol(), setSearchStrategy(), and setSSS() member functions, respectively; `output` verbosity is used to toggle OPT++'s debug mode using the setDebug() member function. Internal to OPT++, there are 3 search strategies, while the DAKOTA `search_method` specification supports 4 (`value_based_line_search`, `gradient_based_line_search`, `trust_region`, or `tr_pds`). The difference stems from the "is_expensive" flag in OPT++. If the search strategy is LineSearch and "is_expensive" is turned on, then the `value_based_line_search` is used. Otherwise (the "is_expensive" default is off), the algorithm will use the `gradient_based_line_search`. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/packages/OPTPP directory for information on OPT++ class member functions.

43.126.2 Member Function Documentation

43.126.2.1 void post_run (std::ostream & *s*) `[protected, virtual]`

invokes snll_post_run and re-implements post_run (does not call parent) and performs other solution processing SNLLLeastSq requires fn DB lookup, so overrides LeastSq::post_run and directly invokes Iterator::post_run when complete.

Reimplemented from LeastSq.

References Iterator::activeSet, Iterator::bestResponseArray, Iterator::bestVariablesArray, SNLLBase::copy_con_vals_optpp_to_dak(), Dakota::copy_data_partial(), Minimizer::cvScaleMultipliers, Minimizer::cvScaleOffsets, Minimizer::cvScaleTypes, Dakota::data_pairs, Response::function_values(), LeastSq::get_confidence_intervals(), Model::interface_id(), Iterator::iteratedModel, Dakota::lookup_by_val(), Minimizer::modify_s2n(), SNLLLeastSq::nlfObjective, Iterator::numFunctions, LeastSq::numLeastSqTerms, Minimizer::numNonlinearConstraints, ActiveSet::request_values(), ActiveSet::request_vector(), Minimizer::responseScaleMultipliers, Minimizer::responseScaleOffsets, Minimizer::responseScaleTypes, Minimizer::secondaryRespScaleFlag, SNLLBase::snll_post_run(), SNLLLeastSq::theOptimizer, and Minimizer::varsScaleFlag.

43.126.2.2 void nlf2_evaluator_gn (int *mode*, int *n*, const RealVector & *x*, double & *f*, RealVector & *grad_f*, RealSymMatrix & *hess_f*, int & *result_mode*) [static, private]

objective function evaluator function which obtains values and gradients for least square terms and computes objective function value, gradient, and Hessian using the Gauss-Newton approximation. This nlf2 evaluator function is used for the Gauss-Newton method in order to exploit the special structure of the nonlinear least squares problem. Here, fx = sum (T_i - Tbar_i)^2 and Response is made up of residual functions and their gradients along with any nonlinear constraints. The objective function and its gradient vector and Hessian matrix are computed directly from the residual functions and their derivatives (which are returned from the Response object).

References Dakota::abort_handler(), Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), Model::current_response(), Response::function_gradients(), Response::function_values(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Iterator::numFunctions, LeastSq::numLeastSqTerms, Minimizer::numNonlinearConstraints, Iterator::outputLevel, ActiveSet::request_vector(), SNLLLeastSq::snlllLSqInstance, Dakota::write_data(), and Dakota::write_precision.

Referenced by SNLLLeastSq::SNLLLeastSq().

43.126.2.3 void constraint1_evaluator_gn (int *mode*, int *n*, const RealVector & *x*, RealVector & *g*, RealMatrix & *grad_g*, int & *result_mode*) [static, private]

constraint evaluator function which provides constraint values and gradients to OPT++ Gauss-Newton methods. While it does not employ the Gauss-Newton approximation, it is distinct from constraint1_evaluator() due to its need to anticipate the required modes for the least squares terms. This constraint evaluator function is used with diaggregated Hessian NIPS and is currently active.

References Dakota::abort_handler(), Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), SNLLBase::copy_con_grad(), SNLLBase::copy_con_vals_dak_to_optpp(), Model::current_response(), Response::function_gradients(), Response::function_values(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Iterator::numFunctions, LeastSq::numLeastSqTerms, Iterator::outputLevel, ActiveSet::request_vector(), and SNLLLeastSq::snlllLSqInstance.

Referenced by SNLLLeastSq::SNLLLeastSq().

43.126.2.4 void constraint2_evaluator_gn (int *mode*, int *n*, const RealVector & *x*, RealVector & *g*, RealMatrix & *grad_g*, OPTPP::OptppArray< RealSymMatrix > & *hess_g*, int & *result_mode*) [static, private]

constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ Gauss-Newton methods. While it does not employ the Gauss-Newton approximation, it is distinct from constraint2_evaluator() due to its need to anticipate the required modes for the least squares terms. This constraint evaluator function is used with full Newton NIPS and is currently inactive.

References Dakota::abort_handler(), Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), SNLLBase::copy_con_grad(), SNLLBase::copy_con_hess(), SNLLBase::copy_con_vals_dak_to_optpp(), Model::current_response(), Response::function_gradients(), Response::function_hessians(), Response::function_values(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, SNLLBase::modeOverrideFlag, Iterator::numFunctions,

LeastSq::numLeastSqTerms, Iterator::outputLevel, ActiveSet::request_vector(), and SNLL-LeastSq::snllLSqInstance.

The documentation for this class was generated from the following files:

- SNLLLeastSq.H
- SNLLLeastSq.C

43.127 SNLLOptimizer Class Reference

Wrapper class for the OPT++ optimization library. Inheritance diagram for SNLLOptimizer::

Public Member Functions

- SNLLOptimizer (Model &model)

  standard constructor

- SNLLOptimizer (const String &method_name, Model &model)

  alternate constructor for instantiations "on the fly"

- SNLLOptimizer (const RealVector &initial_pt, const RealVector &var_l_bnds, const RealVector &var_u_bnds, const RealMatrix &lin_ineq_coeffs, const RealVector &lin_ineq_l_bnds, const RealVector &lin_ineq_u_bnds, const RealMatrix &lin_eq_coeffs, const RealVector &lin_eq_tgts, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_tgts, void(∗user_obj_eval)(int mode, int n, const RealVector &x, double &f, RealVector &grad_f, int &result_mode), void(∗user_con_eval)(int mode, int n, const RealVector &x, RealVector &g, RealMatrix &grad_g, int &result_mode))

  alternate constructor for instantiations "on the fly"

- ∼SNLLOptimizer ()

  destructor

- void find_optimum ()

  Performs the iterations to determine the optimal solution.

Protected Member Functions

- void initialize_run ()

  invokes Optimizer::initialize_run(), SNLLBase::snll_initialize_run(), and performs other set-up

- void post_run (std::ostream &s)

  performs data recovery and calls Optimizer::post_run()

- void finalize_run ()

  performs cleanup, restores instances and calls parent finalize

Static Private Member Functions

- static void nlf0_evaluator (int n, const RealVector &x, double &f, int &result_mode)

  objective function evaluator function for OPT++ methods which require only function values.

- static void nlf1_evaluator (int mode, int n, const RealVector &x, double &f, RealVector &grad_f, int &result_mode)

  objective function evaluator function which provides function values and gradients to OPT++ methods.

- static void nlf2_evaluator (int mode, int n, const RealVector &x, double &f, RealVector &grad_f, RealSymMatrix &hess_f, int &result_mode)

  objective function evaluator function which provides function values, gradients, and Hessians to OPT++ methods.

- static void constraint0_evaluator (int n, const RealVector &x, RealVector &g, int &result_mode)

  constraint evaluator function for OPT++ methods which require only constraint values.

- static void constraint1_evaluator (int mode, int n, const RealVector &x, RealVector &g, RealMatrix &grad_g, int &result_mode)

  constraint evaluator function which provides constraint values and gradients to OPT++ methods.

- static void constraint2_evaluator (int mode, int n, const RealVector &x, RealVector &g, RealMatrix &grad_g, OPTPP::OptppArray< RealSymMatrix > &hess_g, int &result_mode)

  constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ methods.

Private Attributes

- SNLLOptimizer * prevSnllOptInstance

  pointer to the previously active object instance used for restoration in the case of iterator/model recursion

- OPTPP::NLP0 * nlfObjective

  objective NLF base class pointer

- OPTPP::NLP0 * nlfConstraint

  constraint NLF base class pointer

- OPTPP::NLP * nlpConstraint

  constraint NLP pointer

- OPTPP::NLF0 * nlf0

pointer to objective NLF for nongradient optimizers

- OPTPP::NLF1 * nlf1

  pointer to objective NLF for (analytic) gradient-based optimizers

- OPTPP::NLF1 * nlf1Con

  pointer to constraint NLF for (analytic) gradient-based optimizers

- OPTPP::FDNLF1 * fdnlf1

  pointer to objective NLF for (finite diff) gradient-based optimizers

- OPTPP::FDNLF1 * fdnlf1Con

  pointer to constraint NLF for (finite diff) gradient-based optimizers

- OPTPP::NLF2 * nlf2

  pointer to objective NLF for full Newton optimizers

- OPTPP::NLF2 * nlf2Con

  pointer to constraint NLF for full Newton optimizers

- OPTPP::OptimizeClass * theOptimizer

  optimizer base class pointer

- OPTPP::OptPDS * optpds

  PDS optimizer pointer.

- OPTPP::OptCG * optcg

  CG optimizer pointer.

- OPTPP::OptLBFGS * optlbfgs

  L-BFGS optimizer pointer.

- OPTPP::OptNewton * optnewton

  Newton optimizer pointer.

- OPTPP::OptQNewton * optqnewton

  Quasi-Newton optimizer pointer.

- OPTPP::OptFDNewton * optfdnewton

  Finite Difference Newton opt pointer.

- OPTPP::OptBCNewton * optbcnewton

  Bound constrained Newton opt pointer.

- OPTPP::OptBCQNewton * optbcqnewton

  Bnd constrained Quasi-Newton opt ptr.

- OPTPP::OptBCFDNewton * optbcfdnewton

  Bnd constrained FD-Newton opt ptr.

- OPTPP::OptNIPS * optnips

  NIPS optimizer pointer.

- OPTPP::OptQNIPS * optqnips

  Quasi-Newton NIPS optimizer pointer.

- OPTPP::OptFDNIPS * optfdnips

  Finite Difference NIPS opt pointer.

- String setUpType

  flag for iteration mode: "model" (normal usage) or "user_functions" (user-supplied functions mode for "on the fly" instantiations). NonDReliability currently uses the user_functions mode.

- RealVector initialPoint

  holds initial point passed in for "user_functions" mode.

- RealVector lowerBounds

  holds variable lower bounds passed in for "user_functions" mode.

- RealVector upperBounds

  holds variable upper bounds passed in for "user_functions" mode.

Static Private Attributes

- static SNLLOptimizer * snllOptInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

43.127.1 Detailed Description

Wrapper class for the OPT++ optimization library. The SNLLOptimizer class provides a wrapper for OPT++, a C++ optimization library of nonlinear programming and pattern search techniques from the Computational Sciences and Mathematics Research (CSMR) department at Sandia's Livermore CA site. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function, a static member, or accessed by static pointer.

The user input mappings are as follows: `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, `max_step`, `gradient_tolerance`, `search_method`, and `search_scheme_size` are set using OPT++'s setMaxIter(), setMaxFeval(), setFcnTol(), setMaxStep(), setGradTol(), setSearchStrategy(), and setSSS() member functions, respectively; `output` verbosity is used to toggle OPT++'s

debug mode using the setDebug() member function. Internal to OPT++, there are 3 search strategies, while the DAKOTA `search_method` specification supports 4 (`value_based_line_search`, `gradient_based_line_search`, `trust_region`, or `tr_pds`). The difference stems from the "is_expensive" flag in OPT++. If the search strategy is LineSearch and "is_expensive" is turned on, then the `value_based_line_search` is used. Otherwise (the "is_expensive" default is off), the algorithm will use the `gradient_based_line_search`. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/packages/OPTPP directory for information on OPT++ class member functions.

43.127.2 Constructor & Destructor Documentation

43.127.2.1 SNLLOptimizer (Model & *model*)

standard constructor This constructor is used for normal instantiations using data from the ProblemDescDB.

References Dakota::abort_handler(), Minimizer::boundConstraintFlag, SNLLOptimizer::constraint0_evaluator(), SNLLOptimizer::constraint1_evaluator(), SNLLOptimizer::constraint2_evaluator(), Iterator::convergenceTol, Iterator::fdGradStepSize, SNLLOptimizer::fdnlf1, SNLLOptimizer::fdnlf1Con, ProblemDescDB::get_int(), ProblemDescDB::get_real(), ProblemDescDB::get_string(), Model::init_communicators(), SNLLBase::init_fn(), Iterator::intervalType, Iterator::iteratedModel, Dakota::LARGE_SCALE, Optimizer::localObjectiveRecast, Iterator::maxConcurrency, Iterator::maxFunctionEvals, Iterator::maxIterations, SNLLBase::meritFn, Iterator::methodName, SNLLOptimizer::nlf0, SNLLOptimizer::nlf0_evaluator(), SNLLOptimizer::nlf1, SNLLOptimizer::nlf1_evaluator(), SNLLOptimizer::nlf1Con, SNLLOptimizer::nlf2, SNLLOptimizer::nlf2_evaluator(), SNLLOptimizer::nlf2Con, SNLLOptimizer::nlfConstraint, SNLLOptimizer::nlfObjective, SNLLOptimizer::nlpConstraint, Minimizer::numConstraints, Iterator::numContinuousVars, Minimizer::numNonlinearConstraints, SNLLOptimizer::optbcfdnewton, SNLLOptimizer::optbcnewton, SNLLOptimizer::optbcqnewton, SNLLOptimizer::optcg, SNLLOptimizer::optfdnewton, SNLLOptimizer::optfdnips, SNLLOptimizer::optlbfgs, SNLLOptimizer::optnewton, SNLLOptimizer::optnips, SNLLOptimizer::optpds, SNLLOptimizer::optqnewton, SNLLOptimizer::optqnips, Iterator::outputLevel, Iterator::probDescDB, Minimizer::scaleFlag, SNLLBase::searchStrat, SNLLBase::snll_post_instantiate(), SNLLBase::snll_pre_instantiate(), SNLLOptimizer::theOptimizer, and Minimizer::vendorNumericalGradFlag.

43.127.2.2 SNLLOptimizer (const String & *method_name*, Model & *model*)

alternate constructor for instantiations "on the fly" This is an alternate constructor for instantiations on the fly using a Model but no ProblemDescDB.

References Minimizer::boundConstraintFlag, SNLLOptimizer::constraint1_evaluator(), Iterator::convergenceTol, Iterator::fdGradStepSize, SNLLBase::init_fn(), Iterator::intervalType, Dakota::LARGE_SCALE, Iterator::maxFunctionEvals, Iterator::maxIterations, SNLLBase::meritFn, Iterator::methodName, SNLLOptimizer::nlf1, SNLLOptimizer::nlf1_evaluator(), SNLLOptimizer::nlf1Con, SNLLOptimizer::nlfConstraint, SNLLOptimizer::nlfObjective, SNLLOptimizer::nlpConstraint, Minimizer::numConstraints, Iterator::numContinuousVars, Minimizer::numNonlinearConstraints, SNLLOptimizer::optbcqnewton, SNLLOptimizer::optlbfgs, SNLLOptimizer::optqnewton, SNLLOptimizer::optqnips, Iterator::outputLevel, SNLLBase::searchStrat, SNLLBase::snll_post_instantiate(), SNLLBase::snll_pre_instantiate(), SNLLOptimizer::theOptimizer, and Minimizer::vendorNumericalGradFlag.

43.127.2.3 SNLLOptimizer (const RealVector & *initial_pt*, const RealVector & *var_l_bnds*, const RealVector & *var_u_bnds*, const RealMatrix & *lin_ineq_coeffs*, const RealVector & *lin_ineq_l_bnds*, const RealVector & *lin_ineq_u_bnds*, const RealMatrix & *lin_eq_coeffs*, const RealVector & *lin_eq_tgts*, const RealVector & *nln_ineq_l_bnds*, const RealVector & *nln_ineq_u_bnds*, const RealVector & *nln_eq_tgts*, void(∗)(int mode, int n, const RealVector &x, double &f, RealVector &grad_f, int &result_mode) *user_obj_eval*, void(∗)(int mode, int n, const RealVector &x, RealVector &g, RealMatrix &grad_g, int &result_mode) *user_con_eval*)

alternate constructor for instantiations "on the fly" This is an alternate constructor for performing an optimization using the passed in objective function and constraint function pointers.

References Minimizer::bigRealBoundSize, Minimizer::boundConstraintFlag, SNLLBase::init_fn(), SNLLOptimizer::initialPoint, Dakota::LARGE_SCALE, SNLLOptimizer::lowerBounds, SNLLBase::meritFn, SNLLOptimizer::nlf1, SNLLOptimizer::nlf1Con, SNLLOptimizer::nlfConstraint, SNLLOptimizer::nlfObjective, SNLLOptimizer::nlpConstraint, Minimizer::numConstraints, Iterator::numContinuousVars, Minimizer::numNonlinearConstraints, SNLLOptimizer::optbcqnewton, SNLLOptimizer::optlbfgs, SNLLOptimizer::optqnewton, SNLLOptimizer::optqnips, Iterator::outputLevel, SNLLBase::searchStrat, SNLLBase::snll_initialize_run(), SNLLBase::snll_post_instantiate(), SNLLBase::snll_pre_instantiate(), SNLLOptimizer::theOptimizer, and SNLLOptimizer::upperBounds.

43.127.3 Member Function Documentation

43.127.3.1 void nlf0_evaluator (int *n*, const RealVector & *x*, double & *f*, int & *result_mode*) `[static, private]`

objective function evaluator function for OPT++ methods which require only function values. For use when DAKOTA computes f and gradients are not directly available. This is used by nongradient-based optimizers such as PDS and by gradient-based optimizers in vendor numerical gradient mode (opt++'s internal finite difference routine is used).

References Model::compute_response(), Model::continuous_variables(), Model::current_response(), Response::function_value(), Iterator::iteratedModel, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Minimizer::numNonlinearConstraints, Iterator::outputLevel, and SNLLOptimizer::snllOptInstance.

Referenced by SNLLOptimizer::SNLLOptimizer().

43.127.3.2 void nlf1_evaluator (int *mode*, int *n*, const RealVector & *x*, double & *f*, RealVector & *grad_f*, int & *result_mode*) `[static, private]`

objective function evaluator function which provides function values and gradients to OPT++ methods. For use when DAKOTA computes f and df/dX (regardless of gradientType). Vendor numerical gradient case is handled by nlf0_evaluator.

References Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), Model::current_response(), Response::function_gradient_copy(), Response::function_value(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Minimizer::numNonlinearConstraints, Iterator::outputLevel, ActiveSet::request_values(), and SNLLOptimizer::snllOptInstance.

Referenced by SNLLOptimizer::SNLLOptimizer().

43.127.3.3 void nlf2_evaluator (int *mode*, int *n*, const RealVector & *x*, double & *f*, RealVector & *grad_f*, RealSymMatrix & *hess_f*, int & *result_mode*) `[static, private]`

objective function evaluator function which provides function values, gradients, and Hessians to OPT++ methods. For use when DAKOTA receives f, df/dX, & d^2f/dx^2 from the ApplicationInterface (analytic only). Finite differencing does not make sense for a full Newton approach, since lack of analytic gradients & Hessian should dictate the use of quasi-newton or fd-newton. Thus, there is no fdnlf2_evaluator for use with full Newton approaches, since it is preferable to use quasi-newton or fd-newton with nlf1. Gauss-Newton does not fit this model; it uses nlf2_evaluator_gn instead of nlf2_evaluator.

References Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), Model::current_response(), Response::function_gradient_copy(), Response::function_hessian(), Response::function_value(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Minimizer::numNonlinearConstraints, Iterator::outputLevel, ActiveSet::request_values(), and SNLLOptimizer::snllOptInstance.

Referenced by SNLLOptimizer::SNLLOptimizer().

43.127.3.4 void constraint0_evaluator (int *n*, const RealVector & *x*, RealVector & *g*, int & *result_mode*) `[static, private]`

constraint evaluator function for OPT++ methods which require only constraint values. For use when DAKOTA computes g and gradients are not directly available. This is used by nongradient-based optimizers and by gradient-based optimizers in vendor numerical gradient mode (opt++'s internal finite difference routine is used).

References Model::compute_response(), Model::continuous_variables(), SNLLBase::copy_con_vals_dak_to_optpp(), Model::current_response(), Response::function_values(), Iterator::iteratedModel, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Optimizer::numObjectiveFns, Iterator::outputLevel, and SNLLOptimizer::snllOptInstance.

Referenced by SNLLOptimizer::SNLLOptimizer().

43.127.3.5 void constraint1_evaluator (int *mode*, int *n*, const RealVector & *x*, RealVector & *g*, RealMatrix & *grad_g*, int & *result_mode*) `[static, private]`

constraint evaluator function which provides constraint values and gradients to OPT++ methods. For use when DAKOTA computes g and dg/dX (regardless of gradientType). Vendor numerical gradient case is handled by constraint0_evaluator.

References Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), SNLLBase::copy_con_grad(), SNLLBase::copy_con_vals_dak_to_optpp(), Model::current_response(), Response::function_gradients(), Response::function_values(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Optimizer::numObjectiveFns, Iterator::outputLevel, ActiveSet::request_values(), and SNLLOptimizer::snllOptInstance.

Referenced by SNLLOptimizer::SNLLOptimizer().

43.127.3.6 void constraint2_evaluator (int *mode*, int *n*, const RealVector & *x*, RealVector & *g*, RealMatrix & *grad_g*, OPTPP::OptppArray< RealSymMatrix > & *hess_g*, int & *result_mode*) `[static, private]`

constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ methods. For use when DAKOTA computes g, dg/dX, & d^2g/dx^2 (analytic only).

References Iterator::activeSet, Model::compute_response(), Model::continuous_variables(), SNLLBase::copy_con_grad(), SNLLBase::copy_con_hess(), SNLLBase::copy_con_vals_dak_to_optpp(), Model::current_response(), Response::function_gradients(), Response::function_hessians(), Response::function_values(), Iterator::iteratedModel, SNLLBase::lastEvalMode, SNLLBase::lastEvalVars, SNLLBase::lastFnEvalLocn, Optimizer::numObjectiveFns, Iterator::outputLevel, ActiveSet::request_values(), and SNLLOptimizer::snllOptInstance.

Referenced by SNLLOptimizer::SNLLOptimizer().

The documentation for this class was generated from the following files:

- SNLLOptimizer.H
- SNLLOptimizer.C

43.128 SOLBase Class Reference

Base class for Stanford SOL software. Inheritance diagram for SOLBase::

Public Member Functions

- SOLBase ()

  default constructor

- SOLBase (Model &model)

  standard constructor

- ~SOLBase ()

  destructor

Protected Member Functions

- void allocate_arrays (const int &num_cv, const size_t &num_nln_con, const RealMatrix &lin_ineq_coeffs, const RealMatrix &lin_eq_coeffs)

  Allocates miscellaneous arrays for the SOL algorithms.

- void deallocate_arrays ()

  Deallocates memory previously allocated by allocate_arrays().

- void allocate_workspace (const int &num_cv, const int &num_nln_con, const int &num_lin_con, const int &num_lsq)

  Allocates real and integer workspaces for the SOL algorithms.

- void set_options (bool speculative_flag, bool vendor_num_grad_flag, short output_lev, const int &verify_-lev, const Real &fn_prec, const Real &linesrch_tol, const int &max_iter, const Real &constr_tol, const Real &conv_tol, const std::string &grad_type, const Real &fdss)

  Sets SOL method options using calls to npoptn2.

- void augment_bounds (RealVector &augmented_l_bnds, RealVector &augmented_u_bnds, const RealVector &lin_ineq_l_bnds, const RealVector &lin_ineq_u_bnds, const RealVector &lin_eq_targets, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_targets)

  augments variable bounds with linear and nonlinear constraint bounds.

Static Protected Member Functions

- static void constraint_eval (int &mode, int &ncnln, int &n, int &nrowj, int *needc, double *x, double *c, double *cjac, int &nstate)

  CONFUN in NPSOL manual: computes the values and first derivatives of the nonlinear constraint functions.

Protected Attributes

- int realWorkSpaceSize

  size of realWorkSpace

- int intWorkSpaceSize

  size of intWorkSpace

- RealArray realWorkSpace

  real work space for NPSOL/NLSSOL

- IntArray intWorkSpace

  int work space for NPSOL/NLSSOL

- int nlnConstraintArraySize

  used for non-zero array sizing (nonlinear constraints)

- int linConstraintArraySize

  used for non-zero array sizing (linear constraints)

- RealArray cLambda

  CLAMBDA from NPSOL manual: Langrange multipliers.

- IntArray constraintState

  ISTATE from NPSOL manual: constraint status.

- int informResult

  INFORM from NPSOL manual: optimization status on exit.

- int numberIterations

  ITER from NPSOL manual: number of (major) iterations performed.

- int boundsArraySize

  length of augmented bounds arrays (variable bounds plus linear and nonlinear constraint bounds)

- double * linConstraintMatrixF77

  [A] matrix from NPSOL manual: linear constraint coefficients

- double * upperFactorHessianF77

[R] matrix from NPSOL manual: upper Cholesky factor of the Hessian of the Lagrangian.

- double * constraintJacMatrixF77

  [CJAC] matrix from NPSOL manual: nonlinear constraint Jacobian

- int fnEvalCntr

  counter for testing against maxFunctionEvals

- size_t constrOffset

  used in constraint_eval() to bridge NLSSOLLeastSq::numLeastSqTerms and NPSOLOptimizer::numObjectiveFns

Static Protected Attributes

- static SOLBase * solInstance

  pointer to the active object instance used within the static evaluator functions in order to avoid the need for static data

- static Minimizer * optLSqInstance

  pointer to the active base class object instance used within the static evaluator functions in order to avoid the need for static data

43.128.1 Detailed Description

Base class for Stanford SOL software. The SOLBase class provides a common base class for NPSOLOptimizer and NLSSOLLeastSq, both of which are Fortran 77 sequential quadratic programming algorithms from Stanford University marketed by Stanford Business Associates.

The documentation for this class was generated from the following files:

- SOLBase.H
- SOLBase.C

43.129 Strategy Class Reference

Base class for the strategy class hierarchy. Inheritance diagram for Strategy::

Public Member Functions

- Strategy ()

  default constructor

- Strategy (ProblemDescDB &problem_db)

  envelope constructor

- Strategy (const Strategy &strat)

  copy constructor

- virtual ~Strategy ()

  destructor

- Strategy operator= (const Strategy &strat)

  assignment operator

- virtual void run_strategy ()

  the run function for the strategy: invoke the iterator(s) on the model(s). Called from main.C.

- virtual const Variables & variables_results () const

  return the final strategy solution (variables)

- virtual const Response & response_results () const

  return the final strategy solution (response)

- ProblemDescDB & problem_description_db () const

  returns the problem description database (probDescDB)

Protected Member Functions

- Strategy (BaseConstructor, ProblemDescDB &problem_db)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- virtual void initialize_iterator (int index)

  initialize the iterator about to be executed within a parallel iterator scheduling function (serve_iterators() or static_schedule_iterators())

- virtual void pack_parameters_buffer (MPIPackBuffer &send_buffer, int job_index)

  pack a send_buffer for assigning an iterator job to a server

- virtual void unpack_parameters_buffer (MPIUnpackBuffer &recv_buffer)

  unpack a recv_buffer for accepting an iterator job from the scheduler

- virtual void pack_results_buffer (MPIPackBuffer &send_buffer, int job_index)

  pack a send_buffer for returning iterator results from a server

- virtual void unpack_results_buffer (MPIUnpackBuffer &recv_buffer, int job_index)

  unpack a recv_buffer for accepting iterator results from a server

- virtual void update_local_results (int job_index)

  update local PRP results arrays with current iteration results

- void init_iterator_parallelism ()

  convenience function for initializing iterator communicators, setting parallel configuration attributes, and managing outputs and restart.

- void init_iterator (Iterator &the_iterator, Model &the_model)

  convenience function for allocating comms prior to running an iterator

- void run_iterator (Iterator &the_iterator, Model &the_model)

  Convenience function for invoking an iterator and managing parallelism. This version omits communicator repartitioning. Function must be public due to use by MINLPNode.

- void free_iterator (Iterator &the_iterator, Model &the_model)

  convenience function for deallocating comms after running an iterator

- void schedule_iterators (Iterator &the_iterator, Model &the_model)

  short convenience function for distributing control among self_schedule_iterators(), serve_iterators(), and static_schedule_iterators()

- void self_schedule_iterators (Model &the_model)

  executed by the strategy master to self-schedule iterator jobs among slave iterator servers (called by derived run_strategy())

- void serve_iterators (Iterator &the_iterator, Model &the_model)

  executed on the slave iterator servers to perform iterator jobs assigned by the strategy master (called by derived run_strategy())

- void static_schedule_iterators (Iterator &the_iterator, Model &the_model)

  executed on iterator peers to statically schedule iterator jobs (called by derived run_strategy())

Protected Attributes

- ProblemDescDB & probDescDB

  class member reference to the problem description database

- ParallelLibrary & parallelLib

  class member reference to the parallel library

- String strategyName

  type of strategy: single_method, hybrid, multi_start, or pareto_set.

- bool stratIterMessagePass

  flag for message passing at si level

- bool stratIterDedMaster

  flag for dedicated master part. at si level

- int worldRank

  processor rank in MPI_COMM_WORLD

- int worldSize

  size of MPI_COMM_WORLD

- int iteratorCommRank

  processor rank in iteratorComm

- int iteratorCommSize

  number of processors in iteratorComm

- int numIteratorServers

  number of concurrent iterator partitions

- int iteratorServerId

  identifier for an iterator server

- bool graph2DFlag

  flag for using 2D graphics plots

- bool tabularDataFlag

  flag for file tabulation of graphics data

- String tabularDataFile

  filename for tabulation of graphics data

- int maxConcurrency

  maximum iterator concurrency possible in Strategy

- int numIteratorJobs

  number of iterator executions to schedule

- int paramsMsgLen

  length of MPI buffer for parameter input instance(s)

- int resultsMsgLen

  length of MPI buffer for results output instance(s)

Private Member Functions

- Strategy * get_strategy ()

  Used by the envelope to instantiate the correct letter class.

Private Attributes

- Strategy * strategyRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing strategyRep

43.129.1 Detailed Description

Base class for the strategy class hierarchy. The Strategy class is the base class for the class hierarchy providing the top level control in DAKOTA. The strategy is responsible for creating and managing iterators and models. For memory efficiency and enhanced polymorphism, the strategy hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Strategy) serves as the envelope and one of the derived classes (selected in Strategy::get_strategy()) serves as the letter.

43.129.2 Constructor & Destructor Documentation

43.129.2.1 Strategy ()

default constructor Default constructor. strategyRep is NULL in this case (a populated problem_db is needed to build a meaningful Strategy object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

43.129.2.2 Strategy (ProblemDescDB & *problem_db*)

envelope constructor Used in main.C instantiation to build the envelope. This constructor only needs to extract enough data to properly execute get_strategy, since Strategy::Strategy(BaseConstructor, problem_db) builds the actual base class data inherited by the derived strategies.

References Dakota::abort_handler(), Strategy::get_strategy(), and Strategy::strategyRep.

43.129.2.3 Strategy (const Strategy & *strat*)

copy constructor Copy constructor manages sharing of strategyRep and incrementing of referenceCount.

References Strategy::referenceCount, and Strategy::strategyRep.

43.129.2.4 ~Strategy () `[virtual]`

destructor Destructor decrements referenceCount and only deletes strategyRep when referenceCount reaches zero.

References ParallelLibrary::free_iterator_communicators(), ParallelLibrary::is_null(), Strategy::parallelLib, Strategy::referenceCount, and Strategy::strategyRep.

43.129.2.5 Strategy (BaseConstructor, ProblemDescDB & *problem_db*) `[protected]`

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all inherited strategies. get_strategy() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_strategy() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~Strategy).

References ProblemDescDB::get_int(), Strategy::probDescDB, and Dakota::write_precision.

43.129.3 Member Function Documentation

43.129.3.1 Strategy operator= (const Strategy & *strat*)

assignment operator Assignment operator decrements referenceCount for old strategyRep, assigns new strategyRep, and increments referenceCount for new strategyRep.

References Strategy::referenceCount, and Strategy::strategyRep.

43.129.3.2 void pack_parameters_buffer (MPIPackBuffer & *send_buffer*, int *job_index*) [protected, virtual]

pack a send_buffer for assigning an iterator job to a server This virtual function redefinition is executed on the dedicated master processor for self scheduling. It is not used for peer partitions.

Reimplemented in ConcurrentStrategy, and SequentialHybridStrategy.

References Strategy::pack_parameters_buffer(), and Strategy::strategyRep.

Referenced by Strategy::pack_parameters_buffer(), and Strategy::self_schedule_iterators().

43.129.3.3 void unpack_parameters_buffer (MPIUnpackBuffer & *recv_buffer*) [protected, virtual]

unpack a recv_buffer for accepting an iterator job from the scheduler This virtual function redefinition is executed on an iterator server for dedicated master self scheduling. It is not used for peer partitions.

Reimplemented in ConcurrentStrategy, and SequentialHybridStrategy.

References Strategy::strategyRep, and Strategy::unpack_parameters_buffer().

Referenced by Strategy::serve_iterators(), and Strategy::unpack_parameters_buffer().

43.129.3.4 void pack_results_buffer (MPIPackBuffer & *send_buffer*, int *job_index*) [protected, virtual]

pack a send_buffer for returning iterator results from a server This virtual function redefinition is executed either on an iterator server for dedicated master self scheduling or on peers 2 through n for static scheduling.

Reimplemented in ConcurrentStrategy, and SequentialHybridStrategy.

References Strategy::pack_results_buffer(), and Strategy::strategyRep.

Referenced by Strategy::pack_results_buffer(), Strategy::serve_iterators(), and Strategy::static_schedule_-iterators().

43.129.3.5 void unpack_results_buffer (MPIUnpackBuffer & *recv_buffer*, int *job_index*) [protected, virtual]

unpack a recv_buffer for accepting iterator results from a server This virtual function redefinition is executed on an strategy master (either the dedicated master processor for self scheduling or peer 1 for static scheduling).

Reimplemented in ConcurrentStrategy, and SequentialHybridStrategy.

References Strategy::strategyRep, and Strategy::unpack_results_buffer().

Referenced by Strategy::self_schedule_iterators(), Strategy::static_schedule_iterators(), and Strategy::unpack_-results_buffer().

43.129.3.6 void init_iterator_parallelism () [protected]

convenience function for initializing iterator communicators, setting parallel configuration attributes, and managing outputs and restart. This function is called from derived class constructors once maxConcurrency is defined but prior to instantiating Iterators and Models.

References ParallelLevel::dedicated_master_flag(), ProblemDescDB::get_int(), ProblemDescDB::get_string(), ParallelLibrary::init_iterator_communicators(), Strategy::iteratorCommRank, Strategy::iteratorCommSize, Strategy::iteratorServerId, ParallelLibrary::manage_outputs_restart(), Strategy::maxConcurrency, ParallelLevel::message_pass(), ParallelLevel::num_servers(), Strategy::numIteratorServers, Strategy::parallelLib, Strategy::probDescDB, ParallelLevel::server_communicator_rank(), ParallelLevel::server_communicator_size(), ParallelLevel::server_id(), Strategy::stratIterDedMaster, and Strategy::stratIterMessagePass.

Referenced by CollaborativeHybridStrategy::CollaborativeHybridStrategy(), ConcurrentStrategy::ConcurrentStrategy(), EmbeddedHybridStrategy::EmbeddedHybridStrategy(), SequentialHybridStrategy::SequentialHybridStrategy(), and SingleMethodStrategy::SingleMethodStrategy().

43.129.3.7 void init_iterator (Iterator & *the_iterator*, Model & *the_model*) [protected]

convenience function for allocating comms prior to running an iterator This is a convenience function for encapsulating the allocation of communicators prior to running an iterator. It does not require a strategyRep forward since it is only used by letter objects.

References ProblemDescDB::get_iterator(), Model::init_comms_bcast_flag(), Model::init_communicators(), Strategy::iteratorCommRank, Strategy::iteratorCommSize, Iterator::maximum_concurrency(), Strategy::probDescDB, Model::serve_configurations(), and Model::stop_configurations().

Referenced by HybridStrategy::allocate_methods(), ConcurrentStrategy::ConcurrentStrategy(), and SingleMethodStrategy::SingleMethodStrategy().

43.129.3.8 void run_iterator (Iterator & *the_iterator*, Model & *the_model*) [protected]

Convenience function for invoking an iterator and managing parallelism. This version omits communicator repartitioning. Function must be public due to use by MINLPNode. This is a convenience function for encapsulating the parallel features (run/serve) of running an iterator. This function omits allocation/deallocation of communicators to provide greater efficiency in those strategies which involve multiple iterator executions but only require communicator allocation/deallocation to be performed once.

It does not require a strategyRep forward since it is only used by letter objects. While it is currently a public function due to its use in MINLPNode, this usage still involves a strategy letter object.

References Strategy::iteratorCommRank, Iterator::maximum_concurrency(), Iterator::run_iterator(), Model::serve(), Model::set_communicators(), and Model::stop_servers().

Referenced by SequentialHybridStrategy::run_sequential_adaptive(), SingleMethodStrategy::run_strategy(), Strategy::serve_iterators(), and Strategy::static_schedule_iterators().

43.129.3.9 void free_iterator (Iterator & *the_iterator*, Model & *the_model*) [protected]

convenience function for deallocating comms after running an iterator This is a convenience function for encapsulating the deallocation of communicators after running an iterator. It does not require a strategyRep forward

since it is only used by letter objects.

References Model::free_communicators(), and Iterator::maximum_concurrency().

Referenced by HybridStrategy::deallocate_methods(), ConcurrentStrategy::∼ConcurrentStrategy(), and SingleMethodStrategy::∼SingleMethodStrategy().

43.129.3.10 void schedule_iterators (Iterator & *the_iterator*, Model & *the_model*) [protected]

short convenience function for distributing control among self_schedule_iterators(), serve_iterators(), and static_schedule_iterators() This implementation supports the scheduling of multiple jobs using a single iterator/model pair. Additional future (overloaded) implementations could involve independent iterator instances.

References Strategy::self_schedule_iterators(), Strategy::serve_iterators(), Strategy::static_schedule_iterators(), Strategy::stratIterDedMaster, and Strategy::worldRank.

Referenced by SequentialHybridStrategy::run_sequential(), EmbeddedHybridStrategy::run_strategy(), ConcurrentStrategy::run_strategy(), and CollaborativeHybridStrategy::run_strategy().

43.129.3.11 void self_schedule_iterators (Model & *the_model*) [protected]

executed by the strategy master to self-schedule iterator jobs among slave iterator servers (called by derived run_strategy()) This function is adapted from ApplicationInterface::self_schedule_evaluations().

References ParallelLibrary::free(), ParallelLibrary::irecv_si(), ParallelLibrary::isend_si(), Strategy::numIteratorJobs, Strategy::numIteratorServers, Strategy::pack_parameters_buffer(), Strategy::parallelLib, ParallelLibrary::print_configuration(), MPIPackBuffer::reset(), MPIUnpackBuffer::resize(), Strategy::resultsMsgLen, Strategy::unpack_results_buffer(), ParallelLibrary::waitall(), and ParallelLibrary::waitsome().

Referenced by Strategy::schedule_iterators().

43.129.3.12 void serve_iterators (Iterator & *the_iterator*, Model & *the_model*) [protected]

executed on the slave iterator servers to perform iterator jobs assigned by the strategy master (called by derived run_strategy()) This function is similar in structure to ApplicationInterface::serve_evaluations_synch().

References ParallelLibrary::bcast_i(), Strategy::iteratorCommRank, Strategy::iteratorCommSize, Strategy::pack_results_buffer(), ParallelLibrary::parallel_time(), Strategy::parallelLib, Strategy::paramsMsgLen, ParallelLibrary::recv_si(), Strategy::resultsMsgLen, Strategy::run_iterator(), ParallelLibrary::send_si(), Strategy::unpack_parameters_buffer(), and Strategy::update_local_results().

Referenced by Strategy::schedule_iterators().

43.129.3.13 Strategy ∗ get_strategy () [private]

Used by the envelope to instantiate the correct letter class. Used only by the envelope constructor to initialize strategyRep to the appropriate derived type, as given by the strategyName attribute.

References String::begins(), ProblemDescDB::get_string(), Strategy::probDescDB, and Strategy::strategyName.

Referenced by Strategy::Strategy().

The documentation for this class was generated from the following files:

- DakotaStrategy.H
- DakotaStrategy.C

43.130 String Class Reference

Dakota::String class, used as main string class for Dakota.

Public Member Functions

- String ()

  Default constructor.

- String (const String &a)

  Copy constructor for incoming String.

- String (const String &a, size_t start_index, size_t num_items)

  Copy constructor for portion of incoming String.

- String (const char *c_string)

  Copy constructor for incoming char array.*

- String (const std::string &a)

  Copy constructor for incoming base string.

- ~String ()

  Destructor.

- String & operator= (const String &)

  Assignment operator for incoming String.

- String & operator= (const std::string &)

  Assignment operator for incoming base string.

- String & operator= (const char *)

  Assignment operator for incoming char array.*

- operator const char * () const

  The operator() returns pointer to standard C char array.

- String & toUpper ()

  Convert to upper case string.

- void upper ()
- String & toLower ()

  Convert to lower case string.

- void lower ()
- bool contains (const char *sub_string) const

Returns true if String contains char∗ substring.

- bool begins (const char ∗sub_string) const

  Returns true if String starts with char∗ substring.

- bool ends (const char ∗sub_string) const

  Returns true if String ends with char∗ substring.

- char ∗ data () const

  Returns pointer to standard C char array.

43.130.1 Detailed Description

Dakota::String class, used as main string class for Dakota. The Dakota::String class is the common string class for Dakota. It provides a common interface for string operations whether using the std::string interface or the (legacy) RogueWave RWCString API

43.130.2 Member Function Documentation

43.130.2.1 operator const char ∗ () const `[inline]`

The operator() returns pointer to standard C char array. The operator () returns a pointer to a char string. Uses the STL c_str() method. This allows for the String to be used in method calls without having to call the data() or c_str() methods.

43.130.2.2 void upper ()

Private method which converts String to upper. Utilizes an STL iterator to step through the string and then calls the STL toupper() method. Needs to be done this way because STL only provides a single char toupper method.

Referenced by Dakota::toUpper(), and String::toUpper().

43.130.2.3 void lower ()

Private method which converts String to lower. Utilizes an STL iterator to step through the string and then calls the STL tolower() method. Needs to be done this way because STL only provides a single char tolower method.

Referenced by Dakota::toLower(), and String::toLower().

43.130.2.4 bool contains (const char ∗ *sub_string*) const `[inline]`

Returns true if String contains char∗ substring. Returns true if the String contains the char∗ sub_string. Uses the STL find() method.

Referenced by Interface::algebraic_function_type().

43.130.2.5 bool begins (const char ∗ *sub_string*) const [inline]

Returns true if String starts with char∗ substring. Returns true if the String begins with the char∗ sub_string. Uses the STL compare() method.

Referenced by PecosApproximation::approx_type_to_basis_type(), DataFitSurrModel::approximation_coefficients(), DataFitSurrModel::build_local_multipoint(), ProblemDescDB::check_input(), DataFitSurrModel::DataFitSurrModel(), SurrogateModel::force_rebuild(), ProblemDescDB::get_dis(), ProblemDescDB::get_ds2a(), Iterator::get_iterator(), ProblemDescDB::get_real(), ProblemDescDB::get_rsdm(), Strategy::get_strategy(), ProblemDescDB::get_string(), ProblemDescDB::get_ushort(), Variables::get_view(), NLPQLPOptimizer::initialize(), NCSUOptimizer::initialize(), DOTOptimizer::initialize(), CONMINOptimizer::initialize(), SurrBasedMinimizer::initialize_graphics(), Minimizer::Minimizer(), Optimizer::Optimizer(), ParamStudy::ParamStudy(), SurrBasedMinimizer::print_results(), ProblemDescDB::set(), and SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.130.2.6 bool ends (const char ∗ *sub_string*) const [inline]

Returns true if String ends with char∗ substring. Returns true if the String ends with the char∗ sub_string. Uses the STL compare() method.

Referenced by PecosApproximation::approx_type_to_basis_type(), ProblemDescDB::check_input(), DataFitSurrModel::DataFitSurrModel(), Approximation::get_approx(), Iterator::get_iterator(), ProblemDescDB::get_ushort(), Variables::get_view(), Interface::Interface(), Minimizer::Minimizer(), NonDLocalInterval::NonDLocalInterval(), NonDLocalReliability::NonDLocalReliability(), PStudyDACE::PStudyDACE(), SequentialHybridStrategy::run_strategy(), SequentialHybridStrategy::SequentialHybridStrategy(), and SOLBase::SOLBase().

43.130.2.7 char ∗ data () const [inline]

Returns pointer to standard C char array. Returns a pointer to C style char array. Needed to mimic the Rogue Wave string class. USE WITH CARE.

Referenced by GridApplicInterface::derived_map_asynch(), GridApplicInterface::grid_file_test(), Interface::Interface(), Dakota::print_restart_tabular(), and ConcurrentStrategy::print_results().

The documentation for this class was generated from the following files:

- DakotaString.H
- DakotaString.C

43.131 SurfpackApproximation Class Reference

Derived approximation class for Surfpack approximation classes. Interface between Surfpack and Dakota. Inheritance diagram for SurfpackApproximation::

Public Member Functions

- SurfpackApproximation ()

  default constructor

- SurfpackApproximation (const String &approx_type, const UShortArray &approx_order, size_t num_vars, short data_order)

  alternate constructor

- SurfpackApproximation (const ProblemDescDB &problem_db, const size_t &num_acv)

  standard constructor: Surfpack surface of appropriate type will be created

- ~SurfpackApproximation ()

  destructor

Protected Member Functions

- int min_coefficients () const

  return the minimum number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- int recommended_coefficients () const

  return the recommended number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- void build ()

  SurfData object will be created from Dakota's SurrogateData, and the appropriate Surfpack build method will be invoked.

- Real get_value (const RealVector &x)

  Return the value of the Surfpack surface for a given parameter vector x.

- Real get_prediction_variance (const RealVector &x)

retrieve the variance of the predicted value for a given parameter set x (KrigingModel only)

- const RealVector & get_gradient (const RealVector &x)

  retrieve the approximate function gradient for a given parameter vector x

- const RealSymMatrix & get_hessian (const RealVector &x)

  retrieve the approximate function Hessian for a given parameter vector x

- Real get_diagnostic (const String &metric_type)

  retrieve the diagnostic metric for the diagnostic type specified

- const bool diagnostics_available ()

  check if the diagnostics are available (true for the Surfpack types)

Private Member Functions

- SurfData ∗ surrogates_to_surf_data ()

  copy from SurrogateData to SurfPoint/SurfData

- void add_anchor_to_surfdata (SurfData &surf_data)

  set the anchor point (including gradient and hessian if present) into surf_data

- void add_sdp_to_surfdata (const Pecos::SurrogateDataVars &sdv, const Pecos::SurrogateDataResp &sdr, SurfData &surf_data)

  add Pecos::SurrogateData::SurrogateData{Vars,Resp} to SurfData, accounting for buildDataOrder available

- void copy_matrix (const RealSymMatrix &rsdm, SurfpackMatrix< Real > &surfpack_matrix)

  copy RealSymMatrix to SurfpackMatrix (Real type only)

Private Attributes

- unsigned short approxOrder

  order of polynomial approximation

- SurfpackModel ∗ model

  The native Surfpack approximation.

- SurfpackModelFactory ∗ factory

  factory for the SurfpackModel instance

- SurfData ∗ surfData

  The data used to build the approximation, in Surfpack format.

43.131.1 Detailed Description

Derived approximation class for Surfpack approximation classes. Interface between Surfpack and Dakota. The SurfpackApproximation class is the interface between Dakota and Surfpack. Based on the information in the ProblemDescDB that is passed in through the constructor, SurfpackApproximation builds a Surfpack Surface object that corresponds to one of the following data-fitting techniques: polynomial regression, kriging, artificial neural networks, radial basis function network, or multivariate adaptaive regression splines (MARS).

43.131.2 Constructor & Destructor Documentation

43.131.2.1 SurfpackApproximation (const String & *approx_type*, const UShortArray & *approx_order*, size_t *num_vars*, short *data_order*)

alternate constructor On-the-fly constructor which uses mostly Surfpack model defaults.

References Dakota::abort_handler(), Approximation::approxLowerBounds, SurfpackApproximation::approxOrder, Approximation::approxType, Approximation::approxUpperBounds, Approximation::buildDataOrder, Dakota::copy_data(), and SurfpackApproximation::factory.

43.131.3 Member Function Documentation

43.131.3.1 void build () `[protected, virtual]`

SurfData object will be created from Dakota's SurrogateData, and the appropriate Surfpack build method will be invoked.

surfData will be deleted in dtor

Todo

Right now, we're completely deleting the old data and then recopying the current data into a SurfData object. This was just the easiest way to arrive at a solution that would build and run. This function is frequently called from addPoint rebuild, however, and it's not good to go through this whole process every time one more data point is added.

Reimplemented from Approximation.

References Dakota::abort_handler(), Approximation::approxLowerBounds, Approximation::approxUpperBounds, Dakota::copy_data(), SurfpackApproximation::factory, SurfpackApproximation::model, Approximation::outputLevel, SurfpackApproximation::surfData, and SurfpackApproximation::surrogates_to_surf_data().

43.131.3.2 const RealSymMatrix & get_hessian (const RealVector & *x*) `[protected, virtual]`

retrieve the approximate function Hessian for a given parameter vector x

Todo

Make this acceptably efficient

Reimplemented from Approximation.

References Dakota::abort_handler(), Approximation::approxHessian, Approximation::approxType, Dakota::copy_data(), and SurfpackApproximation::model.

43.131.3.3 SurfData * surrogates_to_surf_data () [private]

copy from SurrogateData to SurfPoint/SurfData Copy the data stored in Dakota-style SurrogateData into Surfpack-style SurfPoint and SurfData objects.

References SurfpackApproximation::add_anchor_to_surfdata(), SurfpackApproximation::add_sdp_to_-surfdata(), Approximation::approxData, Approximation::buildDataOrder, SurfpackApproximation::factory, and Approximation::outputLevel.

Referenced by SurfpackApproximation::build().

43.131.3.4 void add_anchor_to_surfdata (SurfData & *surf_data*) [private]

set the anchor point (including gradient and hessian if present) into surf_data If there is an anchor point, add an equality constraint for its response value. Also add constraints for gradient and hessian, if applicable.

Todo

improve efficiency of conversion

References Dakota::abort_handler(), Approximation::approxData, Dakota::copy_data(), Approximation::outputLevel, and Dakota::write_data().

Referenced by SurfpackApproximation::surrogates_to_surf_data().

The documentation for this class was generated from the following files:

- SurfpackApproximation.H
- SurfpackApproximation.C

43.132 SurrBasedGlobalMinimizer Class Reference

The global surrogate-based minimizer which sequentially minimizes and updates a global surrogate model without trust region controls. Inheritance diagram for SurrBasedGlobalMinimizer::

Public Member Functions

- SurrBasedGlobalMinimizer (Model &model)

  constructor

- ~SurrBasedGlobalMinimizer ()

  destructor

Protected Member Functions

- bool returns_multiple_points () const

  Global surrogate-based methods can return multiple points.

Private Member Functions

- void minimize_surrogates ()

  Performs global surrogate-based optimization by repeatedly optimizing on and improving surrogates of the response functions.

Private Attributes

- bool replacePoints

  flag for replacing the previous iteration's point additions, rather than continuing to append, during construction of the next surrogate

43.132.1 Detailed Description

The global surrogate-based minimizer which sequentially minimizes and updates a global surrogate model without trust region controls. This method uses a SurrogateModel to perform minimization (optimization or nonlinear least squares) through a set of iterations. At each iteration, a surrogate is built, the surrogate is minimized, and the optimal points from the surrogate are then evaluated with the "true" function, to generate new points upon which the surrogate for the next iteration is built.

The documentation for this class was generated from the following files:

- SurrBasedGlobalMinimizer.H
- SurrBasedGlobalMinimizer.C

43.133 SurrBasedLocalMinimizer Class Reference

Class for provably-convergent local surrogate-based optimization and nonlinear least squares. Inheritance diagram for SurrBasedLocalMinimizer::

Public Member Functions

- SurrBasedLocalMinimizer (Model &model)

  constructor

- ~SurrBasedLocalMinimizer ()

  destructor

Protected Member Functions

- void reset ()

  reset convergence controls in case of multiple SBLM executions

Private Member Functions

- void minimize_surrogates ()

  Performs local surrogate-based minimization by minimizing local, global, or hierarchical surrogates over a series of trust regions.

- bool tr_bounds (const RealVector &global_lower_bnds, const RealVector &global_upper_bnds, RealVector &tr_lower_bnds, RealVector &tr_upper_bnds)

  compute current trust region bounds

- void find_center_truth (const Iterator &dace_iterator, Model &truth_model)

  retrieve responseCenterTruth if possible, evaluate it if not

- void find_center_approx ()

retrieve responseCenter_approx if possible, evaluate it if not

- void hard_convergence_check (const Response &response_truth, const RealVector &c_vars, const RealVector &lower_bnds, const RealVector &upper_bnds)

  check for hard convergence (norm of projected gradient of merit function near zero)

- void tr_ratio_check (const RealVector &c_vars_star, const RealVector &tr_lower_bounds, const RealVector &tr_upper_bounds)

  compute trust region ratio (for SBLM iterate acceptance and trust region resizing) and check for soft convergence (diminishing returns)

- void update_penalty (const RealVector &fns_center_truth, const RealVector &fns_star_truth)

  initialize and update the penaltyParameter

- void relax_constraints (const RealVector &lower_bnds, const RealVector &upper_bnds)

  relax constraints by updating bounds when current iterate is infeasible

Static Private Member Functions

- static void approx_subprob_objective_eval (const Variables &surrogate_vars, const Variables &recast_vars, const Response &surrogate_response, Response &recast_response)

  static function used to define the approximate subproblem objective.

- static void approx_subprob_constraint_eval (const Variables &surrogate_vars, const Variables &recast_vars, const Response &surrogate_response, Response &recast_response)

  static function used to define the approximate subproblem constraints.

- static void hom_objective_eval (int &mode, int &n, double *tau_and_x, double &f, double *grad_f, int &)

  static function used by NPSOL as the objective function in the homotopy constraint relaxation formulation.

- static void hom_constraint_eval (int &mode, int &ncnln, int &n, int &nrowj, int *needc, double *tau_and_x, double *c, double *cjac, int &nstate)

  static function used by NPSOL as the constraint function in the homotopy constraint relaxation formulation.

Private Attributes

- Real origTrustRegionFactor

  original user specification for trustRegionFactor

- Real trustRegionFactor

  the trust region factor is used to compute the total size of the trust region -- it is a percentage, e.g. for trustRegionFactor = 0.1, the actual size of the trust region will be 10% of the global bounds (upper bound - lower bound for each design variable).

- Real minTrustRegionFactor

  a soft convergence control: stop SBLM when the trust region factor is reduced below the value of minTrustRegionFactor

- Real trRatioContractValue

  trust region ratio min value: contract tr if ratio below this value

- Real trRatioExpandValue

  trust region ratio sufficient value: expand tr if ratio above this value

- Real gammaContract

  trust region contraction factor

- Real gammaExpand

  trust region expansion factor

- short approxSubProbObj

  type of approximate subproblem objective: ORIGINAL_OBJ, LAGRANGIAN_OBJ, or AUGMENTED_LAGRANGIAN_OBJ

- short approxSubProbCon

  type of approximate subproblem constraints: NO_CON, LINEARIZED_CON, or ORIGINAL_CON

- Model approxSubProbModel

  the approximate sub-problem formulation solved on each approximate minimization cycle: may be a shallow copy of iteratedModel, or may involve a RecastModel recursion applied to iteratedModel

- bool recastSubProb

  flag to indicate when approxSubProbModel involves a RecastModel recursion

- short trConstraintRelax

  type of trust region constraint relaxation for infeasible starting points: NO_RELAX or HOMOTOPY

- short meritFnType

  type of merit function used in trust region ratio logic: PENALTY_MERIT, ADAPTIVE_PENALTY_MERIT, LAGRANGIAN_MERIT, or AUGMENTED_LAGRANGIAN_MERIT

- short acceptLogic

  type of iterate acceptance test logic: FILTER or TR_RATIO

- int penaltyIterOffset

  iteration offset used to update the scaling of the penalty parameter for adaptive_penalty merit functions

- short convergenceFlag

  code indicating satisfaction of hard or soft convergence conditions

- short softConvCount

  number of consecutive candidate point rejections. If the count reaches softConvLimit, stop SBLM.

- short softConvLimit

  the limit on consecutive candidate point rejections. If exceeded by softConvCount, stop SBLM.

- bool truthGradientFlag

  flags the use/availability of truth gradients within the SBLM process

- bool approxGradientFlag

  flags the use/availability of surrogate gradients within the SBLM process

- bool truthHessianFlag

  flags the use/availability of truth Hessians within the SBLM process

- bool approxHessianFlag

  flags the use/availability of surrogate Hessians within the SBLM process

- short correctionType

  flags the use of surrogate correction techniques at the center of each trust region

- bool globalApproxFlag

  flags the use of a global data fit surrogate (rsm, ann, mars, kriging)

- bool multiptApproxFlag

  flags the use of a multipoint data fit surrogate (TANA)

- bool localApproxFlag

  flags the use of a local data fit surrogate (Taylor series)

- bool hierarchApproxFlag

  flags the use of a model hierarchy/multifidelity surrogate

- bool newCenterFlag

  flags the acceptance of a candidate point and the existence of a new trust region center

- bool daceCenterPtFlag

  flags the availability of the center point in the DACE evaluations for global approximations (CCD, Box-Behnken)

- bool multiLayerBypassFlag

  flags the simultaneous presence of two conditions: (1) additional layerings w/i actual_model (e.g., surrogateModel = layered/nested/layered -> actual_model = nested/layered), and (2) a user-specification to bypass all layerings within actual_model for the evaluation of truth data (responseCenterTruth and responseStarTruth).

- bool useDerivsFlag

  flag for the "use_derivatives" specification for which derivatives are to be evaluated at each DACE point in global surrogate builds.

- RealVector nonlinIneqLowerBndsSlack

  individual violations of nonlinear inequality constraint lower bounds

- RealVector nonlinIneqUpperBndsSlack

  individual violations of nonlinear inequality constraint upper bounds

- RealVector nonlinEqTargetsSlack

  individual violations of nonlinear equality constraint targets

- Real tau

  constraint relaxation parameter

- Real alpha

  constraint relaxation parameter backoff parameter (multiplier)

- Variables varsCenter

  variables at the trust region center

- Response responseCenterApprox

  approx response at trust region center

- Response responseStarApprox

  approx response at SBLM cycle minimum

- IntResponsePair responseCenterTruth

  truth response at trust region center

- IntResponsePair responseStarTruth

  truth response at SBLM cycle minimum

Static Private Attributes

- static SurrBasedLocalMinimizer $*$ sblmInstance

  pointer to SBLM instance used in static member functions

43.133.1 Detailed Description

Class for provably-convergent local surrogate-based optimization and nonlinear least squares. This minimizer uses a SurrogateModel to perform minimization based on local, global, or hierarchical surrogates. It achieves provable convergence through the use of a sequence of trust regions and the application of surrogate corrections at the trust region centers.

43.133.2 Member Function Documentation

43.133.2.1 void minimize_surrogates () [private, virtual]

Performs local surrogate-based minimization by minimizing local, global, or hierarchical surrogates over a series of trust regions. Trust region-based strategy to perform surrogate-based optimization in subregions (trust regions) of the parameter space. The minimizer operates on approximations in lieu of the more expensive simulation-based response functions. The size of the trust region is varied according to the goodness of the agreement between the approximations and the true response functions.

Implements SurrBasedMinimizer.

References Dakota::abort_handler(), Iterator::active_set(), Response::active_set(), Model::active_variables(), Graphics::add_datapoint(), DiscrepancyCorrection::apply(), SurrBasedLocalMinimizer::approxGradientFlag, SurrBasedLocalMinimizer::approxHessianFlag, SurrBasedMinimizer::approxSubProbMinimizer, SurrBasedLocalMinimizer::approxSubProbModel, Iterator::bestResponseArray, Iterator::bestVariablesArray, Model::build_approximation(), Model::component_parallel_mode(), DiscrepancyCorrection::compute(), Model::compute_response(), Model::continuous_lower_bounds(), Model::continuous_upper_bounds(), Model::continuous_variables(), Variables::continuous_variables(), SurrBasedLocalMinimizer::convergenceFlag, Variables::copy(), Dakota::copy_data(), SurrBasedLocalMinimizer::correctionType, Model::current_response(), Model::current_variables(), SurrBasedLocalMinimizer::daceCenterPtFlag, Dakota::dakota_graphics, Model::discrepancy_correction(), Model::evaluation_id(), SurrBasedLocalMinimizer::find_center_approx(), SurrBasedLocalMinimizer::find_center_truth(), SurrBasedLocalMinimizer::globalApproxFlag, SurrBasedLocalMinimizer::hard_convergence_check(), Iterator::is_null(), Iterator::iteratedModel, SurrBasedLocalMinimizer::localApproxFlag, Iterator::maxIterations, SurrBasedLocalMinimizer::minTrustRegionFactor, SurrBasedLocalMinimizer::multiLayerBypassFlag, SurrBasedLocalMinimizer::multiptApproxFlag, SurrBasedLocalMinimizer::newCenterFlag, Model::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Iterator::numContinuousVars, SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, SurrBasedLocalMinimizer::recastSubProb, SurrBasedLocalMinimizer::relax_constraints(), ActiveSet::request_values(), SurrBasedLocalMinimizer::reset(), Iterator::response_results(), SurrBasedLocalMinimizer::responseCenterApprox, SurrBasedLocalMinimizer::responseCenterTruth, SurrBasedLocalMinimizer::responseStarApprox, SurrBasedLocalMinimizer::responseStarTruth, Iterator::run_iterator(), Iterator::sampling_scheme(), SurrBasedMinimizer::sbIterNum, SurrBasedLocalMinimizer::sblmInstance, SurrBasedLocalMinimizer::softConvCount, SurrBasedLocalMinimizer::softConvLimit, Model::subordinate_iterator(), Model::surrogate_model(), Model::surrogate_response_mode(), SurrBasedLocalMinimizer::tr_bounds(), SurrBasedLocalMinimizer::tr_ratio_check(), SurrBasedLocalMinimizer::trConstraintRelax, SurrBasedLocalMinimizer::trustRegionFactor, Model::truth_model(), SurrBasedLocalMinimizer::truthGradientFlag, SurrBasedLocalMinimizer::truthHessianFlag, Response::update(), SurrBasedLocalMinimizer::useDerivsFlag, Iterator::variables_results(), and SurrBasedLocalMinimizer::varsCenter.

43.133.2.2 void hard_convergence_check (const Response & *response_truth*, const RealVector & *c_vars*, const RealVector & *lower_bnds*, const RealVector & *upper_bnds*) [private]

check for hard convergence (norm of projected gradient of merit function near zero) The hard convergence check computes the gradient of the merit function at the trust region center, performs a projection for active bound constraints (removing any gradient component directed into an active bound), and signals convergence if the 2-norm of this projected gradient is less than convergenceTol.

References SurrBasedLocalMinimizer::acceptLogic, SurrBasedLocalMinimizer::approxSubProbObj, SurrBasedMinimizer::constraint_violation(), Minimizer::constraintTol, SurrBasedLocalMinimizer::convergenceFlag, Iterator::convergenceTol, Response::function_gradients(), Response::function_values(), Iterator::iteratedModel, SurrBasedMinimizer::lagrangian_gradient(), SurrBasedLocalMinimizer::meritFnType, Iterator::numContinuousVars, Minimizer::numNonlinearConstraints, SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, Model::primary_response_fn_weights(), SurrBasedMinimizer::sbIterNum, SurrBasedLocalMinimizer::truthGradientFlag, SurrBasedMinimizer::update_augmented_lagrange_multipliers(), SurrBasedMinimizer::update_filter(), and SurrBasedMinimizer::update_lagrange_multipliers().

Referenced by SurrBasedLocalMinimizer::minimize_surrogates().

43.133.2.3 void tr_ratio_check (const RealVector & *c_vars_star*, const RealVector & *tr_lower_bnds*, const RealVector & *tr_upper_bnds*) `[private]`

compute trust region ratio (for SBLM iterate acceptance and trust region resizing) and check for soft convergence (diminishing returns) Assess acceptance of SBLM iterate (trust region ratio or filter) and compute soft convergence metrics (number of consecutive failures, min trust region size, etc.) to assess whether the convergence rate has decreased to a point where the process should be terminated (diminishing returns).

References SurrBasedLocalMinimizer::acceptLogic, SurrBasedLocalMinimizer::approxSubProbObj, SurrBasedMinimizer::augmented_lagrangian_merit(), SurrBasedMinimizer::constraint_violation(), Minimizer::constraintTol, Iterator::convergenceTol, SurrBasedMinimizer::etaSequence, Response::function_values(), SurrBasedLocalMinimizer::gammaContract, SurrBasedLocalMinimizer::gammaExpand, SurrBasedLocalMinimizer::globalApproxFlag, Iterator::iteratedModel, SurrBasedMinimizer::lagrangian_merit(), SurrBasedLocalMinimizer::meritFnType, SurrBasedLocalMinimizer::newCenterFlag, Iterator::numContinuousVars, SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, SurrBasedMinimizer::penalty_merit(), Model::primary_response_fn_weights(), SurrBasedLocalMinimizer::responseCenterApprox, SurrBasedLocalMinimizer::responseCenterTruth, SurrBasedLocalMinimizer::responseStarApprox, SurrBasedLocalMinimizer::responseStarTruth, SurrBasedLocalMinimizer::softConvCount, SurrBasedLocalMinimizer::trRatioContractValue, SurrBasedLocalMinimizer::trRatioExpandValue, SurrBasedLocalMinimizer::trustRegionFactor, SurrBasedMinimizer::update_augmented_lagrange_multipliers(), SurrBasedMinimizer::update_filter(), and SurrBasedLocalMinimizer::update_penalty().

Referenced by SurrBasedLocalMinimizer::minimize_surrogates().

43.133.2.4 void update_penalty (const RealVector & *fns_center_truth*, const RealVector & *fns_star_truth*) `[private]`

initialize and update the penaltyParameter Scaling of the penalty value is important to avoid rejecting SBLM iterates which must increase the objective to achieve a reduction in constraint violation. In the basic penalty case, the penalty is ramped exponentially based on the iteration counter. In the adaptive case, the ratio of relative change between center and star points for the objective and constraint violation values is used to rescale penalty values.

References SurrBasedMinimizer::alphaEta, SurrBasedLocalMinimizer::approxSubProbObj, SurrBasedMinimizer::constraint_violation(), Minimizer::constraintTol, SurrBasedMinimizer::eta, SurrBasedMinimizer::etaSequence, Iterator::iteratedModel, SurrBasedLocalMinimizer::meritFnType, Minimizer::objective(), SurrBasedLocalMinimizer::penaltyIterOffset, SurrBasedMinimizer::penaltyParameter, Model::primary_response_fn_weights(), and SurrBasedMinimizer::sbIterNum.

Referenced by SurrBasedLocalMinimizer::tr_ratio_check().

43.133.2.5 void approx_subprob_objective_eval (const Variables & *surrogate_vars*, const Variables & *recast_vars*, const Response & *surrogate_response*, Response & *recast_response*) [static, private]

static function used to define the approximate subproblem objective. Objective functions evaluator for solution of approximate subproblem using a RecastModel.

References Response::active_set_request_vector(), SurrBasedLocalMinimizer::approxSubProbCon, SurrBasedLocalMinimizer::approxSubProbModel, SurrBasedLocalMinimizer::approxSubProbObj, SurrBasedMinimizer::augmented_lagrangian_gradient(), SurrBasedMinimizer::augmented_lagrangian_merit(), Response::function_gradient(), Response::function_gradient_view(), Response::function_gradients(), Response::function_value(), Response::function_values(), Iterator::iteratedModel, SurrBasedMinimizer::lagrangian_gradient(), SurrBasedMinimizer::lagrangian_merit(), Model::nonlinear_eq_constraint_targets(), Model::nonlinear_ineq_constraint_lower_bounds(), Model::nonlinear_ineq_constraint_upper_bounds(), Minimizer::numUserPrimaryFns, Minimizer::objective(), Minimizer::objective_gradient(), SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, Model::primary_response_fn_weights(), and SurrBasedLocalMinimizer::sblmInstance.

Referenced by SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.133.2.6 void approx_subprob_constraint_eval (const Variables & *surrogate_vars*, const Variables & *recast_vars*, const Response & *surrogate_response*, Response & *recast_response*) [static, private]

static function used to define the approximate subproblem constraints. Constraint functions evaluator for solution of approximate subproblem using a RecastModel.

References Response::active_set_derivative_vector(), Response::active_set_request_vector(), SurrBasedLocalMinimizer::approxSubProbCon, SurrBasedLocalMinimizer::approxSubProbObj, Variables::continuous_variables(), Response::function_gradient(), Response::function_gradient_view(), Response::function_gradients(), Response::function_value(), Response::function_values(), Minimizer::numUserPrimaryFns, SurrBasedLocalMinimizer::responseCenterApprox, SurrBasedLocalMinimizer::sblmInstance, and SurrBasedLocalMinimizer::varsCenter.

Referenced by SurrBasedLocalMinimizer::SurrBasedLocalMinimizer().

43.133.2.7 void hom_objective_eval (int & *mode*, int & *n*, double * *tau_and_x*, double & *f*, double * *grad_f*, int &) [static, private]

static function used by NPSOL as the objective function in the homotopy constraint relaxation formulation. NPSOL objective functions evaluator for solution of homotopy constraint relaxation parameter optimization. This constrained optimization problem performs the update of the tau parameter in the homotopy heuristic approach used to relax the constraints in the original problem .

Referenced by SurrBasedLocalMinimizer::relax_constraints().

43.133.2.8 void hom_constraint_eval (int & *mode*, int & *ncnln*, int & *n*, int & *nrowj*, int * *needc*, double * *tau_and_x*, double * *c*, double * *cjac*, int & *nstate*) `[static, private]`

static function used by NPSOL as the constraint function in the homotopy constraint relaxation formulation. NPSOL constraint functions evaluator for solution of homotopy constraint relaxation parameter optimization. This constrained optimization problem performs the update of the tau parameter in the homotopy heuristic approach used to relax the constraints in the original problem.

References Response::active_set(), SurrBasedLocalMinimizer::approxSubProbModel, Model::compute_response(), Model::continuous_variables(), Model::current_response(), Response::function_gradients(), Response::function_values(), SurrBasedLocalMinimizer::nonlinEqTargetsSlack, SurrBasedLocalMinimizer::nonlinIneqLowerBndsSlack, SurrBasedLocalMinimizer::nonlinIneqUpperBndsSlack, Model::num_functions(), Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, ActiveSet::request_vector(), SurrBasedLocalMinimizer::sblmInstance, and SurrBasedLocalMinimizer::tau.

Referenced by SurrBasedLocalMinimizer::relax_constraints().

The documentation for this class was generated from the following files:

- SurrBasedLocalMinimizer.H
- SurrBasedLocalMinimizer.C

43.134 SurrBasedMinimizer Class Reference

Base class for local/global surrogate-based optimization/least squares. Inheritance diagram for SurrBasedMinimizer::

Protected Member Functions

- SurrBasedMinimizer (Model &model)

  constructor

- ~SurrBasedMinimizer ()

  destructor

- void initialize_graphics (bool graph_2d, bool tabular_data, const String &tabular_file)

  initialize graphics customized for surrogate-based iteration

- void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- void print_results (std::ostream &s)
- virtual void minimize_surrogates ()=0

  Used for computing the optimal solution using a surrogate-based approach. Redefines the Iterator::run() virtual function.

- void update_lagrange_multipliers (const RealVector &fn_vals, const RealMatrix &fn_grads)

  initialize and update Lagrange multipliers for basic Lagrangian

- void update_augmented_lagrange_multipliers (const RealVector &fn_vals)

  initialize and update the Lagrange multipliers for augmented Lagrangian

- bool update_filter (const RealVector &fn_vals)

  update a filter from a set of function values

- Real lagrangian_merit (const RealVector &fn_vals, const RealVector &primary_wts, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_tgts)

compute a Lagrangian function from a set of function values

- void lagrangian_gradient (const RealVector &fn_vals, const RealMatrix &fn_grads, const RealVector &primary_wts, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_tgts, RealVector &lag_grad)

  compute the gradient of the Lagrangian function

- Real augmented_lagrangian_merit (const RealVector &fn_vals, const RealVector &primary_wts, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_tgts)

  compute an augmented Lagrangian function from a set of function values

- void augmented_lagrangian_gradient (const RealVector &fn_vals, const RealMatrix &fn_grads, const RealVector &primary_wts, const RealVector &nln_ineq_l_bnds, const RealVector &nln_ineq_u_bnds, const RealVector &nln_eq_tgts, RealVector &alag_grad)

  compute the gradient of the augmented Lagrangian function

- Real penalty_merit (const RealVector &fn_vals, const RealVector &primary_wts)

  compute a penalty function from a set of function values

- void penalty_gradient (const RealVector &fn_vals, const RealMatrix &fn_grads, const RealVector &primary_wts, RealVector &pen_grad)

  compute the gradient of the penalty function

- Real constraint_violation (const RealVector &fn_vals, const Real &constraint_tol)

  compute the constraint violation from a set of function values

Protected Attributes

- Iterator approxSubProbMinimizer

  the minimizer used on the surrogate model to solve the approximate subproblem on each surrogate-based iteration

- int sbIterNum

  surrogate-based minimization iteration number

- RealVectorArray sbFilter

  Set of response function vectors defining a filter (objective vs. constraint violation) for iterate selection/rejection.

- RealVector lagrangeMult

  Lagrange multipliers for basic Lagrangian calculations.

- RealVector augLagrangeMult

  Lagrange multipliers for augmented Lagrangian calculations.

- Real penaltyParameter

  the penalization factor for violated constraints used in quadratic penalty calculations; increased in update_penalty()

- RealVector origNonlinIneqLowerBnds

  original nonlinear inequality constraint lower bounds (no relaxation)

- RealVector origNonlinIneqUpperBnds

  original nonlinear inequality constraint upper bounds (no relaxation)

- RealVector origNonlinEqTargets

  original nonlinear equality constraint targets (no relaxation)

- Real eta

  constant used in etaSequence updates

- Real alphaEta

  power for etaSequence updates when updating penalty

- Real betaEta

  power for etaSequence updates when updating multipliers

- Real etaSequence

  decreasing sequence of allowable constraint violation used in augmented Lagrangian updates (refer to Conn, Gould, and Toint, section 14.4)

43.134.1 Detailed Description

Base class for local/global surrogate-based optimization/least squares. These minimizers use a SurrogateModel to perform optimization based either on local trust region methods or global updating methods.

43.134.2 Member Function Documentation

43.134.2.1 void run () `[inline, protected, virtual]`

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Iterator.

References SurrBasedMinimizer::minimize_surrogates().

43.134.2.2 void print_results (std::ostream & *s*) `[protected, virtual]`

Redefines default iterator results printing to include optimization results (objective functions and constraints).

Reimplemented from Iterator.

References Dakota::abort_handler(), Iterator::activeSet, String::begins(), Iterator::bestResponseArray, Iterator::bestVariablesArray, Dakota::data_pairs, Model::interface_id(), Iterator::iteratedModel, Dakota::lookup_by_val(), Iterator::methodName, Iterator::numFunctions, Minimizer::numUserPrimaryFns, Minimizer::optimizationFlag, ActiveSet::request_values(), Model::truth_model(), and Dakota::write_data_partial().

43.134.2.3 void update_lagrange_multipliers (const RealVector & *fn_vals*, const RealMatrix & *fn_grads*) `[protected]`

initialize and update Lagrange multipliers for basic Lagrangian For the Rockafellar augmented Lagrangian, simple Lagrange multiplier updates are available which do not require the active constraint gradients. For the basic Lagrangian, Lagrange multipliers are estimated through solution of a nonnegative linear least squares problem.

References Dakota::abort_handler(), Minimizer::bigRealBoundSize, Minimizer::constraintTol, Iterator::iteratedModel, SurrBasedMinimizer::lagrangeMult, Iterator::numContinuousVars, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, Minimizer::objective_gradient(), SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, and Model::primary_response_fn_weights().

Referenced by SurrBasedLocalMinimizer::hard_convergence_check().

43.134.2.4 void update_augmented_lagrange_multipliers (const RealVector & *fn_vals*) `[protected]`

initialize and update the Lagrange multipliers for augmented Lagrangian For the Rockafellar augmented Lagrangian, simple Lagrange multiplier updates are available which do not require the active constraint gradients. For the basic Lagrangian, Lagrange multipliers are estimated through solution of a nonnegative linear least squares problem.

References SurrBasedMinimizer::augLagrangeMult, SurrBasedMinimizer::betaEta, Minimizer::bigRealBoundSize, SurrBasedMinimizer::etaSequence, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, SurrBasedMinimizer::origNonlinIneqUpperBnds, and SurrBasedMinimizer::penaltyParameter.

Referenced by SurrBasedLocalMinimizer::hard_convergence_check(), EffGlobalMinimizer::minimize_surrogates_on_model(), and SurrBasedLocalMinimizer::tr_ratio_check().

43.134.2.5 bool update_filter (const RealVector & *fn_vals*) `[protected]`

update a filter from a set of function values Update the sbFilter with fn_vals if new iterate is non-dominated.

References SurrBasedMinimizer::constraint_violation(), Iterator::iteratedModel, Minimizer::numNonlinearConstraints, Minimizer::objective(), Model::primary_response_fn_weights(), and SurrBasedMinimizer::sbFilter.

Referenced by SurrBasedLocalMinimizer::hard_convergence_check(), and SurrBasedLocalMinimizer::tr_ratio_check().

43.134.2.6 Real lagrangian_merit (const RealVector & *fn_vals*, const RealVector & *primary_wts*, const RealVector & *nln_ineq_l_bnds*, const RealVector & *nln_ineq_u_bnds*, const RealVector & *nln_eq_tgts*) [protected]

compute a Lagrangian function from a set of function values The Lagrangian function computation sums the objective function and the Lagrange multipler terms for inequality/equality constraints. This implementation follows the convention in Vanderplaats with g<=0 and h=0. The bounds/targets passed in may reflect the original constraints or the relaxed constraints.

References Minimizer::bigRealBoundSize, Minimizer::constraintTol, SurrBasedMinimizer::lagrangeMult, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, and Minimizer::objective().

Referenced by SurrBasedLocalMinimizer::approx_subprob_objective_eval(), and SurrBasedLocalMinimizer::tr_ratio_check().

43.134.2.7 Real augmented_lagrangian_merit (const RealVector & *fn_vals*, const RealVector & *primary_wts*, const RealVector & *nln_ineq_l_bnds*, const RealVector & *nln_ineq_u_bnds*, const RealVector & *nln_eq_tgts*) [protected]

compute an augmented Lagrangian function from a set of function values The Rockafellar augmented Lagrangian function sums the objective function, Lagrange multipler terms for inequality/equality constraints, and quadratic penalty terms for inequality/equality constraints. This implementation follows the convention in Vanderplaats with g<=0 and h=0. The bounds/targets passed in may reflect the original constraints or the relaxed constraints.

References SurrBasedMinimizer::augLagrangeMult, Minimizer::bigRealBoundSize, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, Minimizer::objective(), and SurrBasedMinimizer::penaltyParameter.

Referenced by SurrBasedLocalMinimizer::approx_subprob_objective_eval(), EffGlobalMinimizer::get_best_sample(), EffGlobalMinimizer::minimize_surrogates_on_model(), and SurrBasedLocalMinimizer::tr_ratio_check().

43.134.2.8 Real penalty_merit (const RealVector & *fn_vals*, const RealVector & *primary_wts*) [protected]

compute a penalty function from a set of function values The penalty function computation applies a quadratic penalty to any constraint violations and adds this to the objective function(s) p = f + r_p cv.

References SurrBasedMinimizer::constraint_violation(), Minimizer::constraintTol, Minimizer::objective(), and SurrBasedMinimizer::penaltyParameter.

Referenced by SurrBasedLocalMinimizer::tr_ratio_check().

43.134.2.9 Real constraint_violation (const RealVector & *fn_vals*, const Real & *constraint_tol*) [protected]

compute the constraint violation from a set of function values Compute the quadratic constraint violation defined as cv = g+^T g+ + h+^T h+. This implementation supports equality constraints and 2-sided inequalities. The constraint_tol allows for a small constraint infeasibility (used for penalty methods, but not Lagrangian methods).

References Minimizer::bigRealBoundSize, Minimizer::numNonlinearEqConstraints, Minimizer::numNonlinearIneqConstraints, Minimizer::numUserPrimaryFns, SurrBasedMinimizer::origNonlinEqTargets, SurrBasedMinimizer::origNonlinIneqLowerBnds, and SurrBasedMinimizer::origNonlinIneqUpperBnds.

Referenced by SurrBasedLocalMinimizer::hard_convergence_check(), EffGlobalMinimizer::minimize_surrogates_on_model(), SurrBasedMinimizer::penalty_merit(), SurrBasedLocalMinimizer::relax_constraints(), SurrBasedLocalMinimizer::tr_ratio_check(), SurrBasedMinimizer::update_filter(), and SurrBasedLocalMinimizer::update_penalty().

The documentation for this class was generated from the following files:

- SurrBasedMinimizer.H
- SurrBasedMinimizer.C

43.135 SurrogateModel Class Reference

Base class for surrogate models (DataFitSurrModel and HierarchSurrModel). Inheritance diagram for SurrogateModel::

Protected Member Functions

- SurrogateModel (ProblemDescDB &problem_db)

  constructor

- SurrogateModel (ParallelLibrary &parallel_lib, const SharedVariablesData &svd, const ActiveSet &set, short output_level)

  alternate constructor

- ~SurrogateModel ()

  destructor

- Model & subordinate_model ()

  return truth_model()

- short surrogate_response_mode ()

  return responseMode

- DiscrepancyCorrection & discrepancy_correction ()

  return deltaCorr

- void check_submodel_compatibility (const Model &sub_model)

  verify compatibility between SurrogateModel attributes and attributes of the submodel (DataFitSurrModel::actualModel or HierarchSurrModel::highFidelityModel)

- bool force_rebuild ()

  evaluate whether a rebuild of the approximation should be forced based on changes in the inactive data

- void asv_mapping (const ShortArray &orig_asv, ShortArray &actual_asv, ShortArray &approx_asv, bool build_flag)

  distributes the incoming orig_asv among actual_asv and approx_asv

- void asv_mapping (const ShortArray &actual_asv, const ShortArray &approx_asv, ShortArray &combined_asv)

  reconstitutes a combined_asv from actual_asv and approx_asv

- void response_mapping (const Response &actual_response, const Response &approx_response, Response &combined_response)

  overlays actual_response and approx_response to update combined_response

Protected Attributes

- IntSet surrogateFnIndices

  for mixed response sets, this array specifies the response function subset that is approximated

- IntResponseMap surrResponseMap

  map of surrogate responses used in derived_synchronize() and derived_synchronize_nowait() functions

- IntRealVectorMap rawCVarsMap

  map of raw continuous variables used by apply_correction(). Model::varsList cannot be used for this purpose since it does not contain lower level variables sets from finite differencing.

- IntIntMap truthIdMap

  map from actualModel/highFidelityModel evaluation ids to DataFitSurrModel/HierarchSurrModel ids

- IntIntMap surrIdMap

  map from approxInterface/lowFidelityModel evaluation ids to DataFitSurrModel/HierarchSurrModel ids

- IntResponseMap cachedApproxRespMap

  map of approximate responses retrieved in derived_synchronize_nowait() that could not be returned since corresponding truth model response portions were still pending.

- short responseMode

  an enumeration that controls the response calculation mode in {DataFit,Hierarch}SurrModel approximate response computations

- size_t approxBuilds

  number of calls to build_approximation()

- RealVector referenceCLBnds

  stores a reference copy of active continuous lower bounds when the approximation is built; used to detect when a rebuild is required.

- RealVector referenceCUBnds

  stores a reference copy of active continuous upper bounds when the approximation is built; used to detect when a rebuild is required.

- IntVector referenceDILBnds

  stores a reference copy of active discrete int lower bounds when the approximation is built; used to detect when a rebuild is required.

- IntVector referenceDIUBnds

  stores a reference copy of active discrete int upper bounds when the approximation is built; used to detect when a rebuild is required.

- RealVector referenceDRLBnds

  stores a reference copy of active discrete real lower bounds when the approximation is built; used to detect when a rebuild is required.

- RealVector referenceDRUBnds

  stores a reference copy of active discrete real upper bounds when the approximation is built; used to detect when a rebuild is required.

- RealVector referenceICVars

  stores a reference copy of the inactive continuous variables when the approximation is built using a Distinct view; used to detect when a rebuild is required.

- IntVector referenceIDIVars

  stores a reference copy of the inactive discrete int variables when the approximation is built using a Distinct view; used to detect when a rebuild is required.

- RealVector referenceIDRVars

  stores a reference copy of the inactive discrete real variables when the approximation is built using a Distinct view; used to detect when a rebuild is required.

- DiscrepancyCorrection deltaCorr

  manages construction and application of correction functions that are applied to a surrogate model (DataFitSurr or HierarchSurr) in order to reproduce high fidelity data.

Private Attributes

- Variables truthModelVars

  copy of the truth model variables object used to simplify conversion among differing variable views in force_rebuild()

- Constraints truthModelCons

  copy of the truth model constraints object used to simplify conversion among differing variable views in force_rebuild()

43.135.1 Detailed Description

Base class for surrogate models (DataFitSurrModel and HierarchSurrModel). The SurrogateModel class provides common functions to derived classes for computing and applying corrections to approximations.

43.135.2 Member Function Documentation

43.135.2.1 bool force_rebuild () [protected, virtual]

evaluate whether a rebuild of the approximation should be forced based on changes in the inactive data This function forces a rebuild of the approximation according to the sub-model variables view, the approximation type, and whether the active approximation bounds or inactive variable values have changed since the last approximation build.

Reimplemented from Model.

References Constraints::all_continuous_lower_bounds(), Constraints::all_continuous_upper_bounds(), Variables::all_continuous_variables(), Constraints::all_discrete_int_lower_bounds(), Constraints::all_discrete_int_upper_bounds(), Variables::all_discrete_int_variables(), Constraints::all_discrete_real_lower_bounds(), Constraints::all_discrete_real_upper_bounds(), Variables::all_discrete_real_variables(), String::begins(), Model::continuous_lower_bounds(), Constraints::continuous_lower_bounds(), Model::continuous_upper_bounds(), Constraints::continuous_upper_bounds(), Variables::continuous_variables(), Constraints::copy(), Variables::copy(), Model::current_variables(), Model::currentVariables, Model::discrete_int_lower_bounds(), Constraints::discrete_int_lower_bounds(), Model::discrete_int_upper_bounds(), Constraints::discrete_int_upper_bounds(), Variables::discrete_int_variables(), Model::discrete_real_lower_bounds(), Constraints::discrete_real_lower_bounds(), Model::discrete_real_upper_bounds(), Constraints::discrete_real_upper_bounds(), Variables::discrete_real_variables(), Variables::inactive_continuous_variables(), Variables::inactive_discrete_int_variables(), Variables::inactive_discrete_real_variables(), Constraints::is_null(), Variables::is_null(), Model::is_null(), Model::model_type(), SurrogateModel::referenceCLBnds, SurrogateModel::referenceCUBnds, SurrogateModel::referenceDILBnds, SurrogateModel::referenceDIUBnds, SurrogateModel::referenceDRLBnds, SurrogateModel::referenceDRUBnds, SurrogateModel::referenceICVars, SurrogateModel::referenceIDIVars, SurrogateModel::referenceIDRVars, Model::subordinate_model(), Model::surrogateType, Model::truth_model(), SurrogateModel::truthModelCons, SurrogateModel::truthModelVars, Model::user_defined_constraints(), Model::userDefinedConstraints, and Variables::view().

Referenced by HierarchSurrModel::derived_asynch_compute_response(), DataFitSurrModel::derived_asynch_compute_response(), HierarchSurrModel::derived_compute_response(), and DataFitSurrModel::derived_compute_response().

43.135.3 Member Data Documentation

43.135.3.1 short responseMode [protected]

an enumeration that controls the response calculation mode in {DataFit,Hierarch}SurrModel approximate response computations SurrBasedLocalMinimizer toggles this mode since compute_correction() does not back out old corrections.

Referenced by HierarchSurrModel::derived_asynch_compute_response(), DataFitSurrModel::derived_asynch_compute_response(), HierarchSurrModel::derived_compute_response(), DataFitSurrModel::derived_compute_response(), HierarchSurrModel::derived_synchronize(), DataFitSurrModel::derived_synchronize(), DataFitSurrModel::derived_synchronize_approx(), HierarchSurrModel::derived_synchronize_nowait(), DataFitSurrModel::derived_synchronize_nowait(), SurrogateModel::surrogate_response_mode(), HierarchSurrModel::surrogate_response_mode(), and DataFitSurrModel::surrogate_response_mode().

43.135.3.2 size_t approxBuilds [protected]

number of calls to build_approximation() used as a flag to automatically build the approximation if one of the derived compute_response functions is called prior to build_approximation().

Referenced by DataFitSurrModel::append_approximation(), DataFitSurrModel::approximation_coefficients(), HierarchSurrModel::build_approximation(), DataFitSurrModel::build_approximation(), HierarchSurrModel::derived_asynch_compute_response(), DataFitSurrModel::derived_asynch_compute_response(), HierarchSurrModel::derived_compute_response(), DataFitSurrModel::derived_compute_response(), DataFitSurrModel::update_actual_model(), DataFitSurrModel::update_approximation(), DataFitSurrModel::update_from_actual_model(), and HierarchSurrModel::update_model().

The documentation for this class was generated from the following files:

- SurrogateModel.H
- SurrogateModel.C

43.136 SysCallAnalysisCode Class Reference

Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls. Inheritance diagram for SysCallAnalysisCode::

Public Member Functions

- SysCallAnalysisCode (const ProblemDescDB &problem_db)

  constructor

- ~SysCallAnalysisCode ()

  destructor

- void spawn_evaluation (const bool block_flag)

  spawn a complete function evaluation

- void spawn_input_filter (const bool block_flag)

  spawn the input filter portion of a function evaluation

- void spawn_analysis (const int &analysis_id, const bool block_flag)

  spawn a single analysis as part of a function evaluation

- void spawn_output_filter (const bool block_flag)

  spawn the output filter portion of a function evaluation

43.136.1 Detailed Description

Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls. SysCallAnalysisCode creates separate simulation processes using the C system() command. It utilizes CommandShell to manage shell syntax and asynchronous invocations.

43.136.2 Member Function Documentation

43.136.2.1 void spawn_evaluation (const bool *block_flag*)

spawn a complete function evaluation Put the SysCallAnalysisCode to the shell. This function is used when all portions of the function evaluation (i.e., all analysis drivers) are executed on the local processor.

References CommandShell::asynch_flag(), AnalysisCode::commandLineArgs, AnalysisCode::curWorkdir, Dakota::flush(), AnalysisCode::iFilterName, AnalysisCode::multipleParamsFiles, AnalysisCode::numPrograms, AnalysisCode::oFilterName, AnalysisCode::paramsFileName, AnalysisCode::programNames, AnalysisCode::resultsFileName, CommandShell::suppress_output_flag(), AnalysisCode::suppressOutputFlag, and AnalysisCode::useWorkdir.

Referenced by Dakota::perform_analysis(), and SysCallApplicInterface::spawn_application().

43.136.2.2 void spawn_input_filter (const bool *block_flag*)

spawn the input filter portion of a function evaluation Put the input filter to the shell. This function is used when multiple analysis drivers are spread between processors. No need to check for a Null input filter, as this is checked externally. Use of nonblocking shells is supported in this fn, although its use is currently prevented externally.

References CommandShell::asynch_flag(), AnalysisCode::commandLineArgs, AnalysisCode::curWorkdir, Dakota::flush(), AnalysisCode::iFilterName, AnalysisCode::paramsFileName, AnalysisCode::resultsFileName, CommandShell::suppress_output_flag(), AnalysisCode::suppressOutputFlag, and AnalysisCode::useWorkdir.

Referenced by SysCallApplicInterface::spawn_application().

43.136.2.3 void spawn_analysis (const int & *analysis_id*, const bool *block_flag*)

spawn a single analysis as part of a function evaluation Put a single analysis to the shell. This function is used when multiple analysis drivers are spread between processors. Use of nonblocking shells is supported in this fn, although its use is currently prevented externally.

References CommandShell::asynch_flag(), AnalysisCode::commandLineArgs, AnalysisCode::curWorkdir, Dakota::flush(), AnalysisCode::multipleParamsFiles, AnalysisCode::numPrograms, AnalysisCode::paramsFileName, AnalysisCode::programNames, AnalysisCode::resultsFileName, CommandShell::suppress_output_flag(), AnalysisCode::suppressOutputFlag, and AnalysisCode::useWorkdir.

Referenced by SysCallApplicInterface::derived_synchronous_local_analysis(), GridApplicInterface::derived_synchronous_local_analysis(), and SysCallApplicInterface::spawn_application().

43.136.2.4 void spawn_output_filter (const bool *block_flag*)

spawn the output filter portion of a function evaluation Put the output filter to the shell. This function is used when multiple analysis drivers are spread between processors. No need to check for a Null output filter, as this is checked externally. Use of nonblocking shells is supported in this fn, although its use is currently prevented externally.

References CommandShell::asynch_flag(), AnalysisCode::commandLineArgs, AnalysisCode::curWorkdir, Dakota::flush(), AnalysisCode::oFilterName, AnalysisCode::paramsFileName, AnalysisCode::resultsFileName, CommandShell::suppress_output_flag(), AnalysisCode::suppressOutputFlag, and AnalysisCode::useWorkdir.

Referenced by SysCallApplicInterface::spawn_application().

The documentation for this class was generated from the following files:

- SysCallAnalysisCode.H
- SysCallAnalysisCode.C

43.137 SysCallApplicInterface Class Reference

Derived application interface class which spawns simulation codes using system calls. Inheritance diagram for SysCallApplicInterface::

Public Member Functions

- SysCallApplicInterface (const ProblemDescDB &problem_db)

  constructor

- ~SysCallApplicInterface ()

  destructor

- void derived_map (const Variables &vars, const ActiveSet &set, Response &response, int fn_eval_id)

  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- void derived_map_asynch (const ParamResponsePair &pair)

  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- void derived_synch (PRPQueue &prp_queue)
- void derived_synch_nowait (PRPQueue &prp_queue)
- int derived_synchronous_local_analysis (const int &analysis_id)
- const std::vector< String > & analysis_drivers () const

  retrieve the analysis drivers specification for application interfaces

- const AnalysisCode * analysis_code () const

  return AnalysisCode::fileNameMap when defined for derived Interface class

Private Member Functions

- void spawn_application (const bool block_flag)

  Spawn the application by managing the input filter, analysis drivers, and output filter. Called from derived_map() & derived_map_asynch().

- void derived_synch_kernel (PRPQueue &prp_queue)

  Convenience function for common code between derived_synch() & derived_synch_nowait().

- bool system_call_file_test (const std::string &root_file)

  detect completion of a function evaluation through existence of the necessary results file(s)

Private Attributes

- SysCallAnalysisCode sysCallSimulator

  SysCallAnalysisCode provides convenience functions for passing the input filter, the analysis drivers, and the output filter to a CommandShell in various combinations.

- IntSet sysCallSet

  set of function evaluation id's for active asynchronous system call evaluations

- IntShortMap failCountMap

  map linking function evaluation id's to number of response read failures

43.137.1 Detailed Description

Derived application interface class which spawns simulation codes using system calls. SysCallApplicInterface uses a SysCallAnalysisCode object for performing simulation invocations.

43.137.2 Member Function Documentation

43.137.2.1 void derived_synch (PRPQueue & *prp_queue*) `[inline, virtual]`

Check for completion of active asynch jobs (tracked with sysCallSet). Wait for at least one completion and complete all jobs that have returned. This satisifies a "fairness" principle, in the sense that a completed job will _always_ be processed (whereas accepting only a single completion could always accept the same completion - the case of very inexpensive fn. evals. - and starve some servers).

Reimplemented from ApplicationInterface.

References ApplicationInterface::completionSet, and SysCallApplicInterface::derived_synch_kernel().

43.137.2.2 void derived_synch_nowait (PRPQueue & *prp_queue*) `[inline, virtual]`

Check for completion of active asynch jobs (tracked with sysCallSet). Make one pass through sysCallSet & complete all jobs that have returned.

Reimplemented from ApplicationInterface.

References SysCallApplicInterface::derived_synch_kernel().

43.137.2.3 int derived_synchronous_local_analysis (const int & *analysis_id*) `[inline, virtual]`

This code provides the derived function used by ApplicationInterface::serve_analyses_synch().

Reimplemented from ApplicationInterface.

References SysCallAnalysisCode::spawn_analysis(), and SysCallApplicInterface::sysCallSimulator.

The documentation for this class was generated from the following files:

- SysCallApplicInterface.H
- SysCallApplicInterface.C

43.138 TANA3Approximation Class Reference

Derived approximation class for TANA-3 two-point exponential approximation (a multipoint approximation). Inheritance diagram for TANA3Approximation::

Public Member Functions

- TANA3Approximation ()

  default constructor

- TANA3Approximation (ProblemDescDB &problem_db, size_t num_vars)

  standard constructor

- TANA3Approximation (size_t num_vars, short data_order)

  alternate constructor

- ~TANA3Approximation ()

  destructor

Protected Member Functions

- int min_coefficients () const

  return the minimum number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- int num_constraints () const

  return the number of constraints to be enforced via an anchor point

- void build ()

  builds the approximation from scratch

- Real get_value (const RealVector &x)

  retrieve the approximate function value for a given parameter vector

- const RealVector & get_gradient (const RealVector &x)

  retrieve the approximate function gradient for a given parameter vector

- void clear_current ()

Private Member Functions

- void find_scaled_coefficients ()

  compute TANA coefficients based on scaled inputs

- void offset (const RealVector &x, RealVector &s)

  based on minX, apply offset scaling to x to define s

Private Attributes

- RealVector pExp

  vector of exponent values

- RealVector minX

  vector of minimum parameter values used in scaling

- RealVector scX1

  vector of scaled x1 values

- RealVector scX2

  vector of scaled x2 values

- Real H

  the scalar Hessian value in the TANA-3 approximation

43.138.1 Detailed Description

Derived approximation class for TANA-3 two-point exponential approximation (a multipoint approximation). The TANA3Approximation class provides a multipoint approximation based on matching value and gradient data from two points (typically the current and previous iterates) in parameter space. It forms an exponential approximation in terms of intervening variables.

43.138.2 Member Function Documentation

43.138.2.1 void build () [protected, virtual]

builds the approximation from scratch This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References Dakota::abort_handler(), Approximation::approxData, Approximation::buildDataOrder, TANA3Approximation::find_scaled_coefficients(), TANA3Approximation::minX, Approximation::numVars, and TANA3Approximation::pExp.

43.138.2.2 void clear_current () [inline, protected, virtual]

Redefine default implementation to support history mechanism.

Reimplemented from Approximation.

References Approximation::approxData.

The documentation for this class was generated from the following files:

- TANA3Approximation.H
- TANA3Approximation.C

43.139 TaylorApproximation Class Reference

Derived approximation class for first- or second-order Taylor series (a local approximation). Inheritance diagram for TaylorApproximation::

Public Member Functions

- TaylorApproximation ()

  default constructor

- TaylorApproximation (ProblemDescDB &problem_db, size_t num_vars)

  standard constructor

- TaylorApproximation (size_t num_vars, short data_order)

  alternate constructor

- ~TaylorApproximation ()

  destructor

Protected Member Functions

- int min_coefficients () const

  return the minimum number of samples (unknowns) required to build the derived class approximation type in numVars dimensions

- void build ()

  builds the approximation from scratch

- Real get_value (const RealVector &x)

  retrieve the approximate function value for a given parameter vector

- const RealVector & get_gradient (const RealVector &x)

  retrieve the approximate function gradient for a given parameter vector

- const RealSymMatrix & get_hessian (const RealVector &x)

  retrieve the approximate function Hessian for a given parameter vector

43.139.1 Detailed Description

Derived approximation class for first- or second-order Taylor series (a local approximation). The TaylorApproximation class provides a local approximation based on data from a single point in parameter space. It uses a zeroth-, first- or second-order Taylor series expansion: f(x) = f(x_c) for zeroth-order, plus grad(x_c)' (x - x_c) for first- and second-order, and plus (x - x_c)' Hess(x_c) (x - x_c) / 2 for second-order.

43.139.2 Member Function Documentation

43.139.2.1 void build () [protected, virtual]

builds the approximation from scratch This is the common base class portion of the virtual fn and is insufficient on its own; derived implementations should explicitly invoke (or reimplement) this base class contribution.

Reimplemented from Approximation.

References Dakota::abort_handler(), Approximation::approxData, Approximation::buildDataOrder, and Approximation::numVars.

The documentation for this class was generated from the following files:

- TaylorApproximation.H
- TaylorApproximation.C

43.140 TrackerHTTP Class Reference

TrackerHTTP: a usage tracking module that uses HTTP/HTTPS via the curl library.

Public Member Functions

- TrackerHTTP ()

  default constructor is allowed, but doesn't track methods used and outputs on all ranks

- TrackerHTTP (ProblemDescDB &problem_db, int world_rank=0)

  standard constructor with ProblemDescDB, rank

- ~TrackerHTTP ()

  destructor to free handles

- void post_start ()

  post the start of an analysis and archive start time

- void post_finish (unsigned runtime=0)

  post the completion of an analysis including elapsed time

Private Member Functions

- void initialize (int world_rank=0)

  shared initialization functions across constructors

- void url_add_field (std::string &url, const char *keyword, const std::string &value, bool delimit=true) const

  append keyword/value pair to url in GET style (with &keyword=value); set delimit = false to omit the &

- void build_default_data (std::string &url, std::time_t &rawtime, const std::string &mode) const

  construct URL with shared information for start/finish

- void send_data_using_get (const std::string &urltopost) const

  transmit data to the web server using GET

- void send_data_using_post (const std::string &datatopost) const

  POST separate location and query; datatopost="name=daniel&project=curl".

- void populate_method_list (ProblemDescDB &problem_db)

  extract list of methods from problem database

- std::string get_uid () const

  get the real user ID

- std::string get_username () const

  get the username as reported by the environment

- std::string get_hostname () const

  get the system hostname

- std::string get_os () const

  get the operating system

- std::string get_datetime (const std::time_t &rawtime) const

  get the date and time as a string YYYYMMDDHHMMSS

Private Attributes

- CURL * curlPtr

  pointer to the curl handler instance

- FILE * devNull

  pointer to /dev/null

- std::string trackerLocation

  base URL for the tracker

- std::string proxyLocation

  if empty, proxy may still be specified via environment variables (unlike default CURL behavior)

- long timeoutSeconds

  seconds until the request will timeout (may have issues with signals)

- std::string methodList

  list of active methods

- std::string dakotaVersion

  DAKOTA version.

- std::time_t startTime

  cached starting time in raw seconds

- short outputLevel

  verbosity control

43.140.1 Detailed Description

TrackerHTTP: a usage tracking module that uses HTTP/HTTPS via the curl library.

43.140.2 Member Function Documentation

43.140.2.1 void send_data_using_get (const std::string & *urltopost*) const [private]

transmit data to the web server using GET whole url including location&fields

References TrackerHTTP::curlPtr, and TrackerHTTP::outputLevel.

43.140.2.2 void send_data_using_post (const std::string & *datatopost*) const [private]

POST separate location and query; datatopost="name=daniel&project=curl". separate location and query; datatopost="name=daniel&project=curl"

References TrackerHTTP::curlPtr, TrackerHTTP::outputLevel, and TrackerHTTP::trackerLocation.

Referenced by TrackerHTTP::post_finish(), and TrackerHTTP::post_start().

The documentation for this class was generated from the following files:

- TrackerHTTP.H
- TrackerHTTP.C

43.141 Variables Class Reference

Base class for the variables class hierarchy. Inheritance diagram for Variables::

Public Member Functions

- Variables ()

  default constructor

- Variables (const ProblemDescDB &problem_db)

  standard constructor

- Variables (const SharedVariablesData &svd)

  alternate constructor for instantiations on the fly

- Variables (const Variables &vars)

  copy constructor

- virtual ~Variables ()

  destructor

- Variables operator= (const Variables &vars)

  assignment operator

- virtual void reshape (const SizetArray &vc_totals)

  reshapes an existing Variables object based on the incoming variablesComponents

- virtual void read (std::istream &s)

  read a variables object from an std::istream

- virtual void write (std::ostream &s) const

  write a variables object to an std::ostream

- virtual void write_aprepro (std::ostream &s) const

  write a variables object to an std::ostream in aprepro format

- virtual void read_annotated (std::istream &s)

  read a variables object in annotated format from an istream

- virtual void write_annotated (std::ostream &s) const

  write a variables object in annotated format to an std::ostream

- virtual void read_tabular (std::istream &s)

  read a variables object in tabular format from an istream

- virtual void write_tabular (std::ostream &s) const

  write a variables object in tabular format to an std::ostream

- virtual void read (BiStream &s)

  read a variables object from the binary restart stream

- virtual void write (BoStream &s) const

  write a variables object to the binary restart stream

- virtual void read (MPIUnpackBuffer &s)

  read a variables object from a packed MPI buffer

- virtual void write (MPIPackBuffer &s) const

  write a variables object to a packed MPI buffer

- size_t tv () const

  total number of vars

- size_t cv () const

  number of active continuous vars

- size_t cv_start () const

  start index of active continuous vars

- size_t div () const

  number of active discrete int vars

- size_t div_start () const

  start index of active discrete int vars

- size_t drv () const

  number of active discrete real vars

- size_t drv_start () const

  start index of active discrete real vars

- size_t icv () const

  number of inactive continuous vars

- size_t icv_start () const

start index of inactive continuous vars

- size_t idiv () const

  number of inactive discrete int vars

- size_t idiv_start () const

  start index of inactive discrete int vars

- size_t idrv () const

  number of inactive discrete real vars

- size_t idrv_start () const

  start index of inactive discrete real vars

- size_t acv () const

  total number of continuous vars

- size_t adiv () const

  total number of discrete integer vars

- size_t adrv () const

  total number of discrete real vars

- const SharedVariablesData & shared_data () const

  return sharedVarsData

- const Real & continuous_variable (size_t index) const

  return an active continuous variable

- const RealVector & continuous_variables () const

  return the active continuous variables

- void continuous_variable (const Real &c_var, size_t index)

  set an active continuous variable

- void continuous_variables (const RealVector &c_vars)

  set the active continuous variables

- int discrete_int_variable (size_t index) const

  return an active discrete integer variable

- const IntVector & discrete_int_variables () const

  return the active discrete integer variables

- void discrete_int_variable (int di_var, size_t index)

  set an active discrete integer variable

- void discrete_int_variables (const IntVector &di_vars)
  set the active discrete integer variables
- const Real & discrete_real_variable (size_t index) const
  return an active discrete real variable
- const RealVector & discrete_real_variables () const
  return the active discrete real variables
- void discrete_real_variable (const Real &dr_var, size_t index)
  set an active discrete real variable
- void discrete_real_variables (const RealVector &dr_vars)
  set the active discrete real variables
- StringMultiArrayConstView continuous_variable_labels () const
  return the active continuous variable labels
- void continuous_variable_labels (StringMultiArrayConstView cv_labels)
  set the active continuous variable labels
- void continuous_variable_label (const String &cv_label, size_t index)
  set an active continuous variable label
- StringMultiArrayConstView discrete_int_variable_labels () const
  return the active discrete integer variable labels
- void discrete_int_variable_labels (StringMultiArrayConstView div_labels)
  set the active discrete integer variable labels
- void discrete_int_variable_label (const String &div_label, size_t index)
  set an active discrete integer variable label
- StringMultiArrayConstView discrete_real_variable_labels () const
  return the active discrete real variable labels
- void discrete_real_variable_labels (StringMultiArrayConstView drv_labels)
  set the active discrete real variable labels
- void discrete_real_variable_label (const String &drv_label, size_t index)
  set an active discrete real variable label
- UShortMultiArrayConstView continuous_variable_types () const
  return the active continuous variable types

- UShortMultiArrayConstView discrete_int_variable_types () const
  return the active discrete integer variable types
- UShortMultiArrayConstView discrete_real_variable_types () const
  return the active discrete real variable types
- SizetMultiArrayConstView continuous_variable_ids () const
  return the active continuous variable position identifiers
- const SizetArray & merged_discrete_ids () const
  returns the list of discrete variables merged into a continuous array
- const RealVector & inactive_continuous_variables () const
  return the inactive continuous variables
- void inactive_continuous_variables (const RealVector &ic_vars)
  set the inactive continuous variables
- const IntVector & inactive_discrete_int_variables () const
  return the inactive discrete variables
- void inactive_discrete_int_variables (const IntVector &idi_vars)
  set the inactive discrete variables
- const RealVector & inactive_discrete_real_variables () const
  return the inactive discrete variables
- void inactive_discrete_real_variables (const RealVector &idr_vars)
  set the inactive discrete variables
- StringMultiArrayConstView inactive_continuous_variable_labels () const
  return the inactive continuous variable labels
- void inactive_continuous_variable_labels (StringMultiArrayConstView ic_vars)
  set the inactive continuous variable labels
- StringMultiArrayConstView inactive_discrete_int_variable_labels () const
  return the inactive discrete variable labels
- void inactive_discrete_int_variable_labels (StringMultiArrayConstView idi_vars)
  set the inactive discrete variable labels
- StringMultiArrayConstView inactive_discrete_real_variable_labels () const
  return the inactive discrete variable labels
- void inactive_discrete_real_variable_labels (StringMultiArrayConstView idr_vars)

set the inactive discrete variable labels

- UShortMultiArrayConstView inactive_continuous_variable_types () const
  return the inactive continuous variable types
- UShortMultiArrayConstView inactive_discrete_int_variable_types () const
  return the inactive discrete integer variable types
- UShortMultiArrayConstView inactive_discrete_real_variable_types () const
  return the inactive discrete real variable types
- SizetMultiArrayConstView inactive_continuous_variable_ids () const
  return the inactive continuous variable position identifiers
- const RealVector & all_continuous_variables () const
  returns a single array with all continuous variables
- void all_continuous_variables (const RealVector &ac_vars)
  sets all continuous variables using a single array
- void all_continuous_variable (const Real &ac_var, size_t index)
  set a variable within the all continuous array
- const IntVector & all_discrete_int_variables () const
  returns a single array with all discrete variables
- void all_discrete_int_variables (const IntVector &adi_vars)
  sets all discrete variables using a single array
- void all_discrete_int_variable (int adi_var, size_t index)
  set a variable within the all discrete array
- const RealVector & all_discrete_real_variables () const
  returns a single array with all discrete variables
- void all_discrete_real_variables (const RealVector &adr_vars)
  sets all discrete variables using a single array
- void all_discrete_real_variable (const Real &adr_var, size_t index)
  set a variable within the all discrete array
- StringMultiArrayView all_continuous_variable_labels () const
  returns a single array with all continuous variable labels
- void all_continuous_variable_labels (StringMultiArrayConstView acv_labels)
  sets all continuous variable labels using a single array

- void all_continuous_variable_label (const String &acv_label, size_t index)
  set a label within the all continuous label array
- StringMultiArrayView all_discrete_int_variable_labels () const
  returns a single array with all discrete variable labels
- void all_discrete_int_variable_labels (StringMultiArrayConstView adiv_labels)
  sets all discrete variable labels using a single array
- void all_discrete_int_variable_label (const String &adiv_label, size_t index)
  set a label within the all discrete label array
- StringMultiArrayView all_discrete_real_variable_labels () const
  returns a single array with all discrete variable labels
- void all_discrete_real_variable_labels (StringMultiArrayConstView adrv_labels)
  sets all discrete variable labels using a single array
- void all_discrete_real_variable_label (const String &adrv_label, size_t index)
  set a label within the all discrete label array
- UShortMultiArrayConstView all_continuous_variable_types () const
  return all continuous variable types
- UShortMultiArrayConstView all_discrete_int_variable_types () const
  return all discrete variable types
- UShortMultiArrayConstView all_discrete_real_variable_types () const
  return all discrete variable types
- SizetMultiArrayConstView all_continuous_variable_ids () const
  return all continuous variable position identifiers
- Variables copy () const
  for use when a deep copy is needed (the representation is _not_ shared)
- const std::pair< short, short > & view () const
  returns variablesView
- std::pair< short, short > get_view (const ProblemDescDB &problem_db) const
  defines variablesView from problem_db attributes
- void inactive_view (short view2)
  sets the inactive view based on higher level (nested) context

- const String & variables_id () const

  returns the variables identifier string

- const SizetArray & variables_components_totals () const

  returns the number of variables for each of the constitutive components

- bool is_null () const

  function to check variablesRep (does this envelope contain a letter)

Protected Member Functions

- Variables (BaseConstructor, const ProblemDescDB &problem_db, const std::pair< short, short > &view)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- Variables (BaseConstructor, const SharedVariablesData &svd)

  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139)

- virtual void build_active_views ()

  construct active views of all variables arrays

- virtual void build_inactive_views ()

  construct inactive views of all variables arrays

- void build_views ()

  construct active/inactive views of all variables arrays

Protected Attributes

- SharedVariablesData sharedVarsData

  reference-counted instance of shared variables data: id's, labels, counts

- RealVector allContinuousVars

  array combining all of the continuous variables (design, uncertain, state)

- IntVector allDiscreteIntVars

  array combining all of the discrete integer variables (design, state)

- RealVector allDiscreteRealVars

  array combining all of the discrete real variables (design, state)

- size_t cvStart

start index of active continuous variables within allContinuousVars

- size_t divStart

  start index of active discrete integer variables within allDiscreteIntVars

- size_t drvStart

  start index of active discrete real variables within allDiscreteRealVars

- size_t icvStart

  start index of inactive continuous variables within allContinuousVars

- size_t idivStart

  start index of inactive discrete integer variables w/i allDiscreteIntVars

- size_t idrvStart

  start index of inactive discrete real variables within allDiscreteRealVars

- size_t numCV

  number of active continuous variables

- size_t numDIV

  number of active discrete integer variables

- size_t numDRV

  number of active discrete real variables

- size_t numICV

  number of inactive continuous variables

- size_t numIDIV

  number of inactive discrete integer variables

- size_t numIDRV

  number of inactive discrete real variables

- RealVector continuousVars

  the active continuous variables array view

- IntVector discreteIntVars

  the active discrete integer variables array view

- RealVector discreteRealVars

  the active discrete real variables array view

- RealVector inactiveContinuousVars

  the inactive continuous variables array view

- IntVector inactiveDiscreteIntVars

  the inactive discrete integer variables array view

- RealVector inactiveDiscreteRealVars

  the inactive discrete real variables array view

Private Member Functions

- Variables * get_variables (const ProblemDescDB &problem_db)

  Used by the standard envelope constructor to instantiate the correct letter class.

- Variables * get_variables (const SharedVariablesData &svd) const

  Used by the alternate envelope constructors, by read functions, and by copy() to instantiate a new letter class.

- void check_view_compatibility ()

  perform sanity checks on view.first and view.second after update

Private Attributes

- Variables * variablesRep

  pointer to the letter (initialized only for the envelope)

- int referenceCount

  number of objects sharing variablesRep

Friends

- bool operator== (const Variables &vars1, const Variables &vars2)

  equality operator

- bool operator!= (const Variables &vars1, const Variables &vars2)

  inequality operator

- std::size_t hash_value (const Variables &vars)

  hash_value

- bool binary_equal_to (const Variables &vars1, const Variables &vars2)

  binary_equal_to (since 'operator==' is not suitable for boost/hash_set)

43.141.1 Detailed Description

Base class for the variables class hierarchy. The Variables class is the base class for the class hierarchy providing design, uncertain, and state variables for continuous and discrete domains within a Model. Using the fundamental arrays from the input specification, different derived classes define different views of the data. For memory efficiency and enhanced polymorphism, the variables hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (Variables) serves as the envelope and one of the derived classes (selected in Variables::get_variables()) serves as the letter.

43.141.2 Constructor & Destructor Documentation

43.141.2.1 Variables ()

default constructor The default constructor: variablesRep is NULL in this case (a populated problem_db is needed to build a meaningful Variables object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

43.141.2.2 Variables (const ProblemDescDB & *problem_db*)

standard constructor This is the primary envelope constructor which uses problem_db to build a fully populated variables object. It only needs to extract enough data to properly execute get_variables(problem_db), since the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived classes.

References Dakota::abort_handler(), Variables::get_variables(), and Variables::variablesRep.

43.141.2.3 Variables (const SharedVariablesData & *svd*)

alternate constructor for instantiations on the fly This is the alternate envelope constructor for instantiations on the fly. This constructor executes get_variables(view), which invokes the default derived/base constructors, followed by a resize() based on vars_comps.

References Dakota::abort_handler(), Variables::get_variables(), and Variables::variablesRep.

43.141.2.4 Variables (const Variables & *vars*)

copy constructor Copy constructor manages sharing of variablesRep and incrementing of referenceCount.

References Variables::referenceCount, and Variables::variablesRep.

43.141.2.5 ~Variables () `[virtual]`

destructor Destructor decrements referenceCount and only deletes variablesRep when referenceCount reaches zero.

References Variables::referenceCount, and Variables::variablesRep.

43.141.2.6 Variables (BaseConstructor, const ProblemDescDB & *problem_db*, const std::pair< short, short > & *view*) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_variables() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_variables() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~Variables).

43.141.2.7 Variables (BaseConstructor, const SharedVariablesData & *svd*) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139) This constructor is the one which must build the base class data for all derived classes. get_variables() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_variables() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~Variables).

43.141.3 Member Function Documentation

43.141.3.1 Variables operator= (const Variables & *vars*)

assignment operator Assignment operator decrements referenceCount for old variablesRep, assigns new variablesRep, and increments referenceCount for new variablesRep.

References Variables::referenceCount, and Variables::variablesRep.

43.141.3.2 Variables copy () const

for use when a deep copy is needed (the representation is _not_ shared) Deep copies are used for history mechanisms such as bestVariablesArray and data_pairs since these must catalogue copies (and should not change as the representation within currentVariables changes).

References Variables::allContinuousVars, Variables::allDiscreteIntVars, Variables::allDiscreteRealVars, Variables::build_views(), Variables::get_variables(), Variables::sharedVarsData, and Variables::variablesRep.

Referenced by Model::asynch_compute_response(), ApplicationInterface::continuation(), RecastModel::derived_asynch_compute_response(), EffGlobalMinimizer::EffGlobalMinimizer(), SurrogateModel::force_rebuild(), LeastSq::LeastSq(), SurrBasedLocalMinimizer::minimize_surrogates(), Optimizer::Optimizer(), COLINOptimizer::post_run(), Analyzer::read_variables_responses(), RecastModel::RecastModel(), COLINOptimizer::resize_final_points(), DiscrepancyCorrection::search_db(), SurrBasedLocalMinimizer::SurrBasedLocalMinimizer(), Analyzer::update_best(), and NonDLocalReliability::update_mpp_search_data().

43.141.3.3 void build_views () [inline, protected]

construct active/inactive views of all variables arrays

= EMPTY)

= EMPTY)

References Variables::build_active_views(), Variables::build_inactive_views(), Variables::sharedVarsData, SharedVariablesData::view(), and Variables::view().

Referenced by Variables::copy(), MergedVariables::MergedVariables(), MixedVariables::MixedVariables(), Variables::read(), Variables::read_annotated(), MixedVariables::reshape(), and MergedVariables::reshape().

43.141.3.4 Variables ∗ get_variables (const ProblemDescDB & *problem_db*) [private]

Used by the standard envelope constructor to instantiate the correct letter class. Initializes variablesRep to the appropriate derived type, as given by problem_db attributes. The standard derived class constructors are invoked.

References Variables::get_view(), and Variables::view().

Referenced by Variables::copy(), Variables::read(), Variables::read_annotated(), and Variables::Variables().

43.141.3.5 Variables ∗ get_variables (const SharedVariablesData & *svd*) const [private]

Used by the alternate envelope constructors, by read functions, and by copy() to instantiate a new letter class. Initializes variablesRep to the appropriate derived type, as given by view. The default derived class constructors are invoked.

References SharedVariablesData::view().

The documentation for this class was generated from the following files:

- DakotaVariables.H
- DakotaVariables.C

43.142 Verification Class Reference

Base class for managing common aspects of verification studies. Inheritance diagram for Verification::

Protected Member Functions

- Verification (Model &model)

  constructor

- Verification (NoDBBaseConstructor, Model &model)

  alternate constructor for instantiations "on the fly"

- ~Verification ()

  destructor

- void run ()

  run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post

- void print_results (std::ostream &s)

  print the final iterator results

- virtual void perform_verification ()=0

  Redefines the run_iterator virtual function for the PStudy/DACE branch.

43.142.1 Detailed Description

Base class for managing common aspects of verification studies. The Verification base class manages common data and functions, such as those involving ...

43.142.2 Member Function Documentation

43.142.2.1 void run () [inline, protected, virtual]

run portion of run_iterator; implemented by all derived classes and may include pre/post steps in lieu of separate pre/post Virtual run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Iterator.

References Verification::perform_verification().

43.142.2.2 void print_results (std::ostream & *s*) [protected, virtual]

print the final iterator results This virtual function provides additional iterator-specific final results outputs beyond the function evaluation summary printed in finalize_run().

Reimplemented from Analyzer.

Reimplemented in RichExtrapVerification.

The documentation for this class was generated from the following files:

- DakotaVerification.H
- DakotaVerification.C

Chapter 44

File Documentation

44.1 dll_api.C File Reference

This file contains a DakotaRunner class, which launches DAKOTA.

Namespaces

- namespace Dakota

  The primary namespace for DAKOTA.

Functions

- void DAKOTA_DLL_FN dakota_create (int ∗dakota_ptr_int, char ∗logname)

  create and configure a new DakotaRunner, adding it to list of instances

- int DAKOTA_DLL_FN dakota_readInput (int id, char ∗dakotaInput)

  command DakotaRunner instance id to read from file dakotaInput

- void DAKOTA_DLL_FN dakota_get_variable_info (int id, char ∗∗∗pVarNames, int ∗pNumVarNames, char ∗∗∗pRespNames, int ∗pNumRespNames)

  return the variable and response names

- int DAKOTA_DLL_FN dakota_start (int id)

  command DakotaRunner instance id to start (plugin interface and run strategy)

- void DAKOTA_DLL_FN dakota_destroy (int id)

  delete Dakota runner instance id and remove from active list

- void DAKOTA_DLL_FN dakota_stop (int ∗id)

  command DakotaRunner instance id to stop execution

- const char *DAKOTA_DLL_FN dakota_getStatus (int id)

  return current results output as a string

- int get_mc_ptr_int ()

  get the DAKOTA pointer to ModelCenter

- void set_mc_ptr_int (int ptr_int)

  set the DAKOTA pointer to ModelCenter

- int get_dc_ptr_int ()

  get the DAKOTA pointer to ModelCenter current design point

- void set_dc_ptr_int (int ptr_int)

  set the DAKOTA pointer to ModelCenter current design point

Variables

- PRPCache data_pairs

  contains all parameter/response pairs.

44.1.1 Detailed Description

This file contains a DakotaRunner class, which launches DAKOTA.

44.1.2 Function Documentation

44.1.2.1 void DAKOTA_DLL_FN dakota_stop (int * *id*)

command DakotaRunner instance id to stop execution

TODO: trick application to quit through the syscall interface or throw exception.

44.2 dll_api.h File Reference

API for DLL interactions.

Functions

- void DAKOTA_DLL_FN dakota_create (int ∗dakota_ptr_int, char ∗logname)

  create and configure a new DakotaRunner, adding it to list of instances

- int DAKOTA_DLL_FN dakota_readInput (int id, char ∗dakotaInput)

  command DakotaRunner instance id to read from file dakotaInput

- int DAKOTA_DLL_FN dakota_start (int id)

  command DakotaRunner instance id to start (plugin interface and run strategy)

- void DAKOTA_DLL_FN dakota_destroy (int id)

  delete Dakota runner instance id and remove from active list

- void DAKOTA_DLL_FN dakota_stop (int ∗id)

  command DakotaRunner instance id to stop execution

- const char ∗DAKOTA_DLL_FN dakota_getStatus (int id)

  return current results output as a string

- int DAKOTA_DLL_FN get_mc_ptr_int ()

  get the DAKOTA pointer to ModelCenter

- void DAKOTA_DLL_FN set_mc_ptr_int (int ptr_int)

  set the DAKOTA pointer to ModelCenter

- int DAKOTA_DLL_FN get_dc_ptr_int ()

  get the DAKOTA pointer to ModelCenter current design point

- void DAKOTA_DLL_FN set_dc_ptr_int (int ptr_int)

  set the DAKOTA pointer to ModelCenter current design point

- void DAKOTA_DLL_FN dakota_get_variable_info (int id, char ∗∗∗pVarNames, int ∗pNumVarNames, char ∗∗∗pRespNames, int ∗pNumRespNames)

  return the variable and response names

44.2.1 Detailed Description

API for DLL interactions.

44.2.2 Function Documentation

44.2.2.1 void DAKOTA_DLL_FN dakota_stop (int * *id*)

command DakotaRunner instance id to stop execution

TODO: trick application to quit through the syscall interface or throw exception.

44.3 JEGAOptimizer.C File Reference

Contains the implementation of the JEGAOptimizer class.

Classes

- class Evaluator

  An evaluator specialization that knows how to interact with Dakota.

- class EvaluatorCreator

  A specialization of the JEGA::FrontEnd::EvaluatorCreator that creates a new instance of a Evaluator.

- class Driver

  A subclass of the JEGA front end driver that exposes the individual protected methods to execute the algorithm.

Namespaces

- namespace Dakota

  The primary namespace for DAKOTA.

Functions

- template<typename T >
  string asstring (const T &val)

  Creates a string from the argument val using an ostringstream.

44.3.1 Detailed Description

Contains the implementation of the JEGAOptimizer class.

44.4 JEGAOptimizer.H File Reference

Contains the definition of the JEGAOptimizer class.

Classes

- class JEGAOptimizer

  A version of Dakota::Optimizer for instantiation of John Eddy's Genetic Algorithms (JEGA).

Namespaces

- namespace Dakota

  The primary namespace for DAKOTA.

44.4.1 Detailed Description

Contains the definition of the JEGAOptimizer class.

44.5 library_mode.C File Reference

file containing a mock simulator main for testing DAKOTA in library mode

Namespaces

- namespace Dakota

  The primary namespace for DAKOTA.

Functions

- void nidr_set_input_string (const char ∗)

  Set input to NIDR via string argument instead of input file.

- int **nidr_save_exedir** (const char ∗, int)
- void run_dakota_parse (const char ∗dakota_input_file)

  Function to encapsulate the DAKOTA object instantiations for mode 1: parsing an input file.

- void run_dakota_data ()

  Function to encapsulate the DAKOTA object instantiations for mode 2: direct Data class instantiation.

- void run_dakota_mixed (const char ∗dakota_input_file)

  Function to encapsulate the DAKOTA object instantiations for mode 3: mixed parsing and direct updating.

- void model_interface_plugins (Dakota::ProblemDescDB &problem_db)
- int main (int argc, char ∗argv[ ])

  A mock simulator main for testing DAKOTA in library mode.

- static void my_callback_function (void ∗ptr)

44.5.1 Detailed Description

file containing a mock simulator main for testing DAKOTA in library mode

44.5.2 Function Documentation

44.5.2.1 void run_dakota_parse (const char ∗ *dakota_input_file*)

Function to encapsulate the DAKOTA object instantiations for mode 1: parsing an input file. This function parses from an input file to define the ProblemDescDB data.

References ProblemDescDB::lock(), ProblemDescDB::manage_inputs(), model_interface_plugins(), Strategy::run_strategy(), and ParallelLibrary::world_rank().

Referenced by main().

44.5.2.2 void run_dakota_mixed (const char * *dakota_input_file*)

Function to encapsulate the DAKOTA object instantiations for mode 3: mixed parsing and direct updating. This function showcases multiple features. For parsing, either an input file (dakota_input_file != NULL) or a default input string (dakota_input_file == NULL) are shown. This parsed input is then mixed with input from three sources: (1) input from a user-supplied callback function, (2) updates to the DB prior to Strategy instantiation, (3) updates directly to Iterators/Models following Strategy instantiation.

References ProblemDescDB::broadcast(), ProblemDescDB::get_dsa(), ProblemDescDB::lock(), model_interface_plugins(), ProblemDescDB::model_list(), ParallelLibrary::mpirun_flag(), my_callback_function(), nidr_set_input_string(), ProblemDescDB::parse_inputs(), ProblemDescDB::post_process(), ProblemDescDB::resolve_top_method(), Strategy::run_strategy(), ProblemDescDB::set(), and ParallelLibrary::world_rank().

Referenced by main().

44.5.2.3 void model_interface_plugins (Dakota::ProblemDescDB & *problem_db*)

Iterate over models and plugin appropriate interface: serial rosenbrock or parallel textbook.

References Dakota::abort_handler(), Interface::analysis_drivers(), Interface::assign_rep(), Interface::interface_type(), ProblemDescDB::model_list(), ParallelLevel::server_intra_communicator(), and ProblemDescDB::set_db_model_nodes().

Referenced by Dakota::run_dakota_data(), run_dakota_mixed(), and run_dakota_parse().

44.5.2.4 int main (int *argc*, char * *argv*[ ])

A mock simulator main for testing DAKOTA in library mode. Uses alternative instantiation syntax as described in the library mode documentation within the Developers Manual. Tests several problem specification modes: (1) run_dakota_parse: reads all problem specification data from an input file (2) run_dakota_data: creates all problem specification from direct Data instance instantiations. (3) run_dakota_mixed: a mixture of input parsing (by file or default string) and direct data updates, where the data updates occur: (a) via the DB prior to Strategy instantiation, and (b) via Iterators/Models following Strategy instantiation. Usage: dakota_library_mode [-m] [dakota.in]

References ParallelLibrary::detect_parallel_launch(), Dakota::run_dakota_data(), run_dakota_mixed(), and run_dakota_parse().

44.5.2.5 static void my_callback_function (void * *ptr*) `[static]`

Example of user-provided callback function to override input specified and managed by NIDR, e.g., from an input deck.

References ProblemDescDB::get_dsa(), ProblemDescDB::get_string(), ProblemDescDB::resolve_top_method(), and ProblemDescDB::set().

Referenced by run_dakota_mixed().

44.6 library_split.C File Reference

file containing a mock simulator main for testing DAKOTA in library mode on a split communicator

Functions

- void manage_mpi (MPI_Comm &my_comm, int &color)

  Split MPI_COMM_WORLD, returning the comm and color.

- void gen_dakota_input (const int &color, std::string &input)

  Return the appropriate DAKOTA input based on color (1 or 2).

- void run_dakota (const MPI_Comm &comm, const std::string &input, const int &color)

  Launch DAKOTA on passed communicator, tagging output/error with color.

- void collect_results ()

  Wait for and collect results from DAKOTA runs.

- void nidr_set_input_string (const char ∗)

  Set input to NIDR via string argument instead of input file.

- int main (int argc, char ∗argv[ ])

  Driver routine for testing library mode with partitioned MPI_Comm. This test fixture requires MPI and can be run on 3--8 processors.

44.6.1 Detailed Description

file containing a mock simulator main for testing DAKOTA in library mode on a split communicator

44.7 main.C File Reference

file containing the main program for DAKOTA

Functions

- void fpinit_ASL ()
- int **nidr_save_exedir** (const char ∗, int)
- int main (int argc, char ∗argv[ ])

  The main DAKOTA program.

44.7.1 Detailed Description

file containing the main program for DAKOTA

44.7.2 Function Documentation

44.7.2.1 void fpinit_ASL ()

Floating-point initialization from AMPL: switch to 53-bit rounding if appropriate, to eliminate some cross-platform differences.

Referenced by main().

44.7.2.2 int main (int *argc*, char ∗ *argv*[ ])

The main DAKOTA program. Manage command line inputs, input files, restart file(s), output streams, and top level parallel iterator communicators. Instantiate the Strategy and invoke its run_strategy() virtual function.

References Dakota::abort_handler(), fpinit_ASL(), CommandLineHandler::instantiate_flag(), ProblemDescDB::lock(), ProblemDescDB::manage_inputs(), ParallelLibrary::output_helper(), GetLongOpt::retrieve(), Strategy::run_strategy(), ParallelLibrary::specify_outputs_restart(), and ParallelLibrary::world_rank().

44.8 restart_util.C File Reference

file containing the DAKOTA restart utility main program

Namespaces

- namespace Dakota

  The primary namespace for DAKOTA.

Functions

- void print_restart (int argc, char **argv, String print_dest)

  print a restart file

- void print_restart_tabular (int argc, char **argv, String print_dest)

  print a restart file (tabular format)

- void read_neutral (int argc, char **argv)

  read a restart file (neutral file format)

- void repair_restart (int argc, char **argv, String identifier_type)

  repair a restart file by removing corrupted evaluations

- void concatenate_restart (int argc, char **argv)

  concatenate multiple restart files

- int **nidr_save_exedir** (const char *, int)
- int main (int argc, char *argv[ ])

  The main program for the DAKOTA restart utility.

44.8.1 Detailed Description

file containing the DAKOTA restart utility main program

44.8.2 Function Documentation

44.8.2.1 int main (int *argc*, char * *argv*[ ])

The main program for the DAKOTA restart utility. Parse command line inputs and invoke the appropriate utility function (print_restart(), print_restart_tabular(), read_neutral(), repair_restart(), or concatenate_restart()).

References Dakota::concatenate_restart(), Dakota::print_restart(), Dakota::print_restart_tabular(), Dakota::read_-neutral(), and Dakota::repair_restart().

Index

~Approximation
Dakota::Approximation, 299
~BiStream
Dakota::BiStream, 319
~Constraints
Dakota::Constraints, 358
~EffGlobalMinimizer
Dakota::EffGlobalMinimizer, 455
~Interface
Dakota::Interface, 511
~Iterator
Dakota::Iterator, 520
~Model
Dakota::Model, 582
~ProblemDescDB
Dakota::ProblemDescDB, 794
~Strategy
Dakota::Strategy, 873
~Variables
Dakota::Variables, 930
_initPts
Dakota::JEGAOptimizer, 535
_model
Dakota::JEGAOptimizer::Evaluator, 464

A
Dakota::CONMINOptimizer, 348
abort_handler_t
Dakota, 172
accepts_multiple_points
Dakota::JEGAOptimizer, 534
actualModel
Dakota::DataFitSurrModel, 374
add_anchor_to_surfdata
Dakota::SurfpackApproximation, 884
add_datapoint
Dakota::Graphics, 491
allContinuousIds
Dakota::SharedVariablesDataRep, 844
anisotropic_preference
Dakota::NonDQuadrature, 715
append_approximation
Dakota::ApproximationInterface, 306, 307
Dakota::DataFitSurrModel, 370
approx_subprob_constraint_eval
Dakota::SurrBasedLocalMinimizer, 894
approx_subprob_objective_eval
Dakota::SurrBasedLocalMinimizer, 894
approxBuilds
Dakota::SurrogateModel, 905
Approximation
Dakota::Approximation, 299, 300
APPSEvalMgr
Dakota::APPSEvalMgr, 311
APPSOptimizer
Dakota::APPSOptimizer, 313
assign_rep
Dakota::Interface, 511
Dakota::Iterator, 524
Dakota::Model, 587
asstring
Dakota, 173
asynchronous_local_analyses
Dakota::ForkApplicInterface, 471
asynchronous_local_evaluations
Dakota::ApplicationInterface, 290
asynchronous_local_evaluations_nowait
Dakota::ApplicationInterface, 292
asynchronous_local_evaluations_static
Dakota::ApplicationInterface, 291
augmented_lagrangian_merit
Dakota::SurrBasedMinimizer, 900

B
Dakota::CONMINOptimizer, 347
begins
Dakota::String, 879
Bgen
Dakota, 261
BiStream
Dakota::BiStream, 318
BoStream

Dakota::BoStream, 321
build
Dakota::Approximation, 300
Dakota::PecosApproximation, 784
Dakota::SurfpackApproximation, 883
Dakota::TANA3Approximation, 913
Dakota::TaylorApproximation, 916
build_approximation
Dakota::ApproximationInterface, 307
Dakota::DataFitSurrModel, 368, 369
build_global
Dakota::DataFitSurrModel, 371
build_local_multipoint
Dakota::DataFitSurrModel, 371
build_views
Dakota::Constraints, 360
Dakota::Variables, 931
buildDataOrder
Dakota::Approximation, 302

C
Dakota::CONMINOptimizer, 347
CAUVLbl
Dakota, 258
ccv_index_map
Dakota::NestedModel, 609
cdiv_index_map
Dakota::NestedModel, 609
cdrv_index_map
Dakota::NestedModel, 609
CEUVLbl
Dakota, 259
check_status
Dakota::ForkAnalysisCode, 468
check_variables
Dakota::NonDIntegration, 679
clear_all
Dakota::Approximation, 301
clear_current
Dakota::Approximation, 301
Dakota::TANA3Approximation, 913
Clone
Dakota::JEGAOptimizer::Evaluator, 463
close_streams
Dakota::ParallelLibrary, 766
colin_request_to_dakota_request
Dakota::COLINApplication, 325
COLINOptimizer
Dakota::COLINOptimizer, 329
collect_evaluation_impl
Dakota::COLINApplication, 325
compute
Dakota::DiscrepancyCorrection, 444
compute_final_statistics_metric
Dakota::NonDExpansion, 658
compute_statistics
Dakota::NonDExpansion, 658
concatenate_restart
Dakota, 175
conminInfo
Dakota::CONMINOptimizer, 345
constraint0_evaluator
Dakota::SNLLOptimizer, 864
constraint1_evaluator
Dakota::SNLLOptimizer, 864
constraint1_evaluator_gn
Dakota::SNLLLeastSq, 856
constraint2_evaluator
Dakota::SNLLOptimizer, 864
constraint2_evaluator_gn
Dakota::SNLLLeastSq, 856
constraint_violation
Dakota::SurrBasedMinimizer, 900
constraintMappingIndices
Dakota::CONMINOptimizer, 345
Dakota::DOTOptimizer, 449
constraintMappingMultipliers
Dakota::CONMINOptimizer, 346
Dakota::DOTOptimizer, 450
constraintMappingOffsets
Dakota::CONMINOptimizer, 346
Dakota::DOTOptimizer, 450
Constraints
Dakota::Constraints, 358
constraintValues
Dakota::CONMINOptimizer, 345
Dakota::DOTOptimizer, 449
contains
Dakota::String, 879
converge_order
Dakota::RichExtrapVerification, 830
converge_qoi
Dakota::RichExtrapVerification, 830
copy
Dakota::Constraints, 359
Dakota::Variables, 931
copy_data
Dakota::DDACEDesignCompExp, 430
create_plots_2d

Dakota::Graphics, 490
create_tabular_datastream
Dakota::Graphics, 491
CreateEvaluator
Dakota::JEGAOptimizer::EvaluatorCreator, 465
CT
Dakota::CONMINOptimizer, 347
cv_index_map
Dakota::NestedModel, 608

DakFuncs0
Dakota, 176
Dakota, 107
abort_handler_t, 172
asstring, 173
Bgen, 261
CAUVLbl, 258
CEUVLbl, 259
concatenate_restart, 175
DakFuncs0, 176
DAUIVLbl, 259
DAURVLbl, 259
DiscSetLbl, 259
FIELD_NAMES, 176
flush, 171
get_cwd, 171
get_npath, 171
getdist, 172
getRmax, 172
id_vars_exact_compare, 173
kw_1, 177
kw_10, 180
kw_100, 206
kw_101, 206
kw_102, 206
kw_103, 208
kw_104, 208
kw_105, 208
kw_106, 209
kw_107, 209
kw_108, 209
kw_109, 209
kw_11, 181
kw_110, 210
kw_111, 210
kw_112, 210
kw_113, 210
kw_114, 211
kw_115, 211
kw_116, 211
kw_117, 211
kw_118, 211
kw_119, 212
kw_12, 181
kw_120, 213
kw_121, 213
kw_122, 213
kw_123, 213
kw_124, 214
kw_125, 214
kw_126, 214
kw_127, 215
kw_128, 215
kw_129, 215
kw_13, 182
kw_130, 215
kw_131, 215
kw_132, 216
kw_133, 217
kw_134, 217
kw_135, 218
kw_136, 219
kw_137, 219
kw_138, 219
kw_139, 220
kw_14, 182
kw_140, 220
kw_141, 220
kw_142, 220
kw_143, 221
kw_144, 221
kw_145, 221
kw_146, 221
kw_147, 222
kw_148, 223
kw_149, 223
kw_15, 182
kw_150, 224
kw_151, 224
kw_152, 224
kw_153, 224
kw_154, 225
kw_155, 226
kw_157, 226
kw_158, 226
kw_159, 227
kw_16, 182
kw_160, 227
kw_161, 227

kw_162, 227
kw_163, 227
kw_164, 228
kw_165, 228
kw_166, 228
kw_167, 228
kw_168, 229
kw_169, 229
kw_17, 183
kw_170, 229
kw_171, 229
kw_172, 230
kw_173, 230
kw_174, 230
kw_175, 230
kw_176, 231
kw_177, 231
kw_178, 232
kw_179, 232
kw_18, 183
kw_180, 232
kw_181, 233
kw_182, 233
kw_183, 233
kw_184, 234
kw_185, 234
kw_186, 235
kw_187, 235
kw_188, 235
kw_189, 235
kw_19, 183
kw_190, 236
kw_191, 236
kw_192, 236
kw_193, 236
kw_194, 237
kw_195, 237
kw_196, 238
kw_197, 238
kw_198, 238
kw_199, 238
kw_2, 177
kw_20, 183
kw_200, 239
kw_201, 239
kw_202, 240
kw_203, 240
kw_204, 240
kw_205, 240
kw_206, 241
kw_207, 241
kw_208, 241
kw_209, 241
kw_21, 183
kw_210, 242
kw_211, 242
kw_212, 243
kw_213, 243
kw_214, 244
kw_215, 244
kw_216, 245
kw_217, 245
kw_218, 245
kw_219, 246
kw_22, 184
kw_220, 246
kw_221, 246
kw_222, 247
kw_223, 247
kw_224, 247
kw_225, 248
kw_226, 248
kw_227, 249
kw_228, 249
kw_229, 249
kw_23, 184
kw_230, 250
kw_231, 250
kw_232, 250
kw_233, 251
kw_234, 251
kw_235, 252
kw_236, 252
kw_237, 252
kw_238, 253
kw_239, 253
kw_24, 184
kw_241, 254
kw_242, 254
kw_243, 254
kw_244, 255
kw_245, 255
kw_246, 255
kw_247, 256
kw_248, 256
kw_249, 256
kw_25, 185
kw_250, 257

kw_251, 257
kw_252, 257
kw_253, 258
kw_255, 258
kw_26, 185
kw_27, 186
kw_28, 186
kw_29, 186
kw_3, 178
kw_30, 186
kw_31, 186
kw_32, 187
kw_33, 187
kw_34, 188
kw_35, 188
kw_36, 188
kw_37, 188
kw_38, 189
kw_39, 190
kw_4, 178
kw_40, 191
kw_41, 191
kw_42, 191
kw_43, 191
kw_44, 192
kw_45, 193
kw_46, 193
kw_47, 193
kw_48, 193
kw_49, 194
kw_5, 178
kw_50, 194
kw_51, 194
kw_52, 195
kw_53, 195
kw_54, 195
kw_55, 195
kw_56, 196
kw_57, 196
kw_58, 196
kw_59, 196
kw_6, 178
kw_60, 196
kw_61, 197
kw_62, 197
kw_63, 197
kw_64, 198
kw_65, 198
kw_66, 198
kw_67, 198
kw_68, 198
kw_69, 199
kw_7, 179
kw_70, 199
kw_71, 199
kw_72, 199
kw_73, 200
kw_74, 200
kw_75, 200
kw_76, 200
kw_77, 201
kw_78, 201
kw_79, 201
kw_8, 179
kw_80, 201
kw_81, 201
kw_82, 202
kw_83, 202
kw_84, 202
kw_85, 202
kw_86, 203
kw_87, 203
kw_88, 203
kw_89, 203
kw_9, 180
kw_90, 203
kw_91, 204
kw_92, 204
kw_93, 204
kw_94, 205
kw_95, 205
kw_96, 205
kw_97, 205
kw_98, 206
kw_99, 206
lookup_by_val, 174
mindist, 172
mindistindx, 172
my_cp, 171
NUMBER_OF_FIELDS, 176
path_is_absolute, 175
perform_analysis, 173
print_restart, 174
print_restart_tabular, 174
PRPMultiIndexCache, 170
PRPMultiIndexQueue, 170
read_neutral, 175
rel_change_rv, 171

repair_restart, 175
run_dakota_data, 173
set_compare, 173
slmap, 176
start_dakota_heartbeat, 171
start_grid_computing, 172
stop_grid_computing, 172
var_mp_bgen, 260
var_mp_bgen_audi, 261
var_mp_bgen_audr, 261
var_mp_bgen_dis, 261
var_mp_bgen_eu, 261
var_mp_bndchk, 262
var_mp_ibndchk, 262
Vlch, 259
VLS, 260
Dakota::ActiveSet, 265
derivVarsVector, 267
requestVector, 267
Dakota::AnalysisCode, 269
Dakota::Analyzer, 274
evaluate_parameter_sets, 278
pre_output, 277
print_results, 277
print_sobol_indices, 279
read_variables_responses, 278
variance_based_decomp, 278
Dakota::ApplicationInterface, 280
asynchronous_local_evaluations, 290
asynchronous_local_evaluations_nowait, 292
asynchronous_local_evaluations_static, 291
duplication_detect, 289
init_serial, 287
map, 287
self_schedule_analyses, 288
self_schedule_evaluations, 290
serve_analyses_synch, 289
serve_evaluations, 288
serve_evaluations_asynch, 292
serve_evaluations_peer, 293
serve_evaluations_synch, 292
static_schedule_evaluations, 290
stop_evaluation_servers, 288
synch, 287
synch_nowait, 288
synchronous_local_evaluations, 291
Dakota::Approximation, 294
∼Approximation, 299
Approximation, 299, 300
build, 300
buildDataOrder, 302
clear_all, 301
clear_current, 301
finalize, 301
get_approx, 302
operator=, 300
pop, 300
rebuild, 300
restore, 301
Dakota::ApproximationInterface, 303
append_approximation, 306, 307
build_approximation, 307
functionSurfaces, 308
pop_approximation, 307
rebuild_approximation, 307
restore_approximation, 308
restore_available, 308
update_approximation, 306
Dakota::APPSEvalMgr, 309
APPSEvalMgr, 311
isReadyForWork, 311
recv, 311
submit, 311
Dakota::APPSOptimizer, 312
APPSOptimizer, 313
find_optimum, 314
initialize_variables_and_constraints, 314
set_apps_parameters, 314
Dakota::BaseConstructor, 316
Dakota::BiStream, 317
∼BiStream, 319
BiStream, 318
operator>>, 319
Dakota::BoStream, 320
BoStream, 321
operator<<, 322
Dakota::COLINApplication, 323
colin_request_to_dakota_request, 325
collect_evaluation_impl, 325
dakota_response_to_colin_response, 326
evaluation_available, 325
map_domain, 326
perform_evaluation_impl, 325
set_problem, 324
spawn_evaluation_impl, 324
Dakota::COLINOptimizer, 327
COLINOptimizer, 329
find_optimum, 329

post_run, 330
returns_multiple_points, 330
set_rng, 330
set_solver_parameters, 330
solver_setup, 330
Dakota::CollaborativeHybridStrategy, 332
Dakota::CommandLineHandler, 334
instantiate_flag, 335
Dakota::CommandShell, 336
flush, 337
operator<<, 337
Dakota::ConcurrentStrategy, 338
pack_parameters_buffer, 339
pack_results_buffer, 340
unpack_parameters_buffer, 340
unpack_results_buffer, 340
Dakota::CONMINOptimizer, 341
A, 348
B, 347
C, 347
conminInfo, 345
constraintMappingIndices, 345
constraintMappingMultipliers, 346
constraintMappingOffsets, 346
constraintValues, 345
CT, 347
DF, 348
G1, 347
G2, 347
IC, 348
ISC, 348
MS1, 347
N1, 346
N2, 346
N3, 346
N4, 346
N5, 346
optimizationType, 345
printControl, 345
S, 347
SCAL, 348
Dakota::Constraints, 349
~Constraints, 358
build_views, 360
Constraints, 358
copy, 359
get_constraints, 360
manage_linear_constraints, 360
operator=, 359
reshape, 359
Dakota::DataFitSurrModel, 362
actualModel, 374
append_approximation, 370
build_approximation, 368, 369
build_global, 371
build_local_multipoint, 371
derived_asynch_compute_response, 367
derived_compute_response, 367
derived_init_communicators, 371
derived_synchronize, 368
derived_synchronize_nowait, 368
evaluation_id, 371
update_actual_model, 372
update_approximation, 369, 370
update_from_actual_model, 373
Dakota::DataInterface, 375
Dakota::DataMethod, 377
Dakota::DataMethodRep, 379
Dakota::DataModel, 394
Dakota::DataModelRep, 396
Dakota::DataResponses, 401
Dakota::DataResponsesRep, 403
Dakota::DataStrategy, 408
Dakota::DataStrategyRep, 410
Dakota::DataVariables, 413
Dakota::DataVariablesRep, 415
Dakota::DDACEDesignCompExp, 426
copy_data, 430
DDACEDesignCompExp, 428
num_samples, 429
post_run, 429
pre_run, 429
resolve_samples_symbols, 429
Dakota::DirectApplicInterface, 431
derived_map_ac, 438
derived_synchronous_local_analysis, 438
herbie, 439
herbie1D, 439
python_convert_int, 440
separable_combine, 440
shubert1D, 439
smooth_herbie, 439
smooth_herbie1D, 439
Dakota::DiscrepancyCorrection, 441
compute, 444
Dakota::DOTOptimizer, 446
constraintMappingIndices, 449
constraintMappingMultipliers, 450

constraintMappingOffsets, 450
constraintValues, 449
dotFDSinfo, 448
dotInfo, 448
dotMethod, 449
intCntlParmArray, 449
optimizationType, 449
printControl, 449
realCntlParmArray, 449
Dakota::EffGlobalMinimizer, 453
~EffGlobalMinimizer, 455
get_best_sample, 455
Dakota::EmbeddedHybridStrategy, 456
Dakota::ForkAnalysisCode, 467
check_status, 468
Dakota::ForkApplicInterface, 469
asynchronous_local_analyses, 471
derived_synchronous_local_analysis, 470
fork_application, 470
serve_analyses_asynch, 471
synchronous_local_analyses, 471
Dakota::FSUDesignCompExp, 473
enforce_input_rules, 476
FSUDesignCompExp, 475
num_samples, 476
post_run, 476
pre_run, 476
Dakota::GaussProcApproximation, 478
GaussProcApproximation, 483
GPmodel_apply, 483
trendOrder, 483
Dakota::GetLongOpt, 485
enroll, 487
GetLongOpt, 487
MandatoryValue, 487
OptionalValue, 486
OptType, 486
parse, 487
retrieve, 487
usage, 488
Valueless, 486
Dakota::Graphics, 489
add_datapoint, 491
create_plots_2d, 490
create_tabular_datastream, 491
new_dataset, 491
Dakota::GridApplicInterface, 493
derived_synch_kernel, 495
derived_synchronous_local_analysis, 495
Dakota::HierarchSurrModel, 496
derived_asynch_compute_response, 499
derived_compute_response, 499
derived_synchronize, 499
derived_synchronize_nowait, 500
evaluation_id, 500
Dakota::HybridStrategy, 501
Dakota::Interface, 503
~Interface, 511
assign_rep, 511
get_interface, 512
Interface, 510, 511
operator=, 511
rawResponseMap, 512
Dakota::Iterator, 513
~Iterator, 520
assign_rep, 524
fdGradStepSize, 525
fdHessByFnStepSize, 525
fdHessByGradStepSize, 525
finalize_run, 522
get_iterator, 525
initialize_graphics, 523
initialize_run, 521
Iterator, 520, 521
num_samples, 523
operator=, 521
post_run, 522
pre_run, 522
print_results, 523
run, 522
run_iterator, 524
Dakota::JEGAOptimizer, 527
_initPts, 535
accepts_multiple_points, 534
find_optimum, 533
GetBestMOSolutions, 532
GetBestSolutions, 532
GetBestSOSolutions, 532
initial_points, 534
JEGAOptimizer, 529
LoadAlgorithmConfig, 530
LoadDakotaResponses, 530
LoadProblemConfig, 531
LoadTheConstraints, 531
LoadTheDesignVariables, 531
LoadTheObjectiveFunctions, 531
LoadTheParameterDatabase, 530
resize_response_results_array, 533

resize_variables_results_array, 533
returns_multiple_points, 534
ToDoubleMatrix, 532
Dakota::JEGAOptimizer::Driver, 451
DestroyAlgorithm, 452
Driver, 451
ExtractAllData, 451
PerformIterations, 452
Dakota::JEGAOptimizer::Evaluator, 458
_model, 464
Clone, 463
Description, 460
Evaluate, 462
Evaluator, 459, 460
GetDescription, 463
GetName, 463
GetNumberLinearConstraints, 462
GetNumberNonLinearConstraints, 461
Name, 460
RecordResponses, 461
SeparateVariables, 461
Dakota::JEGAOptimizer::EvaluatorCreator, 465
CreateEvaluator, 465
EvaluatorCreator, 465
Dakota::LeastSq, 536
finalize_run, 539
get_confidence_intervals, 540
initialize_run, 538
LeastSq, 538
post_run, 538
primary_resp_recast, 539
print_results, 539
run, 538
Dakota::MergedConstraints, 541
MergedConstraints, 542
reshape, 542
Dakota::MergedVariables, 543
MergedVariables, 544
read_tabular, 544
Dakota::Minimizer, 545
finalize_run, 550
gnewton_set_recast, 551
initialize_run, 550
initialize_scaling, 551
lin_coeffs_modify_n2s, 553
Minimizer, 550
modify_n2s, 552
modify_s2n, 552
need_resp_trans_byvars, 552
objective, 553
objective_gradient, 554
objective_hessian, 554
response_modify_n2s, 553
response_modify_s2n, 553
secondary_resp_recast, 552
variables_recast, 551
Dakota::MixedConstraints, 555
MixedConstraints, 556
reshape, 556
Dakota::MixedVariables, 557
MixedVariables, 558
read_tabular, 558
Dakota::Model, 559
~Model, 582
assign_rep, 587
derivative_concurrency, 588
estimate_derivatives, 589
estimate_message_lengths, 587
evaluation_cache, 586
FDstep1, 588
FDstep2, 588
get_model, 588
init_communicators, 586
init_serial, 587
interface, 584
interface_id, 585
local_eval_concurrency, 585
local_eval_synchronization, 585
manage_asv, 590
Model, 582
operator=, 583
subordinate_iterator, 583
subordinate_model, 583
subordinate_models, 586
surrogate_model, 584
synchronize_derivatives, 589
truth_model, 584
update_from_subordinate_model, 584
update_quasi_hessians, 590
update_response, 589
Dakota::MPIPackBuffer, 592
Dakota::MPIUnpackBuffer, 595
Dakota::NCSUOptimizer, 598
NCSUOptimizer, 600
objective_eval, 601
Dakota::NestedModel, 602
ccv_index_map, 609
cdiv_index_map, 609

cdrv_index_map, 609
cv_index_map, 608
derived_asynch_compute_response, 607
derived_compute_response, 607
derived_init_communicators, 608
derived_master_overload, 608
div_index_map, 608
drv_index_map, 609
evaluation_id, 608
response_mapping, 609
subModel, 611
Dakota::NIDRProblemDescDB, 612
derived_parse_inputs, 616
Dakota::NL2Res, 617
Dakota::NL2SOLLeastSq, 618
minimize_residuals, 620
Dakota::NLPQLPOptimizer, 621
Dakota::NLSSOLLeastSq, 627
NLSSOLLeastSq, 628
Dakota::NoDBBaseConstructor, 629
Dakota::NonD, 630
finalize_run, 637
initialize_final_statistics, 637
initialize_random_variable_parameters, 638
initialize_random_variable_types, 638
initialize_random_variables, 636, 637
initialize_run, 637
run, 637
set_u_to_x_mapping, 639
vars_u_to_x_mapping, 638
Dakota::NonDAdaptImpSampling, 640
initialize, 643
NonDAdaptImpSampling, 642
Dakota::NonDBayesCalibration, 644
NonDBayesCalibration, 645
Dakota::NonDCalibration, 646
NonDCalibration, 648
Dakota::NonDCubature, 649
increment_grid_preference, 651
increment_reference, 651
NonDCubature, 650
num_samples, 651
sampling_reset, 651
Dakota::NonDExpansion, 653
compute_final_statistics_metric, 658
compute_statistics, 658
useDerivs, 659
Dakota::NonDGlobalEvidence, 660
Dakota::NonDGlobalInterval, 662
Dakota::NonDGlobalReliability, 666
Dakota::NonDGlobalSingleInterval, 669
Dakota::NonDGPMSABayesCalibration, 671
NonDGPMSABayesCalibration, 672
quantify_uncertainty, 672
Dakota::NonDIncremLHSSampling, 674
NonDIncremLHSSampling, 675
quantify_uncertainty, 675
Dakota::NonDIntegration, 677
check_variables, 679
NonDIntegration, 678, 679
Dakota::NonDInterval, 680
print_results, 682
Dakota::NonDLHSEvidence, 683
Dakota::NonDLHSInterval, 685
Dakota::NonDLHSSampling, 687
NonDLHSSampling, 688
quantify_uncertainty, 689
Dakota::NonDLHSSingleInterval, 690
Dakota::NonDLocalEvidence, 692
Dakota::NonDLocalInterval, 694
Dakota::NonDLocalReliability, 697
dg_ds_eval, 704
initial_taylor_series, 701
initialize_class_data, 701
initialize_level_data, 702
initialize_mpp_search_data, 702
probability, 704
reliability, 705
update_level_data, 703
update_mpp_search_data, 703
update_pma_reliability_level, 704
Dakota::NonDLocalSingleInterval, 706
Dakota::NonDPolynomialChaos, 708
increment_expansion, 710
NonDPolynomialChaos, 709, 710
Dakota::NonDQuadrature, 711
anisotropic_preference, 715
initialize_grid, 714
NonDQuadrature, 714
num_samples, 715
sampling_reset, 714
Dakota::NonDQUESOBayesCalibration, 716
NonDQUESOBayesCalibration, 718
quantify_uncertainty, 718
Dakota::NonDReliability, 719
PMA_constraint_eval, 722
PMA_objective_eval, 722
RIA_constraint_eval, 722

RIA_objective_eval, 722
Dakota::NonDSampling, 724
get_parameter_sets, 729
NonDSampling, 728
num_samples, 729
sampling_reset, 729
Dakota::NonDSparseGrid, 731
NonDSparseGrid, 733
num_samples, 734
sampling_reset, 734
Dakota::NonDStochCollocation, 735
NonDStochCollocation, 736
Dakota::NPSOLOptimizer, 737
NPSOLOptimizer, 739
Dakota::Optimizer, 741
finalize_run, 744
initialize_run, 744
local_objective_recast_retrieve, 745
objective_reduction, 745
Optimizer, 743
post_run, 744
primary_resp_recast, 745
print_results, 744
run, 744
Dakota::ParallelConfiguration, 747
Dakota::ParallelLevel, 750
Dakota::ParallelLibrary, 754
close_streams, 766
increment_parallel_configuration, 766
init_communicators, 767
init_mpi_comm, 766
manage_outputs_restart, 765
ParallelLibrary, 764, 765
resolve_inputs, 767
specify_outputs_restart, 765
split_filenames, 767
Dakota::ParamResponsePair, 768
evalInterfaceIds, 771
ParamResponsePair, 770
read, 770
write, 770
Dakota::ParamStudy, 772
post_run, 776
pre_run, 775
Dakota::partial_prp_equality, 777
Dakota::partial_prp_hash, 778
Dakota::PecosApproximation, 779
build, 784
finalize, 784
pop, 784
rebuild, 784
restore, 784
Dakota::ProblemDescDB, 786
~ProblemDescDB, 794
get_db, 796
manage_inputs, 795
operator=, 795
parse_inputs, 795
post_process, 796
ProblemDescDB, 794, 795
Dakota::PStudyDACE, 797
print_results, 798
run, 798
volumetric_quality, 799
Dakota::PSUADEDesignCompExp, 800
enforce_input_rules, 803
num_samples, 802
post_run, 802
pre_run, 802
PSUADEDesignCompExp, 802
Dakota::RecastBaseConstructor, 804
Dakota::RecastModel, 805
initialize, 812
RecastModel, 811
update_from_sub_model, 812
Dakota::Response, 814
Response, 818
Dakota::ResponseRep, 820
functionGradients, 826
read, 823–825
read_annotated, 823
read_tabular, 824
reset, 826
reset_inactive, 826
reshape, 826
ResponseRep, 822, 823
update, 825
update_partial, 825
write, 823–825
write_annotated, 824
write_tabular, 824
Dakota::RichExtrapVerification, 828
converge_order, 830
converge_qoi, 830
estimate_order, 830
print_results, 829
Dakota::SensAnalysisGlobal, 831
Dakota::SequentialHybridStrategy, 833

extract_parameter_sets, 836
pack_parameters_buffer, 835
pack_results_buffer, 835
run_sequential, 836
run_sequential_adaptive, 836
unpack_parameters_buffer, 835
unpack_results_buffer, 836
Dakota::SharedVariablesData, 839
Dakota::SharedVariablesDataRep, 842
allContinuousIds, 844
SharedVariablesDataRep, 844
Dakota::SingleMethodStrategy, 845
Dakota::SingleModel, 847
Dakota::SNLLBase, 850
Dakota::SNLLLeastSq, 853
constraint1_evaluator_gn, 856
constraint2_evaluator_gn, 856
nlf2_evaluator_gn, 855
post_run, 855
Dakota::SNLLOptimizer, 858
constraint0_evaluator, 864
constraint1_evaluator, 864
constraint2_evaluator, 864
nlf0_evaluator, 863
nlf1_evaluator, 863
nlf2_evaluator, 863
SNLLOptimizer, 862
Dakota::SOLBase, 866
Dakota::Strategy, 869
∼Strategy, 873
free_iterator, 875
get_strategy, 876
init_iterator, 875
init_iterator_parallelism, 874
operator=, 873
pack_parameters_buffer, 873
pack_results_buffer, 874
run_iterator, 875
schedule_iterators, 876
self_schedule_iterators, 876
serve_iterators, 876
Strategy, 873
unpack_parameters_buffer, 874
unpack_results_buffer, 874
Dakota::String, 878
begins, 879
contains, 879
data, 880
ends, 880
lower, 879
operator const char ∗, 879
upper, 879
Dakota::SurfpackApproximation, 881
add_anchor_to_surfdata, 884
build, 883
get_hessian, 883
SurfpackApproximation, 883
surrogates_to_surf_data, 884
Dakota::SurrBasedGlobalMinimizer, 885
Dakota::SurrBasedLocalMinimizer, 887
approx_subprob_constraint_eval, 894
approx_subprob_objective_eval, 894
hard_convergence_check, 892
hom_constraint_eval, 894
hom_objective_eval, 894
minimize_surrogates, 892
tr_ratio_check, 893
update_penalty, 893
Dakota::SurrBasedMinimizer, 896
augmented_lagrangian_merit, 900
constraint_violation, 900
lagrangian_merit, 899
penalty_merit, 900
print_results, 898
run, 898
update_augmented_lagrange_multipliers, 899
update_filter, 899
update_lagrange_multipliers, 899
Dakota::SurrogateModel, 902
approxBuilds, 905
force_rebuild, 905
responseMode, 905
Dakota::SysCallAnalysisCode, 907
spawn_analysis, 908
spawn_evaluation, 907
spawn_input_filter, 908
spawn_output_filter, 908
Dakota::SysCallApplicInterface, 909
derived_synch, 910
derived_synch_nowait, 910
derived_synchronous_local_analysis, 910
Dakota::TANA3Approximation, 912
build, 913
clear_current, 913
Dakota::TaylorApproximation, 915
build, 916
Dakota::TrackerHTTP, 917
send_data_using_get, 919

send_data_using_post, 919
Dakota::Variables, 920
~Variables, 930
build_views, 931
copy, 931
get_variables, 932
operator=, 931
Variables, 930, 931
Dakota::Verification, 933
print_results, 934
run, 934
dakota_response_to_colin_response
Dakota::COLINApplication, 326
dakota_stop
dll_api.C, 936
dll_api.h, 938
data
Dakota::String, 880
DAUIVLbl
Dakota, 259
DAURVLbl
Dakota, 259
DDACEDesignCompExp
Dakota::DDACEDesignCompExp, 428
derivative_concurrency
Dakota::Model, 588
derived_asynch_compute_response
Dakota::DataFitSurrModel, 367
Dakota::HierarchSurrModel, 499
Dakota::NestedModel, 607
derived_compute_response
Dakota::DataFitSurrModel, 367
Dakota::HierarchSurrModel, 499
Dakota::NestedModel, 607
derived_init_communicators
Dakota::DataFitSurrModel, 371
Dakota::NestedModel, 608
derived_map_ac
Dakota::DirectApplicInterface, 438
derived_master_overload
Dakota::NestedModel, 608
derived_parse_inputs
Dakota::NIDRProblemDescDB, 616
derived_synch
Dakota::SysCallApplicInterface, 910
derived_synch_kernel
Dakota::GridApplicInterface, 495
derived_synch_nowait
Dakota::SysCallApplicInterface, 910
derived_synchronize
Dakota::DataFitSurrModel, 368
Dakota::HierarchSurrModel, 499
derived_synchronize_nowait
Dakota::DataFitSurrModel, 368
Dakota::HierarchSurrModel, 500
derived_synchronous_local_analysis
Dakota::DirectApplicInterface, 438
Dakota::ForkApplicInterface, 470
Dakota::GridApplicInterface, 495
Dakota::SysCallApplicInterface, 910
derivVarsVector
Dakota::ActiveSet, 267
Description
Dakota::JEGAOptimizer::Evaluator, 460
DestroyAlgorithm
Dakota::JEGAOptimizer::Driver, 452
DF
Dakota::CONMINOptimizer, 348
dg_ds_eval
Dakota::NonDLocalReliability, 704
DiscSetLbl
Dakota, 259
div_index_map
Dakota::NestedModel, 608
dll_api.C, 935
dakota_stop, 936
dll_api.h, 937
dakota_stop, 938
dotFDSinfo
Dakota::DOTOptimizer, 448
dotInfo
Dakota::DOTOptimizer, 448
dotMethod
Dakota::DOTOptimizer, 449
Driver
Dakota::JEGAOptimizer::Driver, 451
drv_index_map
Dakota::NestedModel, 609
duplication_detect
Dakota::ApplicationInterface, 289

ends
Dakota::String, 880
enforce_input_rules
Dakota::FSUDesignCompExp, 476
Dakota::PSUADEDesignCompExp, 803
enroll
Dakota::GetLongOpt, 487
estimate_derivatives

Dakota::Model, 589
estimate_message_lengths
Dakota::Model, 587
estimate_order
Dakota::RichExtrapVerification, 830
evalInterfaceIds
Dakota::ParamResponsePair, 771
Evaluate
Dakota::JEGAOptimizer::Evaluator, 462
evaluate_parameter_sets
Dakota::Analyzer, 278
evaluation_available
Dakota::COLINApplication, 325
evaluation_cache
Dakota::Model, 586
evaluation_id
Dakota::DataFitSurrModel, 371
Dakota::HierarchSurrModel, 500
Dakota::NestedModel, 608
Evaluator
Dakota::JEGAOptimizer::Evaluator, 459, 460
EvaluatorCreator
Dakota::JEGAOptimizer::EvaluatorCreator, 465
extract_parameter_sets
Dakota::SequentialHybridStrategy, 836
ExtractAllData
Dakota::JEGAOptimizer::Driver, 451

fdGradStepSize
Dakota::Iterator, 525
fdHessByFnStepSize
Dakota::Iterator, 525
fdHessByGradStepSize
Dakota::Iterator, 525
FDstep1
Dakota::Model, 588
FDstep2
Dakota::Model, 588
FIELD_NAMES
Dakota, 176
finalize
Dakota::Approximation, 301
Dakota::PecosApproximation, 784
finalize_run
Dakota::Iterator, 522
Dakota::LeastSq, 539
Dakota::Minimizer, 550
Dakota::NonD, 637
Dakota::Optimizer, 744
find_optimum
Dakota::APPSOptimizer, 314
Dakota::COLINOptimizer, 329
Dakota::JEGAOptimizer, 533
flush
Dakota, 171
Dakota::CommandShell, 337
force_rebuild
Dakota::SurrogateModel, 905
fork_application
Dakota::ForkApplicInterface, 470
fpinit_ASL
main.C, 944
free_iterator
Dakota::Strategy, 875
FSUDesignCompExp
Dakota::FSUDesignCompExp, 475
functionGradients
Dakota::ResponseRep, 826
functionSurfaces
Dakota::ApproximationInterface, 308

G1
Dakota::CONMINOptimizer, 347
G2
Dakota::CONMINOptimizer, 347
GaussProcApproximation
Dakota::GaussProcApproximation, 483
get_approx
Dakota::Approximation, 302
get_best_sample
Dakota::EffGlobalMinimizer, 455
get_confidence_intervals
Dakota::LeastSq, 540
get_constraints
Dakota::Constraints, 360
get_cwd
Dakota, 171
get_db
Dakota::ProblemDescDB, 796
get_hessian
Dakota::SurfpackApproximation, 883
get_interface
Dakota::Interface, 512
get_iterator
Dakota::Iterator, 525
get_model
Dakota::Model, 588
get_npath
Dakota, 171
get_parameter_sets

Dakota::NonDSampling, 729
get_strategy
Dakota::Strategy, 876
get_variables
Dakota::Variables, 932
GetBestMOSolutions
Dakota::JEGAOptimizer, 532
GetBestSolutions
Dakota::JEGAOptimizer, 532
GetBestSOSolutions
Dakota::JEGAOptimizer, 532
GetDescription
Dakota::JEGAOptimizer::Evaluator, 463
getdist
Dakota, 172
GetLongOpt
Dakota::GetLongOpt, 487
GetName
Dakota::JEGAOptimizer::Evaluator, 463
GetNumberLinearConstraints
Dakota::JEGAOptimizer::Evaluator, 462
GetNumberNonLinearConstraints
Dakota::JEGAOptimizer::Evaluator, 461
getRmax
Dakota, 172
gnewton_set_recast
Dakota::Minimizer, 551
GPmodel_apply
Dakota::GaussProcApproximation, 483

hard_convergence_check
Dakota::SurrBasedLocalMinimizer, 892
herbie
Dakota::DirectApplicInterface, 439
herbie1D
Dakota::DirectApplicInterface, 439
hom_constraint_eval
Dakota::SurrBasedLocalMinimizer, 894
hom_objective_eval
Dakota::SurrBasedLocalMinimizer, 894

IC
Dakota::CONMINOptimizer, 348
id_vars_exact_compare
Dakota, 173
increment_expansion
Dakota::NonDPolynomialChaos, 710
increment_grid_preference
Dakota::NonDCubature, 651
increment_parallel_configuration
Dakota::ParallelLibrary, 766
increment_reference
Dakota::NonDCubature, 651
init_communicators
Dakota::Model, 586
Dakota::ParallelLibrary, 767
init_iterator
Dakota::Strategy, 875
init_iterator_parallelism
Dakota::Strategy, 874
init_mpi_comm
Dakota::ParallelLibrary, 766
init_serial
Dakota::ApplicationInterface, 287
Dakota::Model, 587
initial_points
Dakota::JEGAOptimizer, 534
initial_taylor_series
Dakota::NonDLocalReliability, 701
initialize
Dakota::NonDAdaptImpSampling, 643
Dakota::RecastModel, 812
initialize_class_data
Dakota::NonDLocalReliability, 701
initialize_final_statistics
Dakota::NonD, 637
initialize_graphics
Dakota::Iterator, 523
initialize_grid
Dakota::NonDQuadrature, 714
initialize_level_data
Dakota::NonDLocalReliability, 702
initialize_mpp_search_data
Dakota::NonDLocalReliability, 702
initialize_random_variable_parameters
Dakota::NonD, 638
initialize_random_variable_types
Dakota::NonD, 638
initialize_random_variables
Dakota::NonD, 636, 637
initialize_run
Dakota::Iterator, 521
Dakota::LeastSq, 538
Dakota::Minimizer, 550
Dakota::NonD, 637
Dakota::Optimizer, 744
initialize_scaling
Dakota::Minimizer, 551
initialize_variables_and_constraints

Dakota::APPSOptimizer, 314
instantiate_flag
Dakota::CommandLineHandler, 335
intCntlParmArray
Dakota::DOTOptimizer, 449
Interface
Dakota::Interface, 510, 511
interface
Dakota::Model, 584
interface_id
Dakota::Model, 585
ISC
Dakota::CONMINOptimizer, 348
isReadyForWork
Dakota::APPSEvalMgr, 311
Iterator
Dakota::Iterator, 520, 521

JEGAOptimizer
Dakota::JEGAOptimizer, 529
JEGAOptimizer.C, 939
JEGAOptimizer.H, 940

kw_1
Dakota, 177
kw_10
Dakota, 180
kw_100
Dakota, 206
kw_101
Dakota, 206
kw_102
Dakota, 206
kw_103
Dakota, 208
kw_104
Dakota, 208
kw_105
Dakota, 208
kw_106
Dakota, 209
kw_107
Dakota, 209
kw_108
Dakota, 209
kw_109
Dakota, 209
kw_11
Dakota, 181
kw_110
Dakota, 210
kw_111
Dakota, 210
kw_112
Dakota, 210
kw_113
Dakota, 210
kw_114
Dakota, 211
kw_115
Dakota, 211
kw_116
Dakota, 211
kw_117
Dakota, 211
kw_118
Dakota, 211
kw_119
Dakota, 212
kw_12
Dakota, 181
kw_120
Dakota, 213
kw_121
Dakota, 213
kw_122
Dakota, 213
kw_123
Dakota, 213
kw_124
Dakota, 214
kw_125
Dakota, 214
kw_126
Dakota, 214
kw_127
Dakota, 215
kw_128
Dakota, 215
kw_129
Dakota, 215
kw_13
Dakota, 182
kw_130
Dakota, 215
kw_131
Dakota, 215
kw_132
Dakota, 216

kw_133
Dakota, 217
kw_134
Dakota, 217
kw_135
Dakota, 218
kw_136
Dakota, 219
kw_137
Dakota, 219
kw_138
Dakota, 219
kw_139
Dakota, 220
kw_14
Dakota, 182
kw_140
Dakota, 220
kw_141
Dakota, 220
kw_142
Dakota, 220
kw_143
Dakota, 221
kw_144
Dakota, 221
kw_145
Dakota, 221
kw_146
Dakota, 221
kw_147
Dakota, 222
kw_148
Dakota, 223
kw_149
Dakota, 223
kw_15
Dakota, 182
kw_150
Dakota, 224
kw_151
Dakota, 224
kw_152
Dakota, 224
kw_153
Dakota, 224
kw_154
Dakota, 225
kw_155
Dakota, 226
kw_157
Dakota, 226
kw_158
Dakota, 226
kw_159
Dakota, 227
kw_16
Dakota, 182
kw_160
Dakota, 227
kw_161
Dakota, 227
kw_162
Dakota, 227
kw_163
Dakota, 227
kw_164
Dakota, 228
kw_165
Dakota, 228
kw_166
Dakota, 228
kw_167
Dakota, 228
kw_168
Dakota, 229
kw_169
Dakota, 229
kw_17
Dakota, 183
kw_170
Dakota, 229
kw_171
Dakota, 229
kw_172
Dakota, 230
kw_173
Dakota, 230
kw_174
Dakota, 230
kw_175
Dakota, 230
kw_176
Dakota, 231
kw_177
Dakota, 231
kw_178
Dakota, 232

kw_179
Dakota, 232
kw_18
Dakota, 183
kw_180
Dakota, 232
kw_181
Dakota, 233
kw_182
Dakota, 233
kw_183
Dakota, 233
kw_184
Dakota, 234
kw_185
Dakota, 234
kw_186
Dakota, 235
kw_187
Dakota, 235
kw_188
Dakota, 235
kw_189
Dakota, 235
kw_19
Dakota, 183
kw_190
Dakota, 236
kw_191
Dakota, 236
kw_192
Dakota, 236
kw_193
Dakota, 236
kw_194
Dakota, 237
kw_195
Dakota, 237
kw_196
Dakota, 238
kw_197
Dakota, 238
kw_198
Dakota, 238
kw_199
Dakota, 238
kw_2
Dakota, 177
kw_20
Dakota, 183
kw_200
Dakota, 239
kw_201
Dakota, 239
kw_202
Dakota, 240
kw_203
Dakota, 240
kw_204
Dakota, 240
kw_205
Dakota, 240
kw_206
Dakota, 241
kw_207
Dakota, 241
kw_208
Dakota, 241
kw_209
Dakota, 241
kw_21
Dakota, 183
kw_210
Dakota, 242
kw_211
Dakota, 242
kw_212
Dakota, 243
kw_213
Dakota, 243
kw_214
Dakota, 244
kw_215
Dakota, 244
kw_216
Dakota, 245
kw_217
Dakota, 245
kw_218
Dakota, 245
kw_219
Dakota, 246
kw_22
Dakota, 184
kw_220
Dakota, 246
kw_221
Dakota, 246

kw_222
Dakota, 247
kw_223
Dakota, 247
kw_224
Dakota, 247
kw_225
Dakota, 248
kw_226
Dakota, 248
kw_227
Dakota, 249
kw_228
Dakota, 249
kw_229
Dakota, 249
kw_23
Dakota, 184
kw_230
Dakota, 250
kw_231
Dakota, 250
kw_232
Dakota, 250
kw_233
Dakota, 251
kw_234
Dakota, 251
kw_235
Dakota, 252
kw_236
Dakota, 252
kw_237
Dakota, 252
kw_238
Dakota, 253
kw_239
Dakota, 253
kw_24
Dakota, 184
kw_241
Dakota, 254
kw_242
Dakota, 254
kw_243
Dakota, 254
kw_244
Dakota, 255
kw_245
Dakota, 255
kw_246
Dakota, 255
kw_247
Dakota, 256
kw_248
Dakota, 256
kw_249
Dakota, 256
kw_25
Dakota, 185
kw_250
Dakota, 257
kw_251
Dakota, 257
kw_252
Dakota, 257
kw_253
Dakota, 258
kw_255
Dakota, 258
kw_26
Dakota, 185
kw_27
Dakota, 186
kw_28
Dakota, 186
kw_29
Dakota, 186
kw_3
Dakota, 178
kw_30
Dakota, 186
kw_31
Dakota, 186
kw_32
Dakota, 187
kw_33
Dakota, 187
kw_34
Dakota, 188
kw_35
Dakota, 188
kw_36
Dakota, 188
kw_37
Dakota, 188
kw_38
Dakota, 189

kw_39
Dakota, 190
kw_4
Dakota, 178
kw_40
Dakota, 191
kw_41
Dakota, 191
kw_42
Dakota, 191
kw_43
Dakota, 191
kw_44
Dakota, 192
kw_45
Dakota, 193
kw_46
Dakota, 193
kw_47
Dakota, 193
kw_48
Dakota, 193
kw_49
Dakota, 194
kw_5
Dakota, 178
kw_50
Dakota, 194
kw_51
Dakota, 194
kw_52
Dakota, 195
kw_53
Dakota, 195
kw_54
Dakota, 195
kw_55
Dakota, 195
kw_56
Dakota, 196
kw_57
Dakota, 196
kw_58
Dakota, 196
kw_59
Dakota, 196
kw_6
Dakota, 178
kw_60
Dakota, 196
kw_61
Dakota, 197
kw_62
Dakota, 197
kw_63
Dakota, 197
kw_64
Dakota, 198
kw_65
Dakota, 198
kw_66
Dakota, 198
kw_67
Dakota, 198
kw_68
Dakota, 198
kw_69
Dakota, 199
kw_7
Dakota, 179
kw_70
Dakota, 199
kw_71
Dakota, 199
kw_72
Dakota, 199
kw_73
Dakota, 200
kw_74
Dakota, 200
kw_75
Dakota, 200
kw_76
Dakota, 200
kw_77
Dakota, 201
kw_78
Dakota, 201
kw_79
Dakota, 201
kw_8
Dakota, 179
kw_80
Dakota, 201
kw_81
Dakota, 201
kw_82
Dakota, 202

kw_83
Dakota, 202
kw_84
Dakota, 202
kw_85
Dakota, 202
kw_86
Dakota, 203
kw_87
Dakota, 203
kw_88
Dakota, 203
kw_89
Dakota, 203
kw_9
Dakota, 180
kw_90
Dakota, 203
kw_91
Dakota, 204
kw_92
Dakota, 204
kw_93
Dakota, 204
kw_94
Dakota, 205
kw_95
Dakota, 205
kw_96
Dakota, 205
kw_97
Dakota, 205
kw_98
Dakota, 206
kw_99
Dakota, 206

lagrangian_merit
Dakota::SurrBasedMinimizer, 899
LeastSq
Dakota::LeastSq, 538
library_mode.C, 941
main, 942
model_interface_plugins, 942
my_callback_function, 942
run_dakota_mixed, 941
run_dakota_parse, 941
library_split.C, 943
lin_coeffs_modify_n2s
Dakota::Minimizer, 553
LoadAlgorithmConfig
Dakota::JEGAOptimizer, 530
LoadDakotaResponses
Dakota::JEGAOptimizer, 530
LoadProblemConfig
Dakota::JEGAOptimizer, 531
LoadTheConstraints
Dakota::JEGAOptimizer, 531
LoadTheDesignVariables
Dakota::JEGAOptimizer, 531
LoadTheObjectiveFunctions
Dakota::JEGAOptimizer, 531
LoadTheParameterDatabase
Dakota::JEGAOptimizer, 530
local_eval_concurrency
Dakota::Model, 585
local_eval_synchronization
Dakota::Model, 585
local_objective_recast_retrieve
Dakota::Optimizer, 745
lookup_by_val
Dakota, 174
lower
Dakota::String, 879

main
library_mode.C, 942
main.C, 944
restart_util.C, 945
main.C, 944
fpinit_ASL, 944
main, 944
manage_asv
Dakota::Model, 590
manage_inputs
Dakota::ProblemDescDB, 795
manage_linear_constraints
Dakota::Constraints, 360
manage_outputs_restart
Dakota::ParallelLibrary, 765
MandatoryValue
Dakota::GetLongOpt, 487
map
Dakota::ApplicationInterface, 287
map_domain
Dakota::COLINApplication, 326
MergedConstraints
Dakota::MergedConstraints, 542
MergedVariables
Dakota::MergedVariables, 544

mindist
Dakota, 172
mindistindx
Dakota, 172
minimize_residuals
Dakota::NL2SOLLeastSq, 620
minimize_surrogates
Dakota::SurrBasedLocalMinimizer, 892
Minimizer
Dakota::Minimizer, 550
MixedConstraints
Dakota::MixedConstraints, 556
MixedVariables
Dakota::MixedVariables, 558
Model
Dakota::Model, 582
model_interface_plugins
library_mode.C, 942
modify_n2s
Dakota::Minimizer, 552
modify_s2n
Dakota::Minimizer, 552
MS1
Dakota::CONMINOptimizer, 347
my_callback_function
library_mode.C, 942
my_cp
Dakota, 171

N1
Dakota::CONMINOptimizer, 346
N2
Dakota::CONMINOptimizer, 346
N3
Dakota::CONMINOptimizer, 346
N4
Dakota::CONMINOptimizer, 346
N5
Dakota::CONMINOptimizer, 346
Name
Dakota::JEGAOptimizer::Evaluator, 460
NCSUOptimizer
Dakota::NCSUOptimizer, 600
need_resp_trans_byvars
Dakota::Minimizer, 552
new_dataset
Dakota::Graphics, 491
nlf0_evaluator
Dakota::SNLLOptimizer, 863
nlf1_evaluator
Dakota::SNLLOptimizer, 863
nlf2_evaluator
Dakota::SNLLOptimizer, 863
nlf2_evaluator_gn
Dakota::SNLLLeastSq, 855
NLSSOLLeastSq
Dakota::NLSSOLLeastSq, 628
NonDAdaptImpSampling
Dakota::NonDAdaptImpSampling, 642
NonDBayesCalibration
Dakota::NonDBayesCalibration, 645
NonDCalibration
Dakota::NonDCalibration, 648
NonDCubature
Dakota::NonDCubature, 650
NonDGPMSABayesCalibration
Dakota::NonDGPMSABayesCalibration, 672
NonDIncremLHSSampling
Dakota::NonDIncremLHSSampling, 675
NonDIntegration
Dakota::NonDIntegration, 678, 679
NonDLHSSampling
Dakota::NonDLHSSampling, 688
NonDPolynomialChaos
Dakota::NonDPolynomialChaos, 709, 710
NonDQuadrature
Dakota::NonDQuadrature, 714
NonDQUESOBayesCalibration
Dakota::NonDQUESOBayesCalibration, 718
NonDSampling
Dakota::NonDSampling, 728
NonDSparseGrid
Dakota::NonDSparseGrid, 733
NonDStochCollocation
Dakota::NonDStochCollocation, 736
NPSOLOptimizer
Dakota::NPSOLOptimizer, 739
num_samples
Dakota::DDACEDesignCompExp, 429
Dakota::FSUDesignCompExp, 476
Dakota::Iterator, 523
Dakota::NonDCubature, 651
Dakota::NonDQuadrature, 715
Dakota::NonDSampling, 729
Dakota::NonDSparseGrid, 734
Dakota::PSUADEDesignCompExp, 802
NUMBER_OF_FIELDS
Dakota, 176

objective

Dakota::Minimizer, 553
objective_eval
Dakota::NCSUOptimizer, 601
objective_gradient
Dakota::Minimizer, 554
objective_hessian
Dakota::Minimizer, 554
objective_reduction
Dakota::Optimizer, 745
operator const char ∗
Dakota::String, 879
operator<<
Dakota::BoStream, 322
Dakota::CommandShell, 337
operator>>
Dakota::BiStream, 319
operator=
Dakota::Approximation, 300
Dakota::Constraints, 359
Dakota::Interface, 511
Dakota::Iterator, 521
Dakota::Model, 583
Dakota::ProblemDescDB, 795
Dakota::Strategy, 873
Dakota::Variables, 931
optimizationType
Dakota::CONMINOptimizer, 345
Dakota::DOTOptimizer, 449
Optimizer
Dakota::Optimizer, 743
OptionalValue
Dakota::GetLongOpt, 486
OptType
Dakota::GetLongOpt, 486

pack_parameters_buffer
Dakota::ConcurrentStrategy, 339
Dakota::SequentialHybridStrategy, 835
Dakota::Strategy, 873
pack_results_buffer
Dakota::ConcurrentStrategy, 340
Dakota::SequentialHybridStrategy, 835
Dakota::Strategy, 874
ParallelLibrary
Dakota::ParallelLibrary, 764, 765
ParamResponsePair
Dakota::ParamResponsePair, 770
parse
Dakota::GetLongOpt, 487
parse_inputs
Dakota::ProblemDescDB, 795
path_is_absolute
Dakota, 175
penalty_merit
Dakota::SurrBasedMinimizer, 900
perform_analysis
Dakota, 173
perform_evaluation_impl
Dakota::COLINApplication, 325
PerformIterations
Dakota::JEGAOptimizer::Driver, 452
PMA_constraint_eval
Dakota::NonDReliability, 722
PMA_objective_eval
Dakota::NonDReliability, 722
pop
Dakota::Approximation, 300
Dakota::PecosApproximation, 784
pop_approximation
Dakota::ApproximationInterface, 307
post_process
Dakota::ProblemDescDB, 796
post_run
Dakota::COLINOptimizer, 330
Dakota::DDACEDesignCompExp, 429
Dakota::FSUDesignCompExp, 476
Dakota::Iterator, 522
Dakota::LeastSq, 538
Dakota::Optimizer, 744
Dakota::ParamStudy, 776
Dakota::PSUADEDesignCompExp, 802
Dakota::SNLLLeastSq, 855
pre_output
Dakota::Analyzer, 277
pre_run
Dakota::DDACEDesignCompExp, 429
Dakota::FSUDesignCompExp, 476
Dakota::Iterator, 522
Dakota::ParamStudy, 775
Dakota::PSUADEDesignCompExp, 802
primary_resp_recast
Dakota::LeastSq, 539
Dakota::Optimizer, 745
print_restart
Dakota, 174
print_restart_tabular
Dakota, 174
print_results
Dakota::Analyzer, 277

Dakota::Iterator, 523
Dakota::LeastSq, 539
Dakota::NonDInterval, 682
Dakota::Optimizer, 744
Dakota::PStudyDACE, 798
Dakota::RichExtrapVerification, 829
Dakota::SurrBasedMinimizer, 898
Dakota::Verification, 934
print_sobol_indices
Dakota::Analyzer, 279
printControl
Dakota::CONMINOptimizer, 345
Dakota::DOTOptimizer, 449
probability
Dakota::NonDLocalReliability, 704
ProblemDescDB
Dakota::ProblemDescDB, 794, 795
PRPMultiIndexCache
Dakota, 170
PRPMultiIndexQueue
Dakota, 170
PSUADEDesignCompExp
Dakota::PSUADEDesignCompExp, 802
python_convert_int
Dakota::DirectApplicInterface, 440

quantify_uncertainty
Dakota::NonDGPMSABayesCalibration, 672
Dakota::NonDIncremLHSSampling, 675
Dakota::NonDLHSSampling, 689
Dakota::NonDQUESOBayesCalibration, 718

rawResponseMap
Dakota::Interface, 512
read
Dakota::ParamResponsePair, 770
Dakota::ResponseRep, 823–825
read_annotated
Dakota::ResponseRep, 823
read_neutral
Dakota, 175
read_tabular
Dakota::MergedVariables, 544
Dakota::MixedVariables, 558
Dakota::ResponseRep, 824
read_variables_responses
Dakota::Analyzer, 278
realCntlParmArray
Dakota::DOTOptimizer, 449
rebuild
Dakota::Approximation, 300
Dakota::PecosApproximation, 784
rebuild_approximation
Dakota::ApproximationInterface, 307
RecastModel
Dakota::RecastModel, 811
RecordResponses
Dakota::JEGAOptimizer::Evaluator, 461
recv
Dakota::APPSEvalMgr, 311
rel_change_rv
Dakota, 171
reliability
Dakota::NonDLocalReliability, 705
repair_restart
Dakota, 175
requestVector
Dakota::ActiveSet, 267
reset
Dakota::ResponseRep, 826
reset_inactive
Dakota::ResponseRep, 826
reshape
Dakota::Constraints, 359
Dakota::MergedConstraints, 542
Dakota::MixedConstraints, 556
Dakota::ResponseRep, 826
resize_response_results_array
Dakota::JEGAOptimizer, 533
resize_variables_results_array
Dakota::JEGAOptimizer, 533
resolve_inputs
Dakota::ParallelLibrary, 767
resolve_samples_symbols
Dakota::DDACEDesignCompExp, 429
Response
Dakota::Response, 818
response_mapping
Dakota::NestedModel, 609
response_modify_n2s
Dakota::Minimizer, 553
response_modify_s2n
Dakota::Minimizer, 553
responseMode
Dakota::SurrogateModel, 905
ResponseRep
Dakota::ResponseRep, 822, 823
restart_util.C, 945
main, 945

restore
Dakota::Approximation, 301
Dakota::PecosApproximation, 784
restore_approximation
Dakota::ApproximationInterface, 308
restore_available
Dakota::ApproximationInterface, 308
retrieve
Dakota::GetLongOpt, 487
returns_multiple_points
Dakota::COLINOptimizer, 330
Dakota::JEGAOptimizer, 534
RIA_constraint_eval
Dakota::NonDReliability, 722
RIA_objective_eval
Dakota::NonDReliability, 722
run
Dakota::Iterator, 522
Dakota::LeastSq, 538
Dakota::NonD, 637
Dakota::Optimizer, 744
Dakota::PStudyDACE, 798
Dakota::SurrBasedMinimizer, 898
Dakota::Verification, 934
run_dakota_data
Dakota, 173
run_dakota_mixed
library_mode.C, 941
run_dakota_parse
library_mode.C, 941
run_iterator
Dakota::Iterator, 524
Dakota::Strategy, 875
run_sequential
Dakota::SequentialHybridStrategy, 836
run_sequential_adaptive
Dakota::SequentialHybridStrategy, 836

S
Dakota::CONMINOptimizer, 347
sampling_reset
Dakota::NonDCubature, 651
Dakota::NonDQuadrature, 714
Dakota::NonDSampling, 729
Dakota::NonDSparseGrid, 734
SCAL
Dakota::CONMINOptimizer, 348
schedule_iterators
Dakota::Strategy, 876
secondary_resp_recast
Dakota::Minimizer, 552
self_schedule_analyses
Dakota::ApplicationInterface, 288
self_schedule_evaluations
Dakota::ApplicationInterface, 290
self_schedule_iterators
Dakota::Strategy, 876
send_data_using_get
Dakota::TrackerHTTP, 919
send_data_using_post
Dakota::TrackerHTTP, 919
separable_combine
Dakota::DirectApplicInterface, 440
SeparateVariables
Dakota::JEGAOptimizer::Evaluator, 461
serve_analyses_asynch
Dakota::ForkApplicInterface, 471
serve_analyses_synch
Dakota::ApplicationInterface, 289
serve_evaluations
Dakota::ApplicationInterface, 288
serve_evaluations_asynch
Dakota::ApplicationInterface, 292
serve_evaluations_peer
Dakota::ApplicationInterface, 293
serve_evaluations_synch
Dakota::ApplicationInterface, 292
serve_iterators
Dakota::Strategy, 876
set_apps_parameters
Dakota::APPSOptimizer, 314
set_compare
Dakota, 173
set_problem
Dakota::COLINApplication, 324
set_rng
Dakota::COLINOptimizer, 330
set_solver_parameters
Dakota::COLINOptimizer, 330
set_u_to_x_mapping
Dakota::NonD, 639
SharedVariablesDataRep
Dakota::SharedVariablesDataRep, 844
shubert1D
Dakota::DirectApplicInterface, 439
SIM, 263
SIM::ParallelDirectApplicInterface, 749
SIM::SerialDirectApplicInterface, 838
slmap

Dakota, 176
smooth_herbie
Dakota::DirectApplicInterface, 439
smooth_herbie1D
Dakota::DirectApplicInterface, 439
SNLLOptimizer
Dakota::SNLLOptimizer, 862
solver_setup
Dakota::COLINOptimizer, 330
spawn_analysis
Dakota::SysCallAnalysisCode, 908
spawn_evaluation
Dakota::SysCallAnalysisCode, 907
spawn_evaluation_impl
Dakota::COLINApplication, 324
spawn_input_filter
Dakota::SysCallAnalysisCode, 908
spawn_output_filter
Dakota::SysCallAnalysisCode, 908
specify_outputs_restart
Dakota::ParallelLibrary, 765
split_filenames
Dakota::ParallelLibrary, 767
start_dakota_heartbeat
Dakota, 171
start_grid_computing
Dakota, 172
static_schedule_evaluations
Dakota::ApplicationInterface, 290
stop_evaluation_servers
Dakota::ApplicationInterface, 288
stop_grid_computing
Dakota, 172
Strategy
Dakota::Strategy, 873
submit
Dakota::APPSEvalMgr, 311
subModel
Dakota::NestedModel, 611
subordinate_iterator
Dakota::Model, 583
subordinate_model
Dakota::Model, 583
subordinate_models
Dakota::Model, 586
SurfpackApproximation
Dakota::SurfpackApproximation, 883
surrogate_model
Dakota::Model, 584
surrogates_to_surf_data
Dakota::SurfpackApproximation, 884
synch
Dakota::ApplicationInterface, 287
synch_nowait
Dakota::ApplicationInterface, 288
synchronize_derivatives
Dakota::Model, 589
synchronous_local_analyses
Dakota::ForkApplicInterface, 471
synchronous_local_evaluations
Dakota::ApplicationInterface, 291

ToDoubleMatrix
Dakota::JEGAOptimizer, 532
tr_ratio_check
Dakota::SurrBasedLocalMinimizer, 893
trendOrder
Dakota::GaussProcApproximation, 483
truth_model
Dakota::Model, 584

unpack_parameters_buffer
Dakota::ConcurrentStrategy, 340
Dakota::SequentialHybridStrategy, 835
Dakota::Strategy, 874
unpack_results_buffer
Dakota::ConcurrentStrategy, 340
Dakota::SequentialHybridStrategy, 836
Dakota::Strategy, 874
update
Dakota::ResponseRep, 825
update_actual_model
Dakota::DataFitSurrModel, 372
update_approximation
Dakota::ApproximationInterface, 306
Dakota::DataFitSurrModel, 369, 370
update_augmented_lagrange_multipliers
Dakota::SurrBasedMinimizer, 899
update_filter
Dakota::SurrBasedMinimizer, 899
update_from_actual_model
Dakota::DataFitSurrModel, 373
update_from_sub_model
Dakota::RecastModel, 812
update_from_subordinate_model
Dakota::Model, 584
update_lagrange_multipliers
Dakota::SurrBasedMinimizer, 899
update_level_data

Dakota::NonDLocalReliability, 703
update_mpp_search_data
Dakota::NonDLocalReliability, 703
update_partial
Dakota::ResponseRep, 825
update_penalty
Dakota::SurrBasedLocalMinimizer, 893
update_pma_reliability_level
Dakota::NonDLocalReliability, 704
update_quasi_hessians
Dakota::Model, 590
update_response
Dakota::Model, 589
upper
Dakota::String, 879
usage
Dakota::GetLongOpt, 488
useDerivs
Dakota::NonDExpansion, 659

Valueless
Dakota::GetLongOpt, 486
var_mp_bgen
Dakota, 260
var_mp_bgen_audi
Dakota, 261
var_mp_bgen_audr
Dakota, 261
var_mp_bgen_dis
Dakota, 261
var_mp_bgen_eu
Dakota, 261
var_mp_bndchk
Dakota, 262
var_mp_ibndchk
Dakota, 262
Variables
Dakota::Variables, 930, 931
variables_recast
Dakota::Minimizer, 551
variance_based_decomp
Dakota::Analyzer, 278
vars_u_to_x_mapping
Dakota::NonD, 638
Vlch
Dakota, 259
VLS
Dakota, 260
volumetric_quality
Dakota::PStudyDACE, 799

write
Dakota::ParamResponsePair, 770
Dakota::ResponseRep, 823–825
write_annotated
Dakota::ResponseRep, 824
write_tabular
Dakota::ResponseRep, 824

SAND2010-2184
Unlimited Release
December 2009
Updated November 30, 2011

DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis

Version 5.2 Reference Manual

Brian M. Adams, Keith R. Dalbey, Michael S. Eldred, Laura P. Swiler
Optimization and Uncertainty Quantification Department

William J. Bohnhoff
Radiation Transport Department

John P. Eddy
System Readiness and Sustainment Technologies Department

Dena M. Vigil
Multiphysics Simulation Technologies Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Patricia D. Hough, Sophia Lefantzi
Quantitative Modeling and Analysis Department

Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551

Abstract

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible and extensible interface between simulation codes and iterative analysis methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods; uncertainty quantification with sampling, reliability, and stochastic expansion methods; parameter estimation with nonlinear least squares methods; and sensitivity/variance analysis with design of experiments and parameter study methods. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for design and performance analysis of computational models on high performance computers.

This report serves as a reference manual for the commands specification for the DAKOTA software, providing input overviews, option descriptions, and example specifications.

Contents

1 DAKOTA Reference Manual **11**

1.1 Overview 11

1.2 Input Specification Reference 11

1.3 Additional Resources 12

2 Introduction **13**

3 Overview **15**

4 NIDR Input Specification File **17**

5 Common Specification Mistakes **19**

6 Sample dakota.in Files **21**

6.1 Sample 1: Optimization 21

6.2 Sample 2: Least Squares (Calibration) 22

6.3 Sample 3: Nondeterministic Analysis 22

6.4 Sample 4: Parameter Study 23

6.5 Sample 5: Hybrid Strategy 24

7 Tabular descriptions **27**

8 Strategy Commands **29**

9 Strategy Description **31**

10 Strategy Specification **33**

11 Strategy Independent Controls **35**

12 Hybrid Minimization Commands **37**

13 Multistart Iteration Commands **39**

14 Pareto Set Optimization Commands **41**

15 Single Method Commands **43**

16 Method Commands **45**

17 Method Description **47**

18 Method Specification **49**

19 Method Independent Controls **51**

20 Optimization Methods **59**

20.1 DOT Methods 59
20.2 NPSOL Method 60
20.3 NLPQL Methods 62
20.4 CONMIN Methods 62
20.5 OPT++ Methods 62
20.6 Asynchronous Parallel Pattern Search (APPS) 66
20.7 SCOLIB Methods 67
20.8 NCSU Methods 81
20.9 JEGA Methods 82

21 Least Squares Methods **93**

21.1 NL2SOL Method 93
21.2 NLSSOL Method 94
21.3 Gauss-Newton Method 96

22 Surrogate-Based Minimization Methods **97**

22.1 Surrogate-Based Local Method 97
22.2 Surrogate-Based Global Method 99
22.3 Efficient Global Method 102

23 Uncertainty Quantification Methods **105**

23.1 Aleatory Uncertainty Quantification Methods . . . 108
23.2 Epistemic Uncertainty Quantification Methods . . . 127

24 Nondeterministic Calibration Methods 133
24.1 Bayesian Calibration Methods . . . 133

25 Solution Verification Methods 135
25.1 Richardson Extrapolation . . . 135

26 Design of Computer Experiments Methods 137
26.1 DDACE . . . 137
26.2 FSUDace . . . 138
26.3 PSUADE . . . 141

27 Parameter Study Methods 143
27.1 Vector parameter study . . . 143
27.2 List parameter study . . . 144
27.3 Centered parameter study . . . 145
27.4 Multidimensional parameter study . . . 145

28 Model Commands 147

29 Model Description 149

30 Model Specification 151

31 Model Independent Controls 153

32 Single Model Controls 155

33 Surrogate Model Controls 157
33.1 Global approximations . . . 158
33.2 Multipoint approximations . . . 162
33.3 Local approximations . . . 164
33.4 Hierarchical approximations . . . 164

34 Nested Model Controls 167

35 Variables Commands 171

36 Variables Description 173

37 Variables Specification 177

38 Variables Set Identifier 179

39 Design Variables 181

39.1 Continuous Design Variables . . . 181
39.2 Discrete Design Range Variables . . . 182
39.3 Discrete Design Integer Set Variables . . . 182
39.4 Discrete Design Real Set Variables . . . 183

40 Aleatory Uncertain Variables 185

40.1 Normal Distribution . . . 185
40.2 Lognormal Distribution . . . 186
40.3 Uniform Distribution . . . 187
40.4 Loguniform Distribution . . . 188
40.5 Triangular Distribution . . . 188
40.6 Exponential Distribution . . . 189
40.7 Beta Distribution . . . 190
40.8 Gamma Distribution . . . 190
40.9 Gumbel Distribution . . . 191
40.10 Frechet Distribution . . . 191
40.11 Weibull Distribution . . . 192
40.12 Histogram Bin Distribution . . . 192
40.13 Poisson Distribution . . . 194
40.14 Binomial Distribution . . . 195
40.15 Negative Binomial Distribution . . . 195
40.16 Geometric Distribution . . . 196
40.17 Hypergeometric Distribution . . . 196
40.18 Histogram Point Distribution . . . 197
40.19 Correlations . . . 198

41 Epistemic Uncertain Variables 201

41.1 Interval Uncertain Variable . . . 201

42 State Variables **203**
42.1 Continuous State Variables . . . 203
42.2 Discrete State Range Variables . . . 204
42.3 Discrete State Integer Set Variables . . . 204
42.4 Discrete State Real Set Variables . . . 204

43 Interface Commands **207**

44 Interface Description **209**

45 Interface Specification **211**

46 Interface Independent Controls **213**

47 Algebraic mappings **217**

48 Simulation interfaces **219**
48.1 System call interface . . . 221
48.2 Fork interface . . . 222
48.3 Direct function interface . . . 222
48.4 Grid interface . . . 224

49 Responses Commands **225**

50 Responses Description **227**

51 Responses Specification **229**

52 Responses Set Identifier **231**

53 Response Labels **233**

54 Function Specification **235**
54.1 Objective and constraint functions (optimization data set) . . . 235
54.2 Calibration terms and constraint functions (least squares data set) . . . 236
54.3 Response functions (generic data set) . . . 239

55 Gradient Specification **241**
55.1 No gradients . . . 241

55.2 Numerical gradients 241
55.3 Analytic gradients 242
55.4 Mixed gradients 243

56 Hessian Specification **245**
56.1 No Hessians 245
56.2 Numerical Hessians 245
56.3 Quasi Hessians 246
56.4 Analytic Hessians 247
56.5 Mixed Hessians 247

57 Bibliography **249**

58 File Index **255**

59 File Documentation **257**
59.1 dakota.input.summary File Reference 257

Chapter 1

DAKOTA Reference Manual

Author:

Brian M. Adams, William J. Bohnhoff, Keith R. Dalbey, John P. Eddy, Michael S. Eldred, Patricia D. Hough, Sophia Lefantzi, Laura P. Swiler, Dena M. Vigil

1.1 Overview

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible, extensible interface between analysis codes and iteration methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods, uncertainty quantification with sampling, reliability, stochastic expansion, and interval estimation methods, parameter estimation with nonlinear least squares methods, and sensitivity/variance analysis with design of experiments and parameter study capabilities. (Solution verification and Bayesian approaches are also in development.) These capabilities may be used on their own or as components within advanced algorithms such as surrogate-based optimization, mixed integer nonlinear programming, mixed aleatory-epistemic uncertainty quantification, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible problem-solving environment for design and performance analysis of computational models on high performance computers.

The Reference Manual focuses on documentation of the various input commands for DAKOTA. It follows closely the structure of dakota.input.summary, the master input specification summary. For information on software structure, refer to the `Developers Manual`, and for a tour of DAKOTA features and capabilities, refer to the Users Manual [Adams et al., 2010].

1.2 Input Specification Reference

In DAKOTA, the *strategy* creates and manages *iterators* and *models*. A model contains a set of *variables*, an *interface*, and a set of *responses*, and the iterator operates on the model to map the variables into responses using the interface. In a DAKOTA input file, the user specifies these components through strategy, method, model, variables, interface, and responses keyword specifications. The Reference Manual closely follows this structure, with introductory material followed by detailed documentation of the strategy, method, model, variables, interface,

and responses keyword specifications:

Introduction

Strategy Commands

Method Commands

Model Commands

Variables Commands

Interface Commands

Responses Commands

1.3 Additional Resources

A bibliography for the Reference Manual is provided in:

Bibliography

Project web pages are maintained at http://dakota.sandia.gov/ including documentation at http://dakota.sandia.gov/documentation.html and a list of publications at http://dakota.sandia.gov/publications.html

Chapter 2

Introduction

Chapter 3

Overview

In DAKOTA, a *strategy* creates and manages *iterators* and *models*. A model, generally speaking, contains a set of *variables*, an *interface*, and a set of *responses*. An iterator repeatedly operates on the model to map the variables into responses using the interface. Each of these six pieces (strategy, method, model, variables, interface, and responses) are separate specifications in the user's input file, and as a whole, determine the study to be performed during an execution of the DAKOTA software. A DAKOTA execution is limited to a single strategy, which may invoke multiple methods. Furthermore, each method may have its own model, consisting of (generally speaking) its own variables, interface, and set of responses. Thus, there may be multiple specifications of the method, model, variables, interface, and responses sections.

DAKOTA input is most commonly specified through a text file, whose syntax is governed by the New Input Deck Reader (NIDR) parsing system [Gay, 2008], in conjunction with the dakota.input.nspec file which describes allowable inputs. A more concise version of the input specification file, dakota.input.summary, offers a quick reference to keywords and data values from which a particular input file (e.g., `dakota.in`) may be derived. This automatically derived shortened form omits implementation details not needed in a quick reference.

This Reference Manual focuses on the details of allowable specifications in a file input to the DAKOTA program. Related details on the name and location of the `dakota` program, command line inputs, and execution syntax are provided in the Users Manual [Adams et al., 2010].

Chapter 4

NIDR Input Specification File

DAKOTA input is governed by the NIDR input specification file. This file (dakota.input.nspec) is used by a code generator to create parsing system components that are compiled into the DAKOTA executable (refer to **Instructions for Modifying DAKOTA's Input Specification** for additional information). Therefore, dakota.input.nspec and its derived summary, dakota.input.summary, are the definitive source for input syntax, capability options, and optional and required capability sub-parameters for any given DAKOTA version. Beginning users may find dakota.input.summary overwhelming or confusing and will likely derive more benefit from adapting example input files to a particular problem. However, advanced users can master the many input specification possibilities by understanding the structure of the input specification file.

Refer to the dakota.input.summary file for current input specifications. From this file listing, it can be seen that the main structure of the strategy specification consists several required group specifications separated by logical OR's (indicated by |): either hybrid OR multi-start OR pareto set OR single method. The method keyword is the most lengthy specification; however, its structure is again relatively simple. The structure is a set of optional method-independent settings followed by a long list of possible methods appearing as required group specifications (containing a variety of method-dependent settings) separated by OR's. The model keyword reflects a structure of three required group specifications separated by OR's. Within the surrogate model type, the type of approximation must be specified with either a global OR multipoint OR local OR hierarchical required group specification. The structure of the variables keyword is that of optional group specifications for continuous and discrete design variables, a number of different uncertain variable distribution types, and continuous and discrete state variables. Each of these specifications can either appear or not appear as a group. Next, the interface keyword allows the specification of either algebraic mappings, simulation-based analysis driver mappings, or both. Within the analysis drivers specification, a system OR fork OR direct OR grid group specification must be selected. Finally, within the responses keyword, the primary structure is the required specification of the function set (either optimization functions OR calibration functions OR generic response functions), followed by the required specification of the gradients (either none OR numerical OR analytic OR mixed) and the required specification of the Hessians (either none OR numerical OR quasi OR analytic OR mixed). Refer to Strategy Commands, Method Commands, Model Commands, Variables Commands, Interface Commands, and Responses Commands for detailed information on the keywords and their various optional and required specifications. And for additional details on NIDR specification logic and rules, refer to [Gay, 2008].

Some keywords, such as those providing bounds on variables, have an associated list of values. When the same value should be repeated several times in a row, you can use the notation `N*value` instead of repeating the value N times. For example, in Sample 2: Least Squares (Calibration) below,

```
lower_bounds    -2.0   -2.0
```

```
  upper_bounds     2.0    2.0
```

could also be written

```
  lower_bounds    2*-2.0
  upper_bounds    2 * 2.0
```

(with optional spaces around the ∗). Another possible abbreviation is for sequences: L:S:U (with optional spaces around the :) is expanded to L L+S L+2∗S ... U, and L:U (with no second colon) is treated as L:1:U. For example, in one of the test examples distributed with DAKOTA (test case 2 of `test/dakota_uq_textbook_sop_-lhs.in`),

```
histogram_point = 2
  abscissas     = 50. 60. 70. 80. 90.
                  30. 40. 50. 60. 70.
  counts        = 10  20  30  20  10
                  10  20  30  20  10
```

could also be written

```
histogram_point = 2
  abscissas     = 50 : 10 : 90
                  30 : 10 : 70
  counts        = 10:10:30  20  10
                  10:10:30  20  10
```

Count and sequence abbreviations can be used together. For example

```
response_levels =
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

can be abbreviated

```
response_levels =
2*0.0:0.1:1.0
```

Chapter 5

Common Specification Mistakes

Spelling mistakes and omission of required parameters are the most common errors. Some causes of errors are more obscure:

- Documentation of new capability sometimes lags its availability in source and executables, especially nightly releases. When parsing errors occur that the documentation cannot explain, reference to the particular input specification used in building the executable, which is installed alongside the executable, will often resolve the errors.

- If you want to compare results with those obtained using an earlier version of DAKOTA (prior to 4.1), your input file for the earlier version must use backslashes to indicate continuation lines for DAKOTA keywords. For example, rather than

  ```
  # Comment about the following "responses" keyword...
  responses,
          objective_functions = 1
          # Comment within keyword "responses"
          analytic_gradients
  # Another comment within keyword "responses"
          no_hessians
  ```

  you would need to write

  ```
  # Comment about the following "responses" keyword...
  responses,                                          \
          objective_functions = 1                     \
          # Comment within keyword "responses"        \
          analytic_gradients                          \
  # Another comment within keyword "responses"        \
          no_hessians
  ```

  with no white space (blanks or tabs) after the \ character.

In most cases, the NIDR system provides error messages that help the user isolate errors in DAKOTA input files.

Chapter 6

Sample dakota.in Files

A DAKOTA input file is a collection of fields from the dakota.input.summary file that describe the problem to be solved by DAKOTA. Several examples follow.

6.1 Sample 1: Optimization

The following sample input file shows single-method optimization of the Textbook Example using DOT's modified method of feasible directions. A similar file is available as `Dakota/examples/tutorial/dakota_textbook.in`.

```
strategy,
        single_method

method,
#       DOT performs better, but may not be available
        dot_mmfd,
#       conmin_mfd,
          max_iterations = 50,
          convergence_tolerance = 1e-4

variables,
        continuous_design = 2
          initial_point    0.9    1.1
          upper_bounds     5.8    2.9
          lower_bounds     0.5   -2.9
          descriptors      'x1'   'x2'

interface,
        direct
          analysis_driver =       'text_book'

responses,
        objective_functions = 1
        nonlinear_inequality_constraints = 2
        numerical_gradients
          method_source dakota
          interval_type central
          fd_gradient_step_size = 1.e-4
        no_hessians
```

6.2 Sample 2: Least Squares (Calibration)

The following sample input file shows a nonlinear least squares (calibration) solution of the Rosenbrock Example using the NL2SOL method. A similar file is available as `Dakota/examples/tutorial/dakota_rosenbrock_ls.in`.

```
strategy,
        single_method

method,
        nl2sol
          max_iterations = 50
          convergence_tolerance = 1e-4

model,
        single

variables,
        continuous_design = 2
          initial_point   -1.2    1.0
          lower_bounds    -2.0   -2.0
          upper_bounds     2.0    2.0
          descriptor       'x1'   'x2'

interface,
        fork
          analysis_driver = 'rosenbrock'

responses,
        calibration_terms = 2
        analytic_gradients
        no_hessians
```

6.3 Sample 3: Nondeterministic Analysis

The following sample input file shows Latin Hypercube Monte Carlo sampling using the Textbook Example. A similar file is available as `Dakota/test/dakota_uq_textbook_lhs.in`.

```
strategy,
        single_method

method,
        sampling,
          samples = 100 seed = 1
          complementary distribution
          response_levels = 3.6e+11 4.0e+11 4.4e+11
                            6.0e+04 6.5e+04 7.0e+04
                            3.5e+05 4.0e+05 4.5e+05
          sample_type lhs

variables,
        normal_uncertain = 2
          means             =  248.89, 593.33
          std_deviations    =   12.4,   29.7
          descriptors       =  'TF1n'  'TF2n'
        uniform_uncertain = 2
          lower_bounds      =  199.3,  474.63
```

```
          upper_bounds      =  298.5,  712.
          descriptors       =  'TF1u'  'TF2u'
        weibull_uncertain = 2
          alphas            =   12.,    30.
          betas             =  250.,   590.
          descriptors       =  'TF1w'  'TF2w'
        histogram_bin_uncertain = 2
          num_pairs   =  3         4
          abscissas   =  5  8 10  .1  .2  .3  .4
          counts      = 17 21  0  12  24  12   0
          descriptors = 'TF1h'  'TF2h'
        histogram_point_uncertain = 1
          num_pairs   = 2
          abscissas   = 3 4
          counts      = 1 1
          descriptors = 'TF3h'

interface,
        fork asynch evaluation_concurrency = 5
          analysis_driver = 'text_book'

responses,
        response_functions = 3
        no_gradients
        no_hessians
```

6.4 Sample 4: Parameter Study

The following sample input file shows a 1-D vector parameter study using the Textbook Example. It makes use of the default strategy and model specifications (`single_method` and `single`, respectively), so they can be omitted. A similar file is available in the test directory as `Dakota/examples/tutorial/dakota_rosenbrock_vector.in`.

```
strategy,
        single_method

method,
        vector_parameter_study
          final_point = 1.1  1.3
          num_steps = 10

model,
        single

variables,
        continuous_design = 2
          initial_point   -0.3      0.2
          descriptors      'x1'     "x2"

interface,
        direct
          analysis_driver = 'rosenbrock'

responses,
        objective_functions = 1
        no_gradients
        no_hessians
```

6.5 Sample 5: Hybrid Strategy

The following sample input file shows a hybrid strategy using three methods. It employs a genetic algorithm, pattern search, and full Newton gradient-based optimization in succession to solve the Textbook Example. A similar file is available as `Dakota/examples/methods/dakota_hybrid.in`.

```
strategy,
        graphics
        hybrid sequential
          method_list = 'GA' 'PS' 'NLP'

method,
        id_method = 'GA'
        model_pointer = 'M1'
        coliny_ea
          seed = 1234
          population_size = 10
          verbose output

method,
        id_method = 'PS'
        model_pointer = 'M1'
        coliny_pattern_search stochastic
          seed = 1234
          initial_delta = 0.1
          threshold_delta = 1.e-4
          solution_accuracy = 1.e-10
          exploratory_moves basic_pattern
          verbose output

method,
        id_method = 'PS2'
        model_pointer = 'M1'
        max_function_evaluations = 10
        coliny_pattern_search stochastic
          seed = 1234
          initial_delta = 0.1
          threshold_delta = 1.e-4
          solution_accuracy = 1.e-10
          exploratory_moves basic_pattern
          verbose output

method,
        id_method = 'NLP'
        model_pointer = 'M2'
        optpp_newton
          gradient_tolerance = 1.e-12
          convergence_tolerance = 1.e-15
          verbose output

model,
        id_model = 'M1'
        single
          variables_pointer = 'V1'
          interface_pointer = 'I1'
          responses_pointer = 'R1'

model,
        id_model = 'M2'
        single
          variables_pointer = 'V1'
```

```
          interface_pointer = 'I1'
          responses_pointer = 'R2'

variables,
        id_variables = 'V1'
        continuous_design = 2
          initial_point    0.6    0.7
          upper_bounds     5.8    2.9
          lower_bounds     0.5   -2.9
          descriptors      'x1'   'x2'

interface,
        id_interface = 'I1'
        direct
          analysis_driver=  'text_book'

responses,
        id_responses = 'R1'
        objective_functions = 1
        no_gradients
        no_hessians

responses,
        id_responses = 'R2'
        objective_functions = 1
        analytic_gradients
        analytic_hessians
```

Additional example input files, as well as the corresponding output and graphics, are provided in the Tutorial chapter of the Users Manual [Adams et al., 2010].

Chapter 7

Tabular descriptions

In the following discussion of keyword specifications, tabular formats (Tables 4.1 through 9.10) are used to present a short description of the specification, the keyword used in the specification, the type of data associated with the keyword, the status of the specification (required, optional, required group, or optional group), and the default for an optional specification.

It can be difficult to capture in a simple tabular format the complex relationships that can occur when specifications are nested within multiple groupings. For example, in the model keyword, the `actual_model_pointer` specification is a required specification within the `multipoint` and `local` required group specifications, which are separated from each other and from other required group specifications (`global` and `hierarchical`) by logical OR's. The selection between the `global`, `multipoint`, `local`, or `hierarchical` required groups is contained within another required group specification (`surrogate`), which is separated from the `single` and `nested` required group specifications by logical OR's. Rather than unnecessarily proliferate the number of tables in attempting to capture all of these inter-relationships, a balance is sought, since some inter-relationships are more easily discussed in the associated text. The general structure of the following sections is to present the outermost specification groups first (e.g., `single`, `surrogate`, or `nested` in Table 6.1), followed by lower levels of group specifications (e.g., `global`, `multipoint`, `local`, or `hierarchical` surrogates in Table 6.3), followed by the components of each group (e.g., Tables 6.4 through 6.8) in succession.

Chapter 8

Strategy Commands

Chapter 9

Strategy Description

The strategy section in a DAKOTA input file specifies the top level technique which will govern the management of iterators and models in the solution of the problem of interest. Four strategies currently exist: `hybrid`, `multi_start`, `pareto_set`, and `single_method`. These algorithms are implemented within the **Strategy** class hierarchy in the **CollaborativeHybridStrategy**, **EmbeddedHybridStrategy**, **SequentialHybridStrategy**, **ConcurrentStrategy**, and **SingleMethodStrategy** classes. For each of the strategies, a brief algorithm description is given below. Additional information on the algorithm logic is available in the Users Manual [Adams et al., 2010].

In a hybrid minimization strategy (`hybrid`), a set of methods synergistically seek an optimal design. The relationships among the methods are categorized as collaborative, embedded, or sequential. The goal in each case is to exploit the strengths of different optimization and nonlinear least squares algorithms through different stages of the minimization process. Global/local hybrids (e.g., genetic algorithms combined with nonlinear programming) are a common example in which the desire for identification of a global optimum is balanced with the need for efficient navigation to a local optimum.

In the multi-start iteration strategy (`multi_start`), a series of iterator runs are performed for different values of parameters in the model. A common use is for multi-start optimization (i.e., different local optimization runs from different starting points for the design variables), but the concept and the code are more general. An important feature is that these iterator runs may be performed concurrently.

In the pareto set optimization strategy (`pareto_set`), a series of optimization or least squares calibration runs are performed for different weightings applied to multiple objective functions. This set of optimal solutions defines a "Pareto set," which is useful for investigating design trade-offs between competing objectives. Again, these optimizations can be performed concurrently, similar to the multi-start strategy discussed above. The code is similar enough to the `multi_start` technique that both strategies are implemented in the same **ConcurrentStrategy** class.

Lastly, the `single_method` strategy is the default strategy in that it does not provide control over multiple iterators or multiple models. Rather, it provides the means for simple execution of a single iterator on a single model. It is most commonly used when specifying other strategy-independent controls, e.g., tabular output, where the existence of the `strategy` block mandates selection of a strategy.

Each of the strategy specifications identifies one or more method pointers (e.g., `method_list`, `method_pointer`) to identify the iterators that will be used in the strategy. These method pointers are strings that correspond to the `id_method` identifier strings from the method specifications (see Method Independent Controls). These string identifiers (e.g., 'NLP1') should *not* be confused with method selections (e.g., `dot_-`

mmfd). Each of the method specifications identified in this manner has the responsibility for identifying corresponding model specifications (using model_pointer from Method Independent Controls), which in turn identify the variables, interface, and responses specifications (using variables_pointer, interface_pointer, and responses_pointer from Model Commands) that are used to build the model used by the iterator. If one of these specifications does not provide an optional pointer, then that component will be constructed using the last specification parsed. In addition to method pointers, a variety of graphics options (e.g., tabular_graphics_data), iterator concurrency controls (e.g., iterator_servers), and strategy data (e.g., starting_points) can be specified.

Specification of a strategy block in an input file is optional, with single_method being the default strategy. If no strategy is specified or if single_method is specified without its optional method_pointer specification, then the default behavior is to employ the last method, variables, interface, and responses specifications parsed. This default behavior is most appropriate if only one specification is present for method, variables, interface, and responses, since there is no ambiguity in this case.

Example specifications for each of the strategies follow. A hybrid example is:

```
strategy,
hybrid sequential
  method_list = 'GA', 'PS', 'NLP'
```

A multi_start example specification is:

```
strategy,
multi_start
  method_pointer = 'NLP1'
  random_starts = 10
```

A pareto_set example specification is:

```
strategy,
pareto_set
  method_pointer = 'NLP1'
          random_weight_sets = 10
```

And finally, a single_method example specification is:

```
strategy,
single_method
  method_pointer = 'NLP1'
```

Chapter 10

Strategy Specification

The strategy specification has the following structure:

```
strategy,
<strategy independent controls>
<strategy selection>
  <strategy dependent controls>
```

where `<strategy selection>` is one of the following: `hybrid`, `multi_start`, `pareto_set`, or `single_method`.

The `<strategy independent controls>` are those controls which are valid for a variety of strategies. Unlike the Method Independent Controls, which can be abstractions with slightly different implementations from one method to the next, the implementations of each of the strategy independent controls are consistent for all strategies that use them. The `<strategy dependent controls>` are those controls which are only meaningful for a specific strategy. Referring to dakota.input.summary, the strategy independent controls are those controls defined externally from and prior to the strategy selection blocks. They are all optional. The strategy selection blocks are all required group specifications separated by logical OR's (`hybrid` OR `multi_start` OR `pareto_set` OR `single_method`). Thus, one and only one strategy selection must be provided. The strategy dependent controls are those controls defined within the strategy selection blocks. Defaults for strategy independent and strategy dependent controls are defined in **DataStrategy**. The following sections provide additional detail on the strategy independent controls followed by the strategy selections and their corresponding strategy dependent controls.

Chapter 11

Strategy Independent Controls

The strategy independent controls are summarized in Table 4.1, including graphics, output data, and iterator scheduling controls. The `graphics` flag activates a 2D graphics window containing history plots for the variables and response functions in the study. This window is updated in an event loop with approximately a 2 second cycle time. The `tabular_graphics_data` flag activates file tabulation of the same variables and response function history data that gets passed to graphics windows with use of the `graphics` flag. The `tabular_graphics_file` specification optionally specifies a name to use for this file (`dakota_tabular.dat` is the default). Within the file, the variables and response functions appear as columns and each function evaluation provides a new table row. This capability is most useful for post-processing of DAKOTA results with third-party graphics tools such as MATLAB, Tecplot, etc. There is no dependence between the `graphics` flag and the `tabular_graphics_data` flag; they may be used independently or concurrently. Numeric output precision (as passed to the C++ control std::setprecision) can be controlled with `output_precision` with an upper limit of 16 enforced at run-time. When this control is not specified, most DAKOTA output will default to a precision of 10, though filesystem interfaces and pre-run output use higher precision for accuracy and better results reproducibility.

The `iterator_servers`, `iterator_self_scheduling`, and `iterator_static_scheduling` specifications provide manual overrides for the number of concurrent iterator partitions and the scheduling policy for concurrent iterator jobs. These settings are normally determined automatically in the parallel configuration routines (see **ParallelLibrary**) but can be overridden with user inputs if desired. The `graphics`, `tabular_graphics_data`, and `tabular_graphics_file` specifications are valid for all strategies. However, the `iterator_servers`, `iterator_self_scheduling`, and `iterator_static_scheduling` overrides are only useful inputs for those strategies supporting concurrency in iterators, i.e., `multi_start` and `pareto_set`.

Description	Keyword	Associated Data	Status	Default
Graphics flag	graphics	none	Optional	no graphics
Tabulation of graphics data	tabular_graphics_data	none	Optional group	no data tabulation
File name for tabular graphics data	tabular_graphics_file	string	Optional	dakota_tabular.dat
Numeric output precision	output_precision	integer	Optional	10 (for most output)
Number of iterator servers	iterator_servers	integer	Optional	no override of auto configure
Self-scheduling of iterator jobs	iterator_self_scheduling	none	Optional	no override of auto configure
Static scheduling of iterator jobs	iterator_static_scheduling	none	Optional	no override of auto configure

Table 11.1: Specification detail for strategy independent controls

Chapter 12

Hybrid Minimization Commands

The hybrid minimization strategy has `sequential`,`embedded`, and `collaborative` approaches (see the Users Manual [Adams et al., 2010] for more information on the algorithms employed). In the sequential approach, the best solutions are transferred from one method to the next through a specified sequence. In the embedded approach, a tightly-coupled hybrid is employed in which a subordinate local method provides periodic refinements to a top-level global method. And in the collaborative approach, multiple methods work together and share solutions while executing concurrently.

In the `sequential` approach, a list of method strings supplied with the `method_list` specification specifies the identity and sequence of iterators to be used. Any number of iterators may be specified. In the sequential approach, method switching is managed through the separate convergence controls of each method. The number of solutions transferred between methods is specified by the particular method through its `final_solutions` method control. For example, if one sets up a two-level strategy with a first method that generates multiple solutions such as a genetic algorithm, followed by a second method that is initialized only at a single point such as a gradient-based algorithm, it is possible to take the multiple solutions generated by the first method and create several instances of the second method, each one with a different initial starting point. The logic governing the transfer of multiple solutions between methods is as follows: if one solution is returned from method A, then one solution is transferred to method B. If multiple solutions are returned from method A, and method B can accept multiple solutions as input (for example, as a genetic algorithm population), then one instance of method B is initialized with multiple solutions. If multiple solutions are returned from method A but method B only can accept one initial starting point, then method B is run several times, each one with a separate starting point from the results of method A. Table 4.2 summarizes the sequential hybrid strategy inputs.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Hybrid strategy	`hybrid`	none	Required group (1 of 4 selections)	N/A
Sequential hybrid	`sequential`	none	Required group (1 of 3 selections)	N/A
List of methods	`method_list`	list of strings	Required	N/A

Table 12.1: Specification detail for sequential hybrid strategies

In the `embedded` approach, global and local method strings supplied with the `global_method_pointer` and `local_method_pointer` specifications identify the two methods to be used. The `local_search_probability` setting is an optional specification for supplying the probability (between 0.0 and 1.0) of em-

ploying local search to improve estimates within the global search. Table 4.3 summarizes the embedded hybrid strategy inputs.

Description	Keyword	Associated Data	Status	Default
Hybrid strategy	`hybrid`	none	Required group (1 of 4 selections)	N/A
Embedded hybrid	`embedded`	none	Required group (1 of 3 selections)	N/A
Pointer to the global method specification	`global_-method_-pointer`	string	Required	N/A
Pointer to the local method specification	`local_-method_-pointer`	string	Required	N/A
Probability of executing local searches	`local_-search_-probability`	real	Optional	0.1

Table 12.2: Specification detail for embedded hybrid strategies

In the `collaborative` approach, a list of method strings supplied with the `method_list` specification specifies the pool of iterators to be used. Any number of iterators may be specified. The method collaboration logic follows that of either the Agent-Based Optimization or HOPSPACK codes and is currently under development. Table 4.4 summarizes the collaborative hybrid strategy inputs.

Description	Keyword	Associated Data	Status	Default
Hybrid strategy	`hybrid`	none	Required group (1 of 4 selections)	N/A
Collaborative hybrid	`collaborative`	none	Required group (1 of 3 selections)	N/A
List of methods	`method_list`	list of strings	Required	N/A

Table 12.3: Specification detail for collaborative hybrid strategies

Chapter 13

Multistart Iteration Commands

The `multi_start` strategy must specify an iterator using `method_pointer`. This iterator is responsible for completing a series of iterative analyses from a set of different starting points. These starting points can be specified as follows: (1) using `random_starts`, for which the specified number of starting points are selected randomly within the variable bounds, (2) using `starting_points`, in which the starting values are provided in a list, or (3) using both `random_starts` and `starting_points`, for which the combined set of points will be used. In aggregate, at least one starting point must be specified. The most common example of a multi-start strategy is multi-start optimization, in which a series of optimizations are performed from different starting values for the design variables. This can be an effective approach for problems with multiple minima. Table 4.5 summarizes the multi-start strategy inputs.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Multi-start iteration strategy	`multi_start`	none	Required group (1 of 4 selections)	N/A
Method pointer	`method_-pointer`	string	Required	N/A
Number of random starting points	`random_-starts`	integer	Optional group	no random starting points
Seed for random starting points	`seed`	integer	Optional	system-generated seed
List of user-specified starting points	`starting_-points`	list of reals	Optional	no user-specified starting points

Table 13.1: Specification detail for multi-start strategies

Chapter 14

Pareto Set Optimization Commands

The `pareto_set` strategy must specify an optimization or least squares calibration method using `method_pointer`. This minimizer is responsible for computing a set of optimal solutions from a set of response weightings (multi-objective weights or least squares term weights). These weightings can be specified as follows: (1) using `random_weight_sets`, in which case weightings are selected randomly within [0,1] bounds, (2) using `weight_sets`, in which the weighting sets are specified in a list, or (3) using both `random_weight_sets` and `weight_sets`, for which the combined set of weights will be used. In aggregate, at least one set of weights must be specified. The set of optimal solutions is called the "pareto set," which can provide valuable design trade-off information when there are competing objectives. Table 4.6 summarizes the pareto set strategy inputs.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Pareto set optimization strategy	`pareto_set`	none	Required group (1 of 4 selections)	N/A
Optimization method pointer	`method_pointer`	string	Required	N/A
Number of random weighting sets	`random_weight_sets`	integer	Optional	no random weighting sets
Seed for random weighting sets	`seed`	integer	Optional	system-generated seed
List of user-specified weighting sets	`weight_sets`	list of reals	Optional	no user-specified weighting sets

Table 14.1: Specification detail for pareto set strategies

Chapter 15

Single Method Commands

The single method strategy is the default if no strategy specification is included in a user input file. It may also be specified using the `single_method` keyword within a strategy specification. An optional `method_pointer` specification may be used to point to a particular method specification. If `method_pointer` is not used, then the last method specification parsed will be used as the iterator. Table 4.7 summarizes the single method strategy inputs.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Single method strategy	`single_method`	string	Required group (1 of 4 selections)	N/A
Method pointer	`method_pointer`	string	Optional	use of last method parsed

Table 15.1: Specification detail for single method strategies

Chapter 16

Method Commands

Chapter 17

Method Description

The method section in a DAKOTA input file specifies the name and controls of an iterator. The terms "method" and "iterator" can be used interchangeably, although method often refers to an input specification whereas iterator usually refers to an object within the **Iterator** hierarchy. A method specification, then, is used to select an iterator from the iterator hierarchy, which includes optimization, uncertainty quantification, least squares, design of experiments, and parameter study iterators (see the Users Manual [Adams et al., 2010] for more information on these iterator branches). This iterator may be used alone or in combination with other iterators as dictated by the strategy specification (refer to Strategy Commands for strategy command syntax and to the Users Manual [Adams et al., 2010] for strategy algorithm descriptions).

Several examples follow. The first example shows a minimal specification for an optimization method.

```
method,
dot_sqp
```

This example uses all of the defaults for this method.

A more sophisticated example would be

```
method,
id_method = 'NLP1'
model_pointer = 'M1'
dot_sqp
  max_iterations = 50
  convergence_tolerance = 1e-4
  output verbose
  optimization_type minimize
```

This example demonstrates the use of identifiers and pointers (see Method Independent Controls) as well as some method independent and method dependent controls for the sequential quadratic programming (SQP) algorithm from the DOT library. The `max_iterations`, `convergence_tolerance`, and `output` settings are method independent controls, in that they are defined for a variety of methods (see DOT method independent controls for DOT usage of these controls). The `optimization_type` control is a method dependent control, in that it is only meaningful for DOT methods (see DOT method dependent controls).

The next example shows a specification for a least squares method.

```
method,
optpp_g_newton
```

```
  max_iterations = 10
  convergence_tolerance = 1.e-8
  search_method trust_region
  gradient_tolerance = 1.e-6
```

Some of the same method independent controls are present along with a new set of method dependent controls (`search_method` and `gradient_tolerance`) which are only meaningful for OPT++ methods (see OPT++ method dependent controls).

The next example shows a specification for a nondeterministic method with several method dependent controls (refer to Nondeterministic sampling method).

```
method,
sampling
  samples = 100 seed = 12345
  sample_type lhs
  response_levels = 1000. 500.
```

The last example shows a specification for a parameter study method where, again, each of the controls are method dependent (refer to Vector parameter study).

```
method,
vector_parameter_study
  step_vector = 1. 1. 1.
  num_steps = 10
```

Chapter 18

Method Specification

As alluded to in the examples above, the method specification has the following structure:

```
method,
<method independent controls>
<method selection>
  <method dependent controls>
```

where <method selection> is for example one of the following: dot_frcg, dot_mmfd, dot_bfgs, dot_slp, dot_sqp, conmin_frcg, conmin_mfd, npsol_sqp, nlssol_sqp, nlpql_sqp, nl2sol, nonlinear_cg, optpp_cg, optpp_q_newton, optpp_fd_newton, optpp_g_newton, optpp_newton, optpp_pds, asynch_pattern_search, coliny_cobyla, coliny_direct, coliny_pattern_search, coliny_solis_wets, coliny_ea, moga, soga, ncsu_direct, dl_solver, surrogate_based_local, surrogate_based_global, efficient_global, polynomial_chaos, stoch_collocation, sampling, importance, local_reliability, global_reliability, local_evidence, global_evidence, local_interval_est, global_interval_est, bayes_calibration, dace, fsu_quasi_mc, fsu_cvt, psuade_moat, vector_parameter_study, list_parameter_study, centered_parameter_study, or multidim_parameter_study.

The <method independent controls> are those controls which are valid for a variety of methods. In some cases, these controls are abstractions which may have slightly different implementations from one method to the next. The <method dependent controls> are those controls which are only meaningful for a specific method or library. Referring to dakota.input.summary, the method independent controls are those controls defined externally from and prior to the method selection blocks. They are all optional. The method selection blocks are all required group specifications separated by logical OR's. The method dependent controls are those controls defined within the method selection blocks. Defaults for method independent and method dependent controls are defined in **DataMethod**. The following sections provide additional detail on the method independent controls followed by the method selections and their corresponding method dependent controls.

Chapter 19

Method Independent Controls

The method independent controls include a method identifier string, a model type specification with pointers to variables, interface, and responses specifications, a speculative gradient selection, an output verbosity control, maximum iteration and function evaluation limits, constraint and convergence tolerance specifications, a scaling selection, and a set of linear inequality and equality constraint specifications. While each of these controls is not valid for every method, the controls are valid for enough methods that it was reasonable to pull them out of the method dependent blocks and consolidate the specifications.

The method identifier string is supplied with `id_method` and is used to provide a unique identifier string for use with strategy specifications (refer to Strategy Description). It is appropriate to omit a method identifier string if only one method is included in the input file and `single_method` is the selected strategy (all other strategies require one or more method pointers), since the single method to use is unambiguous in this case.

The model pointer string is specified with `model_pointer` and is used to identify the model used to perform function evaluations for the method. If a model pointer string is specified and no corresponding id is available, DAKOTA will exit with an error message. If no model pointer string is specified, then the last model specification parsed will be used. If no model pointer string is specified and no model specification is provided by the user, then a default model specification is used (similar to the default strategy specification, see Strategy Description). This default model specification is of type `single` with no `variables_pointer`, `interface_pointer`, or `responses_pointer` (see Single Model Controls). It is appropriate to omit a model specification whenever the relationships are unambiguous due to the presence of single variables, interface, and responses specifications.

When performing gradient-based optimization in parallel, `speculative` gradients can be selected to address the load imbalance that can occur between gradient evaluation and line search phases. In a typical gradient-based optimization, the line search phase consists primarily of evaluating the objective function and any constraints at a trial point, and then testing the trial point for a sufficient decrease in the objective function value and/or constraint violation. If a sufficient decrease is not observed, then one or more additional trial points may be attempted sequentially. However, if the trial point is accepted then the line search phase is complete and the gradient evaluation phase begins. By speculating that the gradient information associated with a given line search trial point will be used later, additional coarse grained parallelism can be introduced by computing the gradient information (either by finite difference or analytically) in parallel, at the same time as the line search phase trial-point function values. This balances the total amount of computation to be performed at each design point and allows for efficient utilization of multiple processors. While the total amount of work performed will generally increase (since some speculative gradients will not be used when a trial point is rejected in the line search phase), the run time will usually decrease (since gradient evaluations needed at the start of each new optimization cycle were already performed in parallel during the line search phase). Refer to [Byrd et al., 1998] for additional details.

The speculative specification is implemented for the gradient-based optimizers in the DOT, CONMIN, and OPT++ libraries, and it can be used with dakota numerical or analytic gradient selections in the responses specification (refer to Gradient Specification for information on these specifications). It should not be selected with vendor numerical gradients since vendor internal finite difference algorithms have not been modified for this purpose. In full-Newton approaches, the Hessian is also computed speculatively. NPSOL and NLSSOL do not support speculative gradients, as their gradient-based line search in user-supplied gradient mode (dakota numerical or analytic gradients) is a superior approach for load-balanced parallel execution.

Output verbosity control is specified with `output` followed by `silent`, `quiet`, `verbose` or `debug`. If there is no user specification for output verbosity, then the default setting is `normal`. This gives a total of five output levels to manage the volume of data that is returned to the user during the course of a study, ranging from full run annotation plus internal debug diagnostics (`debug`) to the bare minimum of output containing little more than the total number of simulations performed and the final solution (`silent`). Output verbosity is observed within the **Iterator** (algorithm verbosity), **Model** (synchronize/fd_gradients verbosity), **Interface** (map/synch verbosity), **Approximation** (global data fit coefficient reporting),and **AnalysisCode** (file operation reporting) class hierarchies; however, not all of these software components observe the full granularity of verbosity settings. Specific mappings are as follows:

- `output silent` (i.e., really quiet): silent iterators, silent model, silent interface, quiet approximation, quiet file operations
- `output quiet`: quiet iterators, quiet model, quiet interface, quiet approximation, quiet file operations
- `output normal`: normal iterators, normal model, normal interface, quiet approximation, quiet file operations
- `output verbose`: verbose iterators, normal model, verbose interface, verbose approximation, verbose file operations
- `output debug` (i.e., really verbose): debug iterators, normal model, debug interface, verbose approximation, verbose file operations

Note that iterators and interfaces utilize the full granularity in verbosity, whereas models, approximations, and file operations do not. With respect to iterator verbosity, different iterators implement this control in slightly different ways (as described below in the method independent controls descriptions for each iterator), however the meaning is consistent. For models, interfaces, approximations, and file operations, `quiet` suppresses parameter and response set reporting and `silent` further suppresses function evaluation headers and scheduling output. Similarly, `verbose` adds file management, approximation evaluation, and global approximation coefficient details, and `debug` further adds diagnostics from nonblocking schedulers.

The `constraint_tolerance` specification determines the maximum allowable value of infeasibility that any constraint in an optimization problem may possess and still be considered to be satisfied. It is specified as a positive real value. If a constraint function is greater than this value then it is considered to be violated by the optimization algorithm. This specification gives some control over how tightly the constraints will be satisfied at convergence of the algorithm. However, if the value is set too small the algorithm may terminate with one or more constraints being violated. This specification is currently meaningful for the NPSOL, NLSSOL, DOT and CONMIN constrained optimizers (refer to DOT method independent controls and NPSOL method independent controls).

The `convergence_tolerance` specification provides a real value for controlling the termination of iteration. In most cases, it is a relative convergence tolerance for the objective function; i.e., if the change in the objective function between successive iterations divided by the previous objective function is less than the amount

specified by convergence_tolerance, then this convergence criterion is satisfied on the current iteration. Since no progress may be made on one iteration followed by significant progress on a subsequent iteration, some libraries require that the convergence tolerance be satisfied on two or more consecutive iterations prior to termination of iteration. This control is used with optimization and least squares iterators (DOT, CONMIN, NPSOL, NLSSOL, OPT++, and SCOLIB) and is not used within the uncertainty quantification, design of experiments, or parameter study iterator branches. Refer to DOT method independent controls, NPSOL method independent controls, OPT++ method independent controls, and SCOLIB method independent controls for specific interpretations of the `convergence_tolerance` specification.

The `max_iterations` and `max_function_evaluations` controls provide integer limits for the maximum number of iterations and maximum number of function evaluations, respectively. The difference between an iteration and a function evaluation is that a function evaluation involves a single parameter to response mapping through an interface, whereas an iteration involves a complete cycle of computation within the iterator. Thus, an iteration generally involves multiple function evaluations (e.g., an iteration contains descent direction and line search computations in gradient-based optimization, population and multiple offset evaluations in nongradient-based optimization, etc.). The `max_function_evaluations` control is not currently used within the uncertainty quantification, design of experiments, and parameter study iterator branches, and in the case of gradient-based methods, does not currently capture function evaluations that occur as part of the `method_source dakota` finite difference routine (since these additional evaluations are intentionally isolated from the iterators).

The `final_solutions` controls the number of final solutions returned by the iterator as the best solutions. For most optimizers, this is one, but some optimizers can produce multiple solutions (e.g. genetic algorithms). In the case of analyzers such as sampling methods, if one specifies 100 samples (for example) but also specifies `final_solutions` = 5, the five best solutions (in order of lowest response function value) are returned. When using a `hybrid` strategy, the number of final solutions dictates how many solutions are passed from one method to another.

Continuous design variable, function, and constraint scaling can be turned on for optimizers and least squares minimizers by providing the `scaling` keyword. Discrete variable scaling is not supported. When scaling is enabled, variables, functions, gradients, Hessians, etc., are transformed such that the optimizer iterates in scaled variable space, whereas evaluations of the computational model as specified in the interface are performed on the original problem scale. Therefore using scaling does not require rewriting the interface to the simulation code. The user may specify no, one, or a vector of scaling type strings through each of the `scale_types` (see Variables Commands); `objective_function_scale_types`, `calibration_term_scale_types`, `nonlinear_inequality_scale_types`, `nonlinear_equality_scale_types` (see Function Specification); `linear_inequality_scale_types`, and `linear_equality_scale_types` (see Method Independent Controls below) specifications. Valid options for types include `'none'` (default), `'value'`, `'auto'`, or `'log'`, for no, characteristic value, automatic, or logarithmic scaling, respectively, although not all types are valid for scaling all entities (see the references for details). If a single string is specified using any of these keywords it will apply to each component of the relevant vector, e.g., `scale_types = 'value'` will enable characteristic value scaling for each continuous design variable. The user may specify no, one, or a vector of nonzero characteristic scale values through each of the `scales` (see Variables Commands); `objective_function_scales`, `calibration_term_scales`, `nonlinear_inequality_scales`, `nonlinear_equality_scales` (see Function Specification); `linear_inequality_scales`, and `linear_equality_scales` (see Method Independent Controls below) specifications. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'` and `'log'`. If a single value is specified using any of these keywords it will apply to each component of the relevant vector, e.g., `scales = 3.0` will apply a characteristic scaling value of `3.0` to each continuous design variable. When the `scaling` keyword is omitted, all `_scale_types` and `*_scales` specifications are ignored in the method, variables, and responses sections.

When scaling is enabled, the following procedures determine the transformations used to scale each component of a variables or response vector. A warning is issued if scaling would result in division by a value smaller in magnitude than `1.0e10*DBL_MIN`. User-provided values violating this lower bound are accepted unaltered, whereas for automatically calculated scaling, the lower bound is enforced.

- None (`'none'`): no scaling performed (`*_scales` ignored) on this component
- Characteristic value (`'value'`): the corresponding quantity is scaled by the (required) characteristic value provided in the `*_scales` specification. If the scale value is negative, the sense of inequalities are changed accordingly.
- Automatic (`'auto'`): First, any characteristic values from the optional `*_scales` specification are applied. Then, automatic scaling will be attempted according to the following scheme:
  - two-sided bounds scaled into the interval [0,1];
  - one-sided bound or targets are scaled by the characteristic value, moving the bound or target to 1 and changing the sense of inequalities where necessary;
  - no bounds or targets: no automatic scaling possible, therefore no scaling for this component

  Automatic scaling is not available for objective functions nor calibration terms since they lack bound constraints. Futher, when automatically scaled, linear constraints are scaled by characteristic values only, not affinely scaled into [0,1].
- Logarithmic (`'log'`): First, any characteristic values from the optional `*_scales` specification are applied. Then, logarithm base 10 scaling is applied. Logarithmic scaling is not available for linear constraints. Further, when continuous design variables are log scaled, linear constraints are not allowed.

Table 5.1 provides the specification detail for the method independent controls involving identifiers, pointers, tolerances, limits, output verbosity, speculative gradients, and scaling.

Linear inequality constraints can be supplied with the `linear_inequality_constraint_matrix`, `linear_inequality_lower_bounds`, and `linear_inequality_upper_bounds` specifications, and linear equality constraints can be supplied with the `linear_equality_constraint_matrix` and `linear_equality_targets` specifications. In the inequality case, the constraint matrix provides coefficients for the variables and the lower and upper bounds provide constraint limits for the following two-sided formulation:

$$a_l \leq Ax \leq a_u$$

As with nonlinear inequality constraints (see Objective and constraint functions (optimization data set)), the default linear inequality constraint bounds are selected so that one-sided inequalities of the form

$$Ax \leq 0.0$$

result when there are no user bounds specifications (this provides backwards compatibility with previous DAKOTA versions). In a user bounds specification, any upper bound values greater than `+bigRealBoundSize` (1.e+30, as defined in **Minimizer**) are treated as +infinity and any lower bound values less than `-bigRealBoundSize` are treated as -infinity. This feature is commonly used to drop one of the bounds in order to specify a 1-sided constraint (just as the default lower bounds drop out since `-DBL_MAX` < `-bigRealBoundSize`). In the equality case, the constraint matrix again provides coefficients for the variables and the targets provide the equality constraint right hand sides:

$$Ax = a_t$$

Description	Keyword	Associated Data	Status	Default
Method set identifier	id_method	string	Optional	strategy use of last method parsed
Model pointer	model_-pointer	string	Optional	method use of last model parsed (or use of default model if none parsed)
Speculative gradients and Hessians	speculative	none	Optional	no speculation
Output verbosity	output	silent | quiet | normal | verbose | debug	Optional	normal
Maximum iterations	max_-iterations	integer	Optional	100 (exceptions: fsu_cvt, local_-reliability: 25; global_-{reliability, interval_est, evidence} / efficient_-global: 25∗n)
Maximum function evaluations	max_-function_-evaluations	integer	Optional	1000
Constraint tolerance	constraint_-tolerance	real	Optional	Library default
Convergence tolerance	convergence_-tolerance	real	Optional	1.e-4
Final solutions	final_-solutions	integer	Optional	1
Scaling flag	scaling	none	Optional	no scaling

Table 19.1: Specification detail for the method independent controls: identifiers, pointers, tolerances, limits, output verbosity, speculative gradients, and scaling

and the defaults for the equality constraint targets enforce a value of `0.` for each constraint

$$Ax = 0.0$$

Currently, DOT, CONMIN, NPSOL, NLSSOL, and OPT++ all support specialized handling of linear constraints (either directly through the algorithm itself or indirectly through the DAKOTA wrapper). SCOLIB optimizers will support linear constraints in future releases. Linear constraints need not be computed by the user's interface on every function evaluation; rather the coefficients, bounds, and targets of the linear constraints can be provided at start up, allowing the optimizers to track the linear constraints internally. It is important to recognize that linear constraints are those constraints that are linear in the *design* variables, e.g.:

$$0.0 \leq 3x_1 - 4x_2 + 2x_3 \leq 15.0$$

$$x_1 + x_2 + x_3 \geq 2.0$$

$$x_1 + x_2 - x_3 = 1.0$$

which is not to be confused with something like

$$s(X) - s_{fail} \leq 0.0$$

where the constraint is linear in a response quantity, but may be a nonlinear implicit function of the design variables. For the three linear constraints above, the specification would appear as:

```
linear_inequality_constraint_matrix =  3.0 -4.0  2.0
                                       1.0  1.0  1.0
linear_inequality_lower_bounds =       0.0  2.0
linear_inequality_upper_bounds =      15.0  1.e+50
linear_equality_constraint_matrix =    1.0  1.0 -1.0
linear_equality_targets =              1.0
```

where the `1.e+50` is a dummy upper bound value which defines a 1-sided inequality since it is greater than `bigRealBoundSize`. The constraint matrix specifications list the coefficients of the first constraint followed by the coefficients of the second constraint, and so on. They are divided into individual constraints based on the number of design variables, and can be broken onto multiple lines for readability as shown above.

The `linear_inequality_scale_types` and `linear_equality_scale_types` specifications provide strings specifying the scaling type for each linear inequality or equality constraint, respectively, in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `linear_*_scale_types` may be selected from `'none'`, `'value'`, or `'auto'` to select no, characteristic value, or automatic scaling, respectively. If a single string is specified it will apply to each constraint component. Each entry in `linear_inequality_scales` or `linear_equality_scales` may be a user-specified nonzero characteristic value to be used in scaling each constraint. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'`. If a single real value is specified it will apply to all components of the constraint. Scaling for linear constraints is applied *after* any continuous variable scaling. For example, for variable scaling on continuous design variables x:

$$\tilde{x}^j = \frac{x^j - x_O^j}{x_M^j}$$

we have the following system for linear inequality constraints

$$a_L \leq A_i x \leq a_U$$

$$a_L \leq A_i \left(\text{diag}(x_M)\tilde{x} + x_O\right) \leq a_U$$
$$a_L - A_i x_O \leq A_i \text{diag}(x_M)\tilde{x} \leq a_U - A_i x_O$$
$$\tilde{a}_L \leq \tilde{A}_i \tilde{x} \leq \tilde{a}_U$$

and user-specified or automatically computed scaling multipliers are appplied to this final transformed system, which accounts for continuous design variable scaling. When automatic scaling is in use for linear constraints they are linearly scaled by a computed characteristic value, but not affinely to [0,1].

Table 5.2 provides the specification detail for the method independent controls involving linear constraints.

Description	Keyword	Associated Data	Status	Default
Linear inequality coefficient matrix	`linear_inequality_constraint_matrix`	list of reals	Optional	no linear inequality constraints
Linear inequality lower bounds	`linear_inequality_lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Linear inequality upper bounds	`linear_inequality_upper_bounds`	list of reals	Optional	vector values = `0.`
Linear inequality scaling types	`linear_inequality_scale_types`	list of strings	Optional	vector values = `'none'`
Linear inequality scales	`linear_inequality_scales`	list of reals	Optional	vector values = `1.` (no scaling)
Linear equality coefficient matrix	`linear_equality_constraint_matrix`	list of reals	Optional	no linear equality constraints
Linear equality targets	`linear_equality_targets`	list of reals	Optional	vector values = `0.`
Linear equality scaling types	`linear_equality_scale_types`	list of strings	Optional	vector values = `'none'`
Linear equality scales	`linear_equality_scales`	list of reals	Optional	vector values = `1.` (no scaling)

Table 19.2: Specification detail for the method independent controls: linear inequality and equality constraints

Chapter 20

Optimization Methods

The DAKOTA project started as an toolbox for optimization methods, and has accumulated a broad variety of gradient-based and nongradient-based optimizers from the DOT, NPSOL, NLPQL, CONMIN, OPT++, APPS, SCOLIB, NCSU, and JEGA packages. These capabilities are described below.

20.1 DOT Methods

The DOT library [Vanderplaats Research and Development, 1995] contains nonlinear programming optimizers, specifically the Broyden-Fletcher-Goldfarb-Shanno (DAKOTA's `dot_bfgs` method) and Fletcher-Reeves conjugate gradient (DAKOTA's `dot_frcg` method) methods for unconstrained optimization, and the modified method of feasible directions (DAKOTA's `dot_mmfd` method), sequential linear programming (DAKOTA's `dot_slp` method), and sequential quadratic programming (DAKOTA's `dot_sqp` method) methods for constrained optimization. DAKOTA provides access to the DOT library through the **DOTOptimizer** class.

We here provide a caution regarding `dot_frcg`. In DOT Version 4.20, we have noticed inconsistent behavior of this algorithm across different versions of Linux. Our best assessment is that it is due to different treatments of uninitialized variables. As we do not know the intention of the code authors and maintaining DOT source code is outside of the DAKOTA project scope, we have not made nor are we recommending any code changes to address this. However, all users who use `dot_frcg` in DOT Version 4.20 should be aware that results may not be reliable.

20.1.1 DOT method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during a DOT optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. This convergence criterion must be satisfied for two consecutive iterations before DOT will terminate. The `constraint_tolerance` specification defines how tightly constraint functions are to be satisfied at convergence. The default value for DOT constrained optimizers is 0.003. Extremely small values for constraint_tolerance may not be attainable. The output verbosity specification controls the amount of information generated by DOT: the `silent` and `quiet` settings result in header information, final results, and objective function, constraint, and parameter information on each iteration; whereas the `verbose` and `debug`

settings add additional information on gradients, search direction, one-dimensional search results, and parameter scaling factors. DOT contains no parallel algorithms which can directly take advantage of concurrent evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, then the finite difference function evaluations can be performed concurrently (using any of the parallel modes described in the Users Manual [Adams et al., 2010]). In addition, if `speculative` is specified, then gradients (`dakota numerical` or `analytic` gradients) will be computed on each line search evaluation in order to balance the load and lower the total run time in parallel optimization studies. Lastly, specialized handling of linear constraints is supported with DOT; linear constraint coefficients, bounds, and targets can be provided to DOT at start-up and tracked internally. Specification detail for these method independent controls is provided in Tables 5.1 through 5.2.

20.1.2 DOT method dependent controls

DOT's only method dependent control is `optimization_type` which may be either `minimize` or `maximize`. DOT provides the only set of methods within DAKOTA which support this control; to convert a maximization problem into the minimization formulation assumed by other methods, simply change the sign on the objective function (i.e., multiply by -1). Table 5.3 provides the specification detail for the DOT methods and their method dependent controls.

Description	Keyword	Associated Data	Status	Default
Optimization type	`optimization_type`	`minimize` | `maximize`	Optional group	`minimize`

Table 20.1: Specification detail for the DOT methods

20.2 NPSOL Method

The NPSOL library [Gill et al., 1986] contains a sequential quadratic programming (SQP) implementation (the `npsol_sqp` method). SQP is a nonlinear programming optimizer for constrained minimization. DAKOTA provides access to the NPSOL library through the **NPSOLOptimizer** class.

20.2.1 NPSOL method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major SQP iterations and the number of function evaluations that can be performed during an NPSOL optimization. The `convergence_tolerance` control defines NPSOL's internal optimality tolerance which is used in evaluating if an iterate satisfies the first-order Kuhn-Tucker conditions for a minimum. The magnitude of `convergence_tolerance` approximately specifies the number of significant digits of accuracy desired in the final objective function (e.g., `convergence_tolerance = 1.e-6` will result in approximately six digits of accuracy in the final objective function). The `constraint_tolerance` control defines how tightly the constraint functions are satisfied at convergence. The default value is dependent upon the machine precision of the platform in use, but is typically on the order of `1.e-8` for double precision computations. Extremely small values for `constraint_tolerance` may not be attainable. The `output` verbosity setting controls the amount of information generated at each major SQP iteration: the `silent` and `quiet` settings result in only one line of diagnostic output for each major iteration and print the final optimization solution, whereas the `verbose`

and `debug` settings add additional information on the objective function, constraints, and variables at each major iteration.

NPSOL is not a parallel algorithm and cannot directly take advantage of concurrent evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, then the finite difference function evaluations can be performed concurrently (using any of the parallel modes described in the Users Manual [Adams et al., 2010]). An important related observation is the fact that NPSOL uses two different line searches depending on how gradients are computed. For either `analytic_gradients` or `numerical_gradients` with `method_source dakota`, NPSOL is placed in user-supplied gradient mode (NPSOL's "Derivative Level" is set to 3) and it uses a gradient-based line search (the assumption is that user-supplied gradients are inexpensive). On the other hand, if `numerical_gradients` are selected with `method_source vendor`, then NPSOL is computing finite differences internally and it will use a value-based line search (the assumption is that finite differencing on each line search evaluation is too expensive). The ramifications of this are: (1) performance will vary between `method_source dakota` and `method_source vendor` for `numerical_gradients`, and (2) gradient speculation is unnecessary when performing optimization in parallel since the gradient-based line search in user-supplied gradient mode is already load balanced for parallel execution. Therefore, a `speculative` specification will be ignored by NPSOL, and optimization with numerical gradients should select `method_source dakota` for load balanced parallel operation and `method_source vendor` for efficient serial operation.

Lastly, NPSOL supports specialized handling of linear inequality and equality constraints. By specifying the coefficients and bounds of the linear inequality constraints and the coefficients and targets of the linear equality constraints, this information can be provided to NPSOL at initialization and tracked internally, removing the need for the user to provide the values of the linear constraints on every function evaluation. Refer to Method Independent Controls for additional information and to Tables 5.1 through 5.2 for method independent control specification detail.

20.2.2 NPSOL method dependent controls

NPSOL's method dependent controls are `verify_level`, `function_precision`, and `linesearch_tolerance`. The `verify_level` control instructs NPSOL to perform finite difference verifications on user-supplied gradient components. The `function_precision` control provides NPSOL an estimate of the accuracy to which the problem functions can be computed. This is used to prevent NPSOL from trying to distinguish between function values that differ by less than the inherent error in the calculation. And the `linesearch_tolerance` setting controls the accuracy of the line search. The smaller the value (between 0 and 1), the more accurately NPSOL will attempt to compute a precise minimum along the search direction. Table 5.4 provides the specification detail for the NPSOL SQP method and its method dependent controls.

Description	Keyword	Associated Data	Status	Default
Gradient verification level	`verify_level`	integer	Optional	-1 (no gradient verification)
Function precision	`function_precision`	real	Optional	1.e-10
Line search tolerance	`linesearch_tolerance`	real	Optional	0.9 (inaccurate line search)

Table 20.2: Specification detail for the NPSOL SQP method

20.3 NLPQL Methods

The NLPQL library is a commercially-licensed library containing a sequential quadratic programming (SQP) optimizer, specified as DAKOTA's `nlpql_sqp` method, for constrained optimization. The particular implementation used is NLPQLP [Schittkowski, 2004], a variant with distributed and non-monotone line search. DAKOTA provides access to the NLPQL library through the **NLPQLPOptimizer** class.

20.3.1 NLPQL method independent controls

The method independent controls for maximum iterations and output verbosity are mapped to NLPQL controls MAXIT and IPRINT, respectively. The maximum number of function evaluations is enforced within the **NLPQLPOptimizer** class.

20.3.2 NLPQL method dependent controls

NLPQL does not currently support any method dependent controls.

20.4 CONMIN Methods

The CONMIN library [Vanderplaats, 1973] is a public domain library of nonlinear programming optimizers, specifically the Fletcher-Reeves conjugate gradient (DAKOTA's `conmin_frcg` method) method for unconstrained optimization, and the method of feasible directions (DAKOTA's `conmin_mfd` method) for constrained optimization. As CONMIN was a predecessor to the DOT commercial library, the algorithm controls are very similar. DAKOTA provides access to the CONMIN library through the **CONMINOptimizer** class.

20.4.1 CONMIN method independent controls

The interpretations of the method independent controls for CONMIN are essentially identical to those for DOT. Therefore, the discussion in DOT method independent controls is relevant for CONMIN.

20.4.2 CONMIN method dependent controls

CONMIN does not currently support any method dependent controls.

20.5 OPT++ Methods

The OPT++ library [Meza et al., 2007] contains primarily gradient-based nonlinear programming optimizers for unconstrained, bound-constrained, and nonlinearly constrained minimization: Polak-Ribiere conjugate gradient (DAKOTA's `optpp_cg` method), quasi-Newton (DAKOTA's `optpp_q_newton` method), finite difference Newton (DAKOTA's `optpp_fd_newton` method), and full Newton (DAKOTA's `optpp_newton` method). The conjugate gradient method is strictly unconstrained, and each of the Newton-based methods are automatically

bound to the appropriate OPT++ algorithm based on the user constraint specification (unconstrained, bound-constrained, or generally-constrained). In the generally-constrained case, the Newton methods use a nonlinear interior-point approach to manage the constraints. The library also contains a direct search algorithm, PDS (parallel direct search, DAKOTA's `optpp_pds` method), which supports bound constraints. DAKOTA provides access to the OPT++ library through the **SNLLOptimizer** class, where "SNLL" denotes Sandia National Laboratories - Livermore.

20.5.1 OPT++ method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during an OPT++ optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. The `output` verbosity specification controls the amount of information generated from OPT++ executions: the `debug` setting turns on OPT++'s internal debug mode and also generates additional debugging information from DAKOTA's **SNLLOptimizer** wrapper class. OPT++'s gradient-based methods are not parallel algorithms and cannot directly take advantage of concurrent function evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, a parallel DAKOTA configuration can utilize concurrent evaluations for the finite difference gradient computations. OPT++'s nongradient-based PDS method can directly exploit asynchronous evaluations; however, this capability has not yet been implemented in the **SNLLOptimizer** class.

The `speculative` specification enables speculative computation of gradient and/or Hessian information, where applicable, for parallel optimization studies. By speculating that the derivative information at the current point will be used later, the complete data set (all available gradient/Hessian information) can be computed on every function evaluation. While some of these computations will be wasted, the positive effects are a consistent parallel load balance and usually shorter wall clock time. The `speculative` specification is applicable only when parallelism in the gradient calculations can be exploited by DAKOTA (it will be ignored for `vendor numerical` gradients).

Lastly, linear constraint specifications are supported by each of the Newton methods (`optpp_newton`, `optpp_q_newton`, `optpp_fd_newton`, and `optpp_g_newton`); whereas `optpp_cg` must be unconstrained and `optpp_pds` can be, at most, bound-constrained. Specification detail for the method independent controls is provided in Tables 5.1 through 5.2.

20.5.2 OPT++ method dependent controls

OPT++'s method dependent controls are `max_step`, `gradient_tolerance`, `search_method`, `merit_function`, `central_path`, `steplength_to_boundary`, `centering_parameter`, and `search_scheme_size`. The `max_step` control specifies the maximum step that can be taken when computing a change in the current design point (e.g., limiting the Newton step computed from current gradient and Hessian information). It is equivalent to a move limit or a maximum trust region size. The `gradient_tolerance` control defines the threshold value on the L2 norm of the objective function gradient that indicates convergence to an unconstrained minimum (no active constraints). The `gradient_tolerance` control is defined for all gradient-based optimizers.

`max_step` and `gradient_tolerance` are the only method dependent controls for the OPT++ conjugate gradient method. Table 5.5 covers this specification.

The `search_method` control is defined for all Newton-based optimizers and is used to select between `trust_region`, `gradient_based_line_search`, and `value_based_line_search` methods. The

Description	Keyword	Associated Data	Status	Default
OPT++ conjugate gradient method	optpp_cg	none	Required	N/A
Maximum step size	max_step	real	Optional	1000.
Gradient tolerance	gradient_-tolerance	real	Optional	1.e-4

Table 20.3: Specification detail for the OPT++ conjugate gradient method

gradient_based_line_search option uses the line search method proposed by [More and Thuente, 1994]. This option satisfies sufficient decrease and curvature conditions; whereas, value_base_line_search only satisfies the sufficient decrease condition. At each line search iteration, the gradient_based_line_search method computes the function and gradient at the trial point. Consequently, given expensive function evaluations, the value_based_line_search method is preferred to the gradient_based_line_search method. Each of these Newton methods additionally supports the tr_pds selection for unconstrained problems. This option performs a robust trust region search using pattern search techniques. Use of a line search is the default for bound-constrained and generally-constrained problems, and use of a trust_region search method is the default for unconstrained problems.

The merit_function, central_path, steplength_to_boundary, and centering_parameter selections are additional specifications that are defined for the solution of generally-constrained problems with nonlinear interior-point algorithms. A merit_function is a function in constrained optimization that attempts to provide joint progress toward reducing the objective function and satisfying the constraints. Valid string inputs are "el_bakry", "argaez_tapia", or "van_shanno", where user input is not case sensitive in this case. Details for these selections are as follows:

- The "el_bakry" merit function is the L2-norm of the first order optimality conditions for the nonlinear programming problem. The cost per linesearch iteration is n+1 function evaluations. For more information, see [El-Bakry et al., 1996].

- The "argaez_tapia" merit function can be classified as a modified augmented Lagrangian function. The augmented Lagrangian is modified by adding to its penalty term a potential reduction function to handle the perturbed complementarity condition. The cost per linesearch iteration is one function evaluation. For more information, see [Tapia and Argaez].

- The "van_shanno" merit function can be classified as a penalty function for the logarithmic barrier formulation of the nonlinear programming problem. The cost per linesearch iteration is one function evaluation. For more information see [Vanderbei and Shanno, 1999].

If the function evaluation is expensive or noisy, set the merit_function to "argaez_tapia" or "van_shanno".

The central_path specification represents a measure of proximity to the central path and specifies an update strategy for the perturbation parameter mu. Refer to [Argaez et al., 2002] for a detailed discussion on proximity measures to the central region. Valid options are, again, "el_bakry", "argaez_tapia", or "van_shanno", where user input is not case sensitive. The default value for central_path is the value of merit_function (either user-selected or default). The steplength_to_boundary specification is a parameter (between 0 and 1) that controls how close to the boundary of the feasible region the algorithm is allowed to move. A value of 1 means that the algorithm is allowed to take steps that may reach the boundary of the feasible region. If the user wishes to

maintain strict feasibility of the design parameters this value should be less than 1. Default values are .8, .99995, and .95 for the "el_bakry", "argaez_tapia", and "van_shanno" merit functions, respectively. The `centering_parameter` specification is a parameter (between 0 and 1) that controls how closely the algorithm should follow the "central path". See [Wright] for the definition of central path. The larger the value, the more closely the algorithm follows the central path, which results in small steps. A value of 0 indicates that the algorithm will take a pure Newton step. Default values are .2, .2, and .1 for the "el_bakry", "argaez_tapia", and "van_shanno" merit functions, respectively.

Table 5.6 provides the details for the Newton-based methods.

Description	Keyword	Associated Data	Status	Default
OPT++ Newton-based methods	`optpp_q_newton` | `optpp_fd_newton` | `optpp_newton`	none	Required group	N/A
Search method	`value_based_line_search` | `gradient_based_line_search` | `trust_region` | `tr_pds`	none	Optional group	`trust_region` (unconstrained), `value_based_line_search` (bound/general constraints)
Maximum step size	`max_step`	real	Optional	1000.
Gradient tolerance	`gradient_tolerance`	real	Optional	1.e-4
Merit function	`merit_function`	string	Optional	`"argaez_tapia"`
Central path	`central_path`	string	Optional	value of `merit_function`
Steplength to boundary	`steplength_to_boundary`	real	Optional	Merit function dependent: 0.8 (el_bakry), 0.99995 (argaez_tapia), 0.95 (van_shanno)
Centering parameter	`centering_parameter`	real	Optional	Merit function dependent: 0.2 (el_bakry), 0.2 (argaez_tapia), 0.1 (van_shanno)

Table 20.4: Specification detail for OPT++ Newton-based optimization methods

The `search_scheme_size` is defined for the PDS method to specify the number of points to be used in the direct search template. PDS does not support parallelism at this time due to current limitations in the OPT++ interface. Table 5.7 provides the detail for the parallel direct search method.

Description	Keyword	Associated Data	Status	Default
OPT++ parallel direct search method	optpp_pds	none	Required group	N/A
Search scheme size	search_-scheme_size	integer	Optional	32

Table 20.5: Specification detail for the OPT++ PDS method

20.6 Asynchronous Parallel Pattern Search (APPS)

The asynchronous parallel pattern search (APPS) algorithm [Gray and Kolda, 2006] is a fully asynchronous pattern search technique in that the search along each offset direction continues without waiting for searches along other directions to finish. It is now made available in DAKOTA through the HOPSPACK software [Plantenga, 2009]. It can handle unconstrained problems as well as those with bound constraints, linear constraints, and general nonlinear constraints. HOPSPACK is available to the public under the GNU LGPL and the source code is included with DAKOTA. HOPSPACK-specific software documentation is available from https://software.sandia.gov/trac/hopspack.

20.6.1 APPS method independent controls

The only method independent controls that are currently mapped to APPS are max_function_-evaluations, constraint_tolerance, and the output verbosity control. We note that while APPS treats the constraint tolerance separately for linear and nonlinear constraints, we apply the same value to both if the user specifies constraint_tolerance. The APPS internal "display" level is mapped to the DAKOTA debug, verbose, normal, quiet, and silent settings as follows:

- DAKOTA "debug": display final solution, all input parameters, variable and constraint info, trial points, search directions, and execution details
- DAKOTA "verbose": display final solution, all input parameters, variable and constraint info, and trial points
- DAKOTA "normal": display final solution, all input parameters, variable and constraint summaries, and new best points
- DAKOTA "quiet": display final solution and all input parameters
- DAKOTA ""silent: display final solution

APPS exploits parallelism through the use of DAKOTA's concurrent function evaluations. The variant of the algorithm that is currently exposed, however, limits the amount of concurrency that can be exploited. In particular, APPS can leverage an evaluation concurrency level of at most twice the number of variables. More options that allow for greater evaluation concurrency may be exposed in future releases.

20.6.2 APPS method dependent controls

The APPS method is invoked using a asynch_pattern_search group specification. Some of the method dependent controls are similar to the SCOLIB controls for coliny_pattern_search described in Pattern

Search. In particular, APPS supports the following step length control parameters:

- `initial_delta`: the initial step length, must be positive
- `threshold_delta`: step length used to determine convergence, must be greater than or equal to 4.4e-16
- `contraction_factor`: amount by which step length is rescaled after unsuccesful iterates, must be strictly between 0 and 1

When the solution to the optimization problem is known, the user may specify a value for `solution_target` as a termination criteria. APPS will terminate when the function value falls below `solution_target`.

Currently, APPS only supports coordinate bases with a total of *2n* function evaluations in the pattern, and these patterns may only contract. The `synchronization` specification can be used to specify the use of either `blocking` or `nonblocking` schedulers for APPS. The `blocking` option causes APPS to behave as a synchronous algorithm. The `nonblocking` option is not available when **Dakota** is used in message-passing mode.

APPS solves nonlinearly constrained problems by solving a sequence of linearly constrained merit function-base subproblems. There are several exact and smoothed exact penalty functions that can be specified with the `merit_function` control. The options are as follows:

- `merit_max`: based on ℓ_∞ norm
- `merit_max_smooth`: based on smoothed ℓ_∞ norm
- `merit1`: based on ℓ_1 norm
- `merit1_smooth`: based on smoothed ℓ_1 norm
- `merit2`: based on ℓ_2 norm
- `merit2_smooth`: based on smoothed ℓ_2 norm
- `merit2_squared`: based on ℓ_2^2 norm

The user can also specify the following:

- `constraint_penalty`: the initial multiplier for the penalty function, must be nonnegative
- `smoothing_parameter`: initial smoothing value for smoothed penalty functions, must be between 0 and 1 (inclusive)

Table 5.8 summarizes the APPS specification.

20.7 SCOLIB Methods

SCOLIB (formerly known as COLINY) is a collection of nongradient-based optimizers that support the Common Optimization Library INterface (COLIN). SCOLIB optimizers currently include `coliny_cobyla`, `coliny_direct`, `coliny_ea`, `coliny_pattern_search` and `coliny_solis_wets`. (Yes, the input spec still has "coliny" prepended to the method name.) Additional SCOLIB information is available from https://software.sandia.gov/trac/acro.

Description	Keyword	Associated Data	Status	Default
APPS method	asynch_-pattern_-search	none	Required group	N/A
Initial offset value	initial_-delta	real	Optional	1.0
Threshold for offset values	threshold_-delta	real	Optional	0.01
Pattern contraction factor	contraction_-factor	real	Optional	0.5
Solution target	solution_-target	real	Optional	not used
Evaluation synchronization	synchronization	blocking | nonblocking	Optional	nonblocking
Merit function	merit_-function	merit_max | merit_max_-smooth | merit1 | merit1_-smooth | merit2 | merit2_-smooth | merit2_-squared	Optional	merit2_-squared
Constraint penalty	constraint_-penalty	real	Optional	1.0
Smoothing factor	smoothing_-factor	real	Optional	0.0

Table 20.6: Specification detail for the APPS method

SCOLIB solvers now support bound constraints and general nonlinear constraints. Supported nonlinear constraints include both equality and two-sided inequality constraints. SCOLIB solvers do not yet support linear constraints. Most SCOLIB optimizers treat constraints with a simple penalty scheme that adds `constraint_penalty` times the sum of squares of the constraint violations to the objective function. Specific exceptions to this method for handling constraint violations are noted below. (The default value of `constraint_penalty` is 1000.0, except for methods that dynamically adapt their constraint penalty, for which the default value is 1.0.)

20.7.1 SCOLIB method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during a SCOLIB optimization, respectively. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. The `output` verbosity specification controls the amount of information generated by SCOLIB: the `silent`, `quiet`, and `normal` settings correspond to minimal reporting from SCOLIB, whereas the `verbose` setting corresponds to a higher level of information, and `debug` outputs method initialization and a variety of internal SCOLIB diagnostics. The majority of SCOLIB's methods perform independent function evaluations that can directly take advantage of DAKOTA's parallel capabilities. Only `coliny_solis_wets`, `coliny_cobyla`, and certain configurations of `coliny_pattern_search` are inherently serial (see Pattern Search). The parallel methods automatically utilize parallel logic when the DAKOTA configuration supports parallelism. Lastly, neither `speculative` gradients nor linear constraints are currently supported with SCOLIB. Specification detail for method independent controls is provided in Tables 5.1 through 5.2.

Some SCOLIB methods exploit parallelism through the use of DAKOTA's concurrent function evaluations. The nature of the algorithms, however, limits the amount of concurrency that can be exploited. The maximum amount of evaluation concurrency that can be leveraged by the various methods is as follows:

- COBYLA: one
- DIRECT: twice the number of variables
- Evolutionary Algorithms: size of the population
- Pattern Search: size of the search pattern
- Solis-Wets: one

20.7.2 SCOLIB method dependent controls

All SCOLIB methods support the `show_misc_options` optional specification which results in a dump of all the allowable method inputs. Note that the information provided by this command refers to optimizer parameters that are internal to SCOLIB, and which may differ from corresponding parameters used by the DAKOTA interface. The `misc_options` optional specification provides a means for inputing additional settings supported by the SCOLIB methods but which are not currently mapped through the DAKOTA input specification. Care must be taken in using this specification; they should only be employed by users familiar with the full range of parameter specifications available directly from SCOLIB and understand any differences that exist between those specifications and the ones available through DAKOTA.

Each of the SCOLIB methods supports the `solution_target` control, which defines a convergence criterion in which the optimizer will terminate if it finds an objective function value lower than the specified target. Specification detail for method dependent controls for all SCOLIB methods is provided in Table 5.9.

Description	Keyword	Associated Data	Status	Default
Show miscellaneous options	show_misc_-options	none	Optional	no dump of specification options
Specify miscellaneous options	misc_options	list of strings	Optional	no miscellaneous options specified
Desired solution target	solution_-target	real	Optional	-DBL_MAX

Table 20.7: Specification detail for SCOLIB method dependent controls

Each SCOLIB method supplements the settings of Table 5.9 with controls which are specific to its particular class of method.

20.7.3 COBYLA

The Constrained Optimization BY Linear Approximations (COBYLA) algorithm is an extension to the Nelder-Mead simplex algorithm for handling general linear/nonlinear constraints and is invoked using the coliny_-cobyla group specification. The COBYLA algorithm employs linear approximations to the objective and constraint functions, the approximations being formed by linear interpolation at N+1 points in the space of the variables. We regard these interpolation points as vertices of a simplex. The step length parameter controls the size of the simplex and it is reduced automatically from initial_delta to threshold_delta. One advantage that COBYLA has over many of its competitors is that it treats each constraint individually when calculating a change to the variables, instead of lumping the constraints together into a single penalty function.

COBYLA currently only supports termination based on the max_function_evaluations and solution_target specifications. The search performed by COBYLA is currently not parallelized.

Table 5.10 summarizes the COBYLA specification.

Description	Keyword	Associated Data	Status	Default
COBYLA method	coliny_-cobyla	none	Required group	N/A
Initial offset value	initial_-delta	real	Required	N/A
Threshold for offset values	threshold_-delta	real	Required	N/A

Table 20.8: Specification detail for the COBYLA method

20.7.4 DIRECT

The DIviding RECTangles (DIRECT) optimization algorithm is a derivative free global optimization method that balances local search in promising regions of the design space with global search in unexplored regions. As shown in Figure 5.1, DIRECT adaptively subdivides the space of feasible design points so as to guarantee that iterates are generated in the neighborhood of a global minimum in finitely many iterations.

Figure 20.1: Design space partitioning with DIRECT

In practice, DIRECT has proven an effective heuristic for engineering design applications, for which it is able to quickly identify candidate solutions that can be further refined with fast local optimizers.

DIRECT uses the `solution_target`, `constraint_penalty` and `show_misc_options` specifications that are described in SCOLIB method dependent controls. Note, however, that DIRECT uses a fixed penalty value for constraint violations (i.e. it is not dynamically adapted as is done in `coliny_pattern_search`).

The `division` specification determines how DIRECT subdivides each subregion of the search space. If `division` is set to `major_dimension`, then the dimension representing the longest edge of the subregion is subdivided (this is the default). If `division` is set to `all_dimensions`, then all dimensions are simultaneously subdivided.

Each subregion considered by DIRECT has a **size**, which corresponds to the longest diagonal of the subregion. The `global_balance_parameter` controls how much global search is performed by only allowing a subregion to be subdivided if the size of the subregion divided by the size of the largest subregion is at least `global_balance_parameter`. Intuitively, this forces large subregions to be subdivided before the smallest subregions are refined. The `local_balance_parameter` provides a tolerance for estimating whether the smallest subregion can provide a sufficient decrease to be worth subdividing; the default value is a small value that is suitable for most applications.

DIRECT can be terminated with the standard `max_function_evaluations` and `solution_target` specifications. Additionally, the `max_boxsize_limit` specification terminates DIRECT if the size of the largest subregion falls below this threshold, and the `min_boxsize_limit` specification terminates DIRECT if the size of the smallest subregion falls below this threshold. In practice, this latter specification is likely to be more effective at limiting DIRECT's search.

Table 5.11 summarizes the DIRECT specification.

Description	Keyword	Associated Data	Status	Default
DIRECT method	coliny_-direct	none	Required group	N/A
Box subdivision approach	division	major_-dimension| all_-dimensions	Optional group	major_-dimension
Global search balancing parameter	global_-balance_-parameter	real	Optional	0.0
Local search balancing parameter	local_-balance_-parameter	real	Optional	1.e-8
Maximum boxsize limit	max_-boxsize_-limit	real	Optional	0.0
Minimum boxsize limit	min_-boxsize_-limit	real	Optional	0.0001
Constraint penalty	constraint_-penalty	real	Optional	1000.0

Table 20.9: Specification detail for the DIRECT method

20.7.5 Evolutionary Algorithms

DAKOTA currently provides several variants of evolutionary algorithms, invoked through the coliny_ea group specification.

The basic steps of an evolutionary algorithm are depicted in Figure 5.2.

Figure 20.2: Depiction of evolutionary algorithm

They can be enumerated as follows:

1. Select an initial population randomly and perform function evaluations on these individuals
2. Perform selection for parents based on relative fitness
3. Apply crossover and mutation to generate `new_solutions_generated` new individuals from the selected parents
   - Apply crossover with a fixed probability from two selected parents
   - If crossover is applied, apply mutation to the newly generated individual with a fixed probability
   - If crossover is not applied, apply mutation with a fixed probability to a single selected parent
4. Perform function evaluations on the new individuals
5. Perform replacement to determine the new population
6. Return to step 2 and continue the algorithm until convergence criteria are satisfied or iteration limits are exceeded

Table 5.12 provides the specification detail for the controls for seeding the method, initializing a population, and for selecting and replacing population members.

The random `seed` control provides a mechanism for making a stochastic optimization repeatable. That is, the use of the same random seed in identical studies will generate identical results. The `population_size` control specifies how many individuals will comprise the EA's population.

The `initialization_type` defines the type of initialization for the population of the EA. There are three types: `simple_random`, `unique_random`, and `flat_file`. `simple_random` creates initial solutions with random variable values according to a uniform random number distribution. It gives no consideration to any previously generated designs. The number of designs is specified by the `population_size`. `unique_random` is the same as `simple_random`, except that when a new solution is generated, it is checked against the rest of the solutions. If it duplicates any of them, it is rejected. `flat_file` allows the initial population to be read from a flat file. If `flat_file` is specified, a file name must be given.

The `fitness_type` controls how strongly differences in "fitness" (i.e., the objective function) are weighted in the process of selecting "parents" for crossover:

- the `linear_rank` setting uses a linear scaling of probability of selection based on the rank order of each individual's objective function within the population
- the `merit_function` setting uses a proportional scaling of probability of selection based on the relative value of each individual's objective function within the population

The `replacement_type` controls how current populations and newly generated individuals are combined to create a new population. Each of the `replacement_type` selections accepts an integer value, which is referred to below and in Table 5.12 as the `replacement_size`:

- The `random` setting creates a new population using (a) `replacement_size` randomly selected individuals from the current population, and (b) `population_size` - `replacement_size` individuals randomly selected from among the newly generated individuals (the number of which is optionally specified using `new_solutions_generated`) that are created for each generation (using the selection, crossover, and mutation procedures).

Description	Keyword	Associated Data	Status	Default
EA selection	coliny_ea	none	Required group	N/A
Random seed	seed	integer	Optional	randomly generated seed
Number of population members	population_-size	integer	Optional	50
Initialization type	initialization_-type	simple_-random | unique_-random | flat_file	Required	unique_-random
Fitness type	fitness_type	linear_rank | merit_-function	Optional	linear_rank
Replacement type	replacement_-type	random | chc | elitist	Optional group	elitist = 1
Random replacement type	random	integer	Required	N/A
CHC replacement type	chc	integer	Required	N/A
Elitist replacement type	elitist	integer	Required	N/A
New solutions generated	new_-solutions_-generated	integer	Optional	population_-size - replacement_-size

Table 20.10: Specification detail for the SCOLIB EA method dependent controls: seed, initialization, selection, and replacement

- The chc setting creates a new population using (a) the replacement_size best individuals from the *combination* of the current population and the newly generated individuals, and (b) population_size - replacement_size individuals randomly selected from among the remaining individuals in this combined pool. The chc setting is the preferred selection for many engineering problems.

- The elitist (default) setting creates a new population using (a) the replacement_size best individuals from the current population, (b) and population_size - replacement_size individuals randomly selected from the newly generated individuals. It is possible in this case to lose a good solution from the newly generated individuals if it is not randomly selected for replacement; however, the default new_solutions_generated value is set such that the entire set of newly generated individuals will be selected for replacement.

Note that new_solutions_generated is not recognized by DAKOTA as a valid keyword unless replacement_type has been specified.

Table 5.13 show the controls in the EA method associated with crossover and mutation.

Description	Keyword	Associated Data	Status	Default
Crossover type	crossover_-type	two_point | blend | uniform	Optional group	two_point
Crossover rate	crossover_-rate	real	Optional	0.8
Mutation type	mutation_-type	replace_-uniform | offset_-normal | offset_-cauchy | offset_-uniform	Optional group	offset_-normal
Mutation scale	mutation_-scale	real	Optional	0.1
Mutation range	mutation_-range	integer	Optional	1
Mutation dimension ratio	dimension_-ratio	real	Optional	1.0
Mutation rate	mutation_-rate	real	Optional	1.0
Non-adaptive mutation flag	non_adaptive	none	Optional	Adaptive mutation

Table 20.11: Specification detail for the SCOLIB EA method: crossover and mutation

The crossover_type controls what approach is employed for combining parent genetic information to create offspring, and the crossover_rate specifies the probability of a crossover operation being performed to generate a new offspring. The SCOLIB EA method supports three forms of crossover, two_point, blend, and uniform, which generate a new individual through combinations of two parent individuals. Two-point crossover divides each parent into three regions, where offspring are created from the combination of the middle region from one parent and the end regions from the other parent. Since the SCOLIB EA does not utilize bit

representations of variable values, the crossover points only occur on coordinate boundaries, never within the bits of a particular coordinate. Uniform crossover creates offspring through random combination of coordinates from the two parents. Blend crossover generates a new individual randomly along the multidimensional vector connecting the two parents.

The `mutation_type` controls what approach is employed in randomly modifying continuous design variables within the EA population. Each of the mutation methods generates coordinate-wise changes to individuals, usually by adding a random variable to a given coordinate value (an "offset" mutation), but also by replacing a given coordinate value with a random variable (a "replace" mutation). Discrete design variables are always mutated using the `offset_uniform` method. The `mutation_rate` controls the probability of mutation being performed on an individual, both for new individuals generated by crossover (if crossover occurs) and for individuals from the existing population. When mutation is performed, all dimensions of each individual are mutated. The `mutation_scale` specifies a scale factor which scales continuous mutation offsets; this is a fraction of the total range of each dimension, so `mutation_scale` is a relative value between 0 and 1. The `mutation_range` is used to control `offset_uniform` mutation used for discrete parameters. The replacement discrete value is the original value plus or minus an integer value up to `mutation_range`. The `offset_normal`, `offset_cauchy`, and `offset_uniform` mutation types are "offset" mutations in that they add a 0-mean random variable with a normal, cauchy, or uniform distribution, respectively, to the existing coordinate value. These offsets are limited in magnitude by `mutation_scale`. The `replace_uniform` mutation type is not limited by `mutation_scale`; rather it generates a replacement value for a coordinate using a uniformly distributed value over the total range for that coordinate.

Note that `mutation_scale` and `mutation_range` are not recognized by DAKOTA as valid keywords unless `mutation_type` has been specified and the type is an "offset" mutations.

The SCOLIB EA method uses self-adaptive mutation, which modifies the mutation scale dynamically. This mechanism is borrowed from EAs like evolution strategies. The `non_adaptive` flag can be used to deactivate the self-adaptation, which may facilitate a more global search.

Note that `non_adaptive` is not recognized by DAKOTA as a valid keyword unless `mutation_type` has been specified.

20.7.6 Pattern Search

Pattern search techniques are nongradient-based optimization methods which use a set of offsets from the current iterate to locate improved points in the design space. The SCOLIB pattern search technique is invoked using a `coliny_pattern_search` group specification, which includes a variety of specification components.

Traditional pattern search methods search with a fixed pattern of search directions to try to find improvements to the current iterate. The SCOLIB pattern search methods generalize this simple algorithmic strategy to enable control of how the search pattern is adapted, as well as how each search pattern is evaluated. The `stochastic` and `synchronization` specifications denote how the the trial points are evaluated. The `stochastic` specification indicates that the trial points are considered in a random order. For parallel pattern search, `synchronization` dictates whether the evaluations are scheduled using a `blocking` scheduler or a `nonblocking` scheduler (i.e., **Model::synchronize()** or **Model::synchronize_nowait()**, respectively). In the `blocking` case, all points in the pattern are evaluated (in parallel), and if the best of these trial points is an improving point, then it becomes the next iterate. These runs are reproducible, assuming use of the same seed in the `stochastic` case. In the `nonblocking` case, all points in the pattern may not be evaluated, since the first improving point found becomes the next iterate. Since the algorithm steps will be subject to parallel timing variabilities, these runs will not generally be repeatable. The `synchronization` specification has similar connotations for sequential pattern search. If `blocking` is specified, then each sequential iteration terminates after

all trial points have been considered, and if `nonblocking` is specified, then each sequential iteration terminates after the first improving trial point is evaluated. In this release, both `blocking` and `nonblocking` specifications result in blocking behavior (except in the case where `exporatory_moves` below is set to `adaptive_pattern`). Nonblocking behavior will be re-enabled after some underlying technical issues have been resolved.

The particular form of the search pattern is controlled by the `pattern_basis` specification. If `pattern_basis` is `coordinate` basis, then the pattern search uses a plus and minus offset in each coordinate direction, for a total of *2n* function evaluations in the pattern. This case is depicted in Figure 5.3 for three coordinate dimensions.

Figure 20.3: Depiction of coordinate pattern search algorithm

If `pattern_basis` is `simplex`, then pattern search uses a minimal positive basis simplex for the parameter space, for a total of *n+1* function evaluations in the pattern. Note that the `simplex` pattern basis can be used for unbounded problems only. The `total_pattern_size` specification can be used to augment the basic `coordinate` and `simplex` patterns with additional function evaluations, and is particularly useful for parallel load balancing. For example, if some function evaluations in the pattern are dropped due to duplication or bound constraint interaction, then the `total_pattern_size` specification instructs the algorithm to generate new offsets to bring the total number of evaluations up to this consistent total.

The `exploratory_moves` specification controls how the search pattern is adapted. (The search pattern can be adapted after an improving trial point is found, or after all trial points in a search pattern have been found to be unimproving points.) The following exploratory moves selections are supported by SCOLIB:

- The basic_pattern case is the simple pattern search approach, which uses the same pattern in each iteration.

- The multi_step case examines each trial step in the pattern in turn. If a successful step is found, the pattern search continues examining trial steps about this new point. In this manner, the effects of multiple successful steps are cumulative within a single iteration. This option does not support any parallelism and will result in a serial pattern search.

- The adaptive_pattern case invokes a pattern search technique that adaptively rescales the different search directions to maximize the number of redundant function evaluations. See [Hart et al., 2001] for details of this method. In preliminary experiments, this method had more robust performance than the standard basic_pattern case in serial tests. This option supports a limited degree of parallelism. After successful iterations (where the step length is not contracted), a parallel search will be performed. After unsuccessful iterations (where the step length is contracted), only a single evaluation is performed.

The initial_delta and threshold_delta specifications provide the initial offset size and the threshold size at which to terminate the algorithm. For any dimension that has both upper and lower bounds, this step length will be internally rescaled to provide search steps of length initial_delta * range. This rescaling does not occur for other dimensions, so search steps in those directions have length initial_delta.

In general, pattern search methods can expand and contract their step lengths. SCOLIB pattern search methods contract the step length by the value contraction_factor, and they expand the step length by the value (1/contraction_factor). The expand_after_success control specifies how many successful objective function improvements must occur with a specific step length prior to expansion of the step length, whereas the no_expansion flag instructs the algorithm to forgo pattern expansion altogether.

Finally, constraint infeasibility can be managed in a somewhat more sophisticated manner than the simple weighted penalty function. If the constant_penalty specification is used, then the simple weighted penalty scheme described above is used. Otherwise, the constraint penalty is adapted to the value constraint_penalty/L, where L is the the smallest step length used so far.

Table 5.14 and Table 5.15 provide the specification detail for the SCOLIB pattern search method and its method dependent controls.

20.7.7 Solis-Wets

DAKOTA's implementation of SCOLIB also contains the Solis-Wets algorithm. The Solis-Wets method is a simple greedy local search heuristic for continuous parameter spaces. Solis-Wets generates trial points using a multivariate normal distribution, and unsuccessful trial points are reflected about the current point to find a descent direction. This algorithm is inherently serial and will not utilize any parallelism. Table 5.16 provides the specification detail for this method and its method dependent controls.

These specifications have the same meaning as corresponding specifications for coliny_pattern_search. In particular, coliny_solis_wets supports dynamic rescaling of the step length, and dynamic rescaling of the constraint penalty. The only new specification is contract_after_failure, which specifies the number of unsuccessful cycles which must occur with a specific delta prior to contraction of the delta.

Description	Keyword	Associated Data	Status	Default
SCOLIB pattern search method	coliny_-pattern_-search	none	Required group	N/A
Stochastic pattern search	stochastic	none	Optional group	N/A
Random seed for stochastic pattern search	seed	integer	Optional	randomly generated seed
Initial offset value	initial_-delta	real	Required	N/A
Threshold for offset values	threshold_-delta	real	Required	N/A
Constraint penalty	constraint_-penalty	real	Optional	1.0
Control of dynamic penalty	constant_-penalty	none	Optional	algorithm dynamically adapts the constraint penalty

Table 20.12: Specification detail for the SCOLIB pattern search method: randomization, delta, and constraint controls

Description	Keyword	Associated Data	Status	Default
Pattern basis selection	pattern_-basis	coordinate | simplex	Optional	coordinate
Total number of points in pattern	total_-pattern_size	integer	Optional	no augmentation of basic pattern
No expansion flag	no_expansion	none	Optional	algorithm may expand pattern size
Number of consecutive improvements before expansion	expand_-after_-success	integer	Optional	5
Pattern contraction factor	contraction_-factor	real	Optional	0.5
Evaluation synchronization	synchronization	blocking | nonblocking	Optional	nonblocking
Exploratory moves selection	exploratory_-moves	basic_-pattern | multi_step | adaptive_-pattern	Optional	basic_-pattern

Table 20.13: Specification detail for the SCOLIB pattern search method: pattern controls

Description	Keyword	Associated Data	Status	Default
SCOLIB Solis-Wets method	coliny_-solis_wets	none	Required group	N/A
Random seed for stochastic pattern search	seed	integer	Optional	randomly generated seed
Initial offset value	initial_-delta	real	Required	N/A
Threshold for offset values	threshold_-delta	real	Required	N/A
No expansion flag	no_expansion	none	Optional	algorithm may expand pattern size
Number of consecutive improvements before expansion	expand_-after_-success	integer	Optional	5
Number of consecutive failures before contraction	contract_-after_-failure	integer	Optional	4∗number of variables
Pattern contraction factor	contraction_-factor	real	Optional	0.5
Constraint penalty	constraint_-penalty	real	Optional	1.0
Control of dynamic penalty	constant_-penalty	none	Optional	algorithm dynamically adapts the constraint penalty

Table 20.14: Specification detail for the SCOLIB Solis-Wets method

20.8 NCSU Methods

North Carolina State University (NCSU) has an implementation of the DIRECT algorithm (DIviding RECTangles algorithm that is outlined in the SCOLIB method section above). This version is documented in [Gablonsky, 2001.] We have found that the NCSU DIRECT implementation works better and is more robust for some problems than `coliny_direct`. Currently, we maintain both versions of DIRECT in DAKOTA; in the future, we may deprecate one. The NCSU DIRECT method is selected with `ncsu_direct`. We have tried to maintain consistency between the keywords in SCOLIB and NCSU implementation of DIRECT, but the algorithms have different parameters, so the keywords sometimes have slightly different meaning.

20.8.1 NCSU method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of iterations and the number of function evaluations that can be performed during an NCSU DIRECT optimization. This methods will always strictly respect the number of iterations, but may slightly exceed the number of function evaluations, as it will always explore all sub-rectangles at the current level.

20.8.2 NCSU method dependent controls

There are four specification controls affecting NCSU DIRECT: `solution_target`, `convergence_tolerance`, `min_boxsize_limit`, and `volume_boxsize_limit`. The solution target specifies a goal toward which the optimizer should track. When `solution_target` is specified, `convergence_tolerance` specifies a percent error on the optimization. This is used for test problems, when the true global minimum is known (call it `solution_target` := fglobal). Then, the optimization terminates when 100(f_min-fglobal)/max(1,abs(fglobal) < convergence_tolerance. The default for fglobal is -1.0e100 and the default for convergence tolerance is as given above.

`min_boxsize_limit` is a setting that terminates the optimization when the measure of a hyperrectangle S with f(c(S)) = fmin is less than min_boxsize_limit. `volume_boxsize_limit` is a setting that terminates the optimization when the volume of a hyperrectangle S with f(c(S)) = fmin is less than volume_boxsize_limit percent of the original hyperrectangle. Basically, volume_boxsize_limit stops the optimization when the volume of the particular rectangle which has fmin is less than a certain percentage of the whole volume. `min_boxsize_limit` uses an arbitrary measure to stop the optimization. The keywords for NCSU DIRECT are described in Table 5.17 below.

Description	Keyword	Associated Data	Status	Default
Solution Target	`solution_target`	real	Optional	0
Min boxsize limit	`min_boxsize_limit`	real in [0,1]	Optional	1.0e-4
Volume boxsize limit	`volume_boxsize_limit`	real in [0,1]	Optional	1.0e-6

Table 20.15: Specification detail for the NCSU DIRECT method

20.9 JEGA Methods

The JEGA library [Eddy and Lewis, 2001] contains two global optimization methods. The first is a Multi-objective Genetic Algorithm (MOGA) which performs Pareto optimization. The second is a Single-objective Genetic Algorithm (SOGA) which performs optimization on a single objective function. Both methods support general constraints and a mixture of real and discrete variables. The JEGA library was written by John Eddy, currently a member of the technical staff in the System Readiness and Sustainment Technologies department at Sandia National Laboratories in Albuquerque. These algorithms are accessed as `moga` and `soga` within DAKOTA. DAKOTA provides access to the JEGA library through the **JEGAOptimizer** class.

20.9.1 JEGA method independent controls

JEGA utilizes the `max_iterations` and `max_function_evaluations` method independent controls to provide integer limits for the maximum number of generations and function evaluations, respectively. Note that currently, the DAKOTA default for `max_iterations` is 100 and for `max_function_evaluations` is 1000. These are the default settings that will be used to "stop" the JEGA algorithms, unless some specific convergence criteria are set (see Tables 5.20 and 5.21 below).

Beginning with v2.0, JEGA also utilizes the `output` method independent control to vary the amount of information presented to the user during execution.

20.9.2 JEGA method dependent controls

The JEGA library currently provides two types of genetic algorithms (GAs): a multi-objective genetic algorithm (`moga`), and a single- objective genetic algorithm (`soga`). Both of these GAs can take real-valued inputs, integer-valued inputs, or a mixture of real and integer-valued inputs. "Real-valued" and "integer-valued" refer to the use of continuous or discrete variable domains, respectively (the response data are real-valued in all cases).

The basic steps of the genetic algorithm are as follows:

1. Initialize the population (by randomly generating population members with or without duplicates allowed, or by flat-file initialization)
2. Evaluate the initial population members (calculate the values of the objective function(s) and constraints for each population member)
3. Perform crossover (several crossover types are available)
4. Perform mutation (several mutation types are available)
5. Evaluate the new population members.
6. Assess the fitness of each member in the population. There are a number of ways to evaluate the fitness of the members of the population. Choice of fitness assessor operators is strongly related to the type of replacement algorithm being used and can have a profound effect on the solutions selected for the next generation. For example, if using `MOGA`, the available assessors are the `layer_rank` and `domination_count` fitness assessors. If using either of these, it is strongly recommended that you use the `replacement_type` called the `below_limit` selector as well (although the roulette wheel selectors can also be used). The functionality of the domination_count selector of JEGA v1.0 can now be achieved using the `domination_count` fitness assessor and `below_limit` replacement selector together. If using `SOGA`, there are a number of possible combinations of fitness assessors and selectors.

7. Replace the population with members selected to continue in the next generation. The pool of potential members is the current population and the current set of offspring. The `replacement_type` of `roulette_wheel` or `unique_roulette_wheel` may be used either with MOGA or SOGA problems however they are not recommended for use with MOGA. Given that the only two fitness assessors for MOGA are the `layer_rank` and `domination_count`, the recommended selector is the `below_limit` selector. The `below_limit` replacement will only keep designs that are dominated by fewer than a limiting number of other designs. The `replacement_type` of `favor_feasible` is specific to a SOGA. This replacement operator will always prefer a more feasible design to a less feasible one. Beyond that, it favors solutions based on an assigned fitness value which must have been installed by the weighted sum only fitness assessor (see the discussion below).

8. Apply niche pressure to the population. This step is specific to the MOGA and is new as of JEGA v2.0. Technically, the step is carried out during runs of the SOGA but only the `null_niching` operator is available for use with SOGA. In MOGA, the `radial` or `distance` operators can be used. The purpose of niching is to encourage differentiation along the Pareto frontier and thus a more even and uniform sampling. The radial nicher takes information input from the user to compute a minimum allowable distance between designs in the performance space and acts as a secondary selection operator whereby it enforces this minimum distance. The distance nicher requires that solutions must be separated from other solutions by a minimum distance in each dimension (vs. Euclidean distance for the radial niching). After niching is complete, all designs in the population will be at least the minimum distance from one another in all directions.

9. Test for convergence. There are two aspects to convergence that must be considered. The first is stopping criteria. A stopping criteria dictates some sort of limit on the algorithm that is independent of its performance. Examples of stopping criteria available for use with JEGA are the `max_iterations` and `max_function_evaluations` inputs. All JEGA convergers respect these stopping criteria in addition to anything else that they do.

   The second aspect to convergence involves repeated assessment of the algorithms progress in solving the problem. In JEGA v1.0, the SOGA fitness tracker convergers (`best_fitness_tracker` and `average_fitness_tracker`) performed this function by asserting that the fitness values (either best or average) of the population continue to improve. There was no such operator for the MOGA. As of JEGA v2.0, the same fitness tracker convergers exist for use with SOGA and there is now a converger available for use with the MOGA. The MOGA converger (`metric_tracker`) operates by tracking various changes in the non-dominated frontier from generation to generation. When the changes occurring over a user specified number of generations fall below a user specified threshold, the algorithm stops.

10. Perform post processing. This step is new as of JEGA v2.1. The purpose of this operation is to perform any needed data manipulations on the final solution deemed necessary. Currently the `distance_postprocessor` is the only one other than the `null_postprocessor`. The `distance_postprocessor` is specifically for use with the MOGA and reduces the final solution set size such that a minimum distance in each direction exists between any two designs.

There are many controls which can be used for both MOGA and SOGA methods. These include among others the random seed, initialization types, crossover and mutation types, and some replacement types. These are described in Tables 5.18 and 5.19 below.

The `seed` control defines the starting seed for the random number generator. The algorithm uses random numbers heavily but a specification of a random seed will cause the algorithm to run identically from one trial to the next so long as all other input specifications remain the same. New as of JEGA v2.0 is the introduction of the `log_file` specification. JEGA now uses a logging library to output messages and status to the user. JEGA can be configured at build time to log to both the console window and a text file, one or the other, or neither. The `log_file` input

is a string name of a file into which to log. If the build was configured without file logging in JEGA, this input is ignored. If file logging is enabled and no `log_file` is specified, the default file name of JEGAGlobal.log is used. Also new to JEGA v2.0 is the introduction of the `print_each_pop` specification. It serves as a flag and if supplied, the population at each generation will be printed to a file named "population<GEN#>.dat" where <GEN#> is the number of the current generation.

The `initialization_type` defines the type of initialization for the GA. There are three types: `simple_random`, `unique_random`, and `flat_file`. `simple_random` creates initial solutions with random variable values according to a uniform random number distribution. It gives no consideration to any previously generated designs. The number of designs is specified by the `population_size`. `unique_random` is the same as `simple_random`, except that when a new solution is generated, it is checked against the rest of the solutions. If it duplicates any of them, it is rejected. `flat_file` allows the initial population to be read from a flat file. If `flat_file` is specified, a file name must be given. Variables can be delimited in the flat file in any way you see fit with a few exceptions. The delimiter must be the same on any given line of input with the exception of leading and trailing whitespace. So a line could look like: 1.1, 2.2 ,3.3 for example but could not look like: 1.1, 2.2 3.3. The delimiter can vary from line to line within the file which can be useful if data from multiple sources is pasted into the same input file. The delimiter can be any string that does not contain any of the characters .+-dDeE or any of the digits 0-9. The input will be read until the end of the file. The algorithm will discard any configurations for which it was unable to retrieve at least the number of design variables. The objective and constraint entries are not required but if ALL are present, they will be recorded and the design will be tagged as evaluated so that evaluators may choose not to re-evaluate them. Setting the size for this initializer has the effect of requiring a minimum number of designs to create. If this minimum number has not been created once the files are all read, the rest are created using the `unique_random` initializer and then the `simple_random` initializer if necessary.

Note that the `population_size` only sets the size of the initial population. The population size may vary in the JEGA methods according to the type of operators chosen for a particular optimization run.

There are many crossover types available. `multi_point_binary` crossover requires an integer number, N, of crossover points. This crossover type performs a bit switching crossover at N crossover points in the binary encoded genome of two designs. Thus, crossover may occur at any point along a solution chromosome (in the middle of a gene representing a design variable, for example). `multi_point_parameterized_binary` crossover is similar in that it performs a bit switching crossover routine at N crossover points. However, this crossover type performs crossover on each design variable individually. So the individual chromosomes are crossed at N locations. `multi_point_real` crossover performs a variable switching crossover routing at N crossover points in the real real valued genome of two designs. In this scheme, crossover only occurs between design variables (chromosomes). Note that the standard solution chromosome representation in the JEGA algorithm is real encoded and can handle integer or real design variables. For any crossover types that use a binary representation, real variables are converted to long integers by multiplying the real number by 10^6 and then truncating. Note that this assumes a precision of only six decimal places. Discrete variables are represented as integers (indices within a list of possible values) within the algorithm and thus require no special treatment by the binary operators.

The final crossover type is `shuffle_random`. This crossover type performs crossover by choosing design variables at random from a specified number of parents enough times that the requested number of children are produced. For example, consider the case of 3 parents producing 2 children. This operator would go through and for each design variable, select one of the parents as the donor for the child. So it creates a random shuffle of the parent design variable values. The relative numbers of children and parents are controllable to allow for as much mixing as desired. The more parents involved, the less likely that the children will wind up exact duplicates of the parents.

All crossover types take a `crossover_rate`. The crossover rate is used to calculate the number of crossover operations that take place. The number of crossovers is equal to the rate $*$ population_size.

There are five mutation types allowed. `replace_uniform` introduces random variation by first randomly choosing a design variable of a randomly selected design and reassigning it to a random valid value for that variable. No consideration of the current value is given when determining the new value. All mutation types have a `mutation_rate`. The number of mutations for the replace_uniform mutator is the product of the mutation_-rate and the population_size.

The `bit_random` mutator introduces random variation by first converting a randomly chosen variable of a randomly chosen design into a binary string. It then flips a randomly chosen bit in the string from a 1 to a 0 or visa versa. In this mutation scheme, the resulting value has more probability of being similar to the original value. The number of mutations performed is the product of the mutation_rate, the number of design variables, and the population size.

The offset mutators all act by adding an "offset" random amount to a variable value. The random amount has a mean of zero in all cases. The `offset_normal` mutator introduces random variation by adding a Gaussian random amount to a variable value. The random amount has a standard deviation dependent on the `mutation_-scale`. The `mutation_scale` is a fraction in the range [0, 1] and is meant to help control the amount of variation that takes place when a variable is mutated. `mutation_scale` is multiplied by the range of the variable being mutated to serve as standard deviation. `offset_cauchy` is similar to `offset_normal`, except that a Cauchy random variable is added to the variable being mutated. The `mutation_scale` also defines the standard deviation for this mutator. Finally, `offset_uniform` adds a uniform random amount to the variable value. For the `offset_uniform` mutator, the `mutation_scale` is interpreted as a fraction of the total range of the variable. The range of possible deviation amounts is +/- 1/2 $*$ (`mutation_scale` $*$ variable range). The number of mutations for all offset mutators is defined as the product of `mutation_rate` and `population_-size`.

As of JEGA v2.0, all replacement types are common to both MOGA and SOGA. They include the `roulette_-wheel`, `unique_roulette_wheel`, `elitist`, and `below_limit` selectors. In roulette_wheel replacement, each design is conceptually allotted a portion of a wheel proportional to its fitness relative to the fitnesses of the other Designs. Then, portions of the wheel are chosen at random and the design occupying those portions are duplicated into the next population. Those Designs allotted larger portions of the wheel are more likely to be selected (potentially many times). `unique_roulette_wheel` replacement is the same as `roulette_-wheel` replacement, with the exception that a design may only be selected once. The `below_limit` selector attempts to keep all designs for which the negated fitness is below a certain limit. The values are negated to keep with the convention that higher fitness is better. The inputs to the `below_limit` selector are the limit as a real value, and a `shrinkage_percentage` as a real value. The `shrinkage_percentage` defines the minimum amount of selections that will take place if enough designs are available. It is interpreted as a percentage of the population size that must go on to the subsequent generation. To enforce this, `below_limit` makes all the selections it would make anyway and if that is not enough, it takes the remaining that it needs from the best of what is left (effectively raising its limit as far as it must to get the minimum number of selections). It continues until it has made enough selections. The `shrinkage_percentage` is designed to prevent extreme decreases in the population size at any given generation, and thus prevent a big loss of genetic diversity in a very short time. Without a shrinkage limit, a small group of "super" designs may appear and quickly cull the population down to a size on the order of the limiting value. In this case, all the diversity of the population is lost and it is expensive to re-diversify and spread the population. The `elitist` selector simply chooses the required number of designs taking the most fit. For example, if 100 selections are requested, then the top 100 designs as ranked by fitness will be selected and the remaining will be discarded.

Description	Keyword	Associated Data	Status	Default
GA Method	moga | soga	none	Required group	N/A
Random seed	seed	integer	Optional	Time based seed
Log file	log_file	string	Optional	JEGAGlobal.log
Number of population members	population_-size	integer	Optional	50
Population output	print_each_-pop	none	Optional	No printing
Output verbosity	output	silent | quiet | verbose | debug	Optional	normal
Initialization type	initialization_-type	simple_-random | unique_-random | flat_file	Optimal	unique_random
Mutation type	mutation_-type	replace_-uniform | bit_random | offset_-cauchy | offset_-uniform | offset_-normal	Optional group	replace_uniform
Mutation scale	mutation_-scale	real	Optional	0.15
Mutation rate	mutation_-rate	real	Optional	0.08
Replacement type	replacement_-type	below_limit | roulette_-wheel | unique_-roulette_-wheel | elitist	Required group	None
Below limit selection	below_limit	real	Optional	6
Shrinkage percentage in below limit selection	shrinkage_-percentage	real	Optional	0.9

Table 20.16: Specification detail for JEGA method dependent controls: seed, output, initialization, mutation, and replacement

Description	Keyword	Associated Data	Status	Default
Crossover type	crossover_type	multi_point_binary | multi_point_parameterized_binary | multi_point_real | shuffle_random	Optional group	shuffle_random
Multi point binary crossover	multi_point_binary	integer	Required	N/A
Multi point parameterized binary crossover	multi_point_parameterized_binary	integer	Required	N/A
Multi point real crossover	multi_point_real	integer	Required	N/A
Random shuffle crossover	shuffle_random	num_parents, num_offspring	Required	N/A
Number of parents in random shuffle crossover	num_parents	integer	optional	2
Number of offspring in random shuffle crossover	num_offspring	integer	optional	2
Crossover rate	crossover_rate	real	optional (applies to all crossover types)	0.8

Table 20.17: Specification detail for JEGA method dependent controls: crossover

20.9.3 Multi-objective Evolutionary Algorithms

The specification for controls specific to Multi-objective Evolutionary algorithms are described here. These controls will be appropriate to use if the user has specified `moga` as the method.

The initialization, crossover, and mutation controls were all described in the preceding section. There are no MOGA specific aspects to these controls. The `fitness_type` for a MOGA may be `domination_count` or `layer_rank`. Both have been specifically designed to avoid problems with aggregating and scaling objective function values and transforming them into a single objective. Instead, the `domination_count` fitness assessor works by ordering population members by the negative of the number of designs that dominate them. The values are negated in keeping with the convention that higher fitness is better. The `layer_rank` fitness assessor works by assigning all non-dominated designs a layer of 0, then from what remains, assigning all the non-dominated a layer of 1, and so on until all designs have been assigned a layer. Again, the values are negated for the higher-is-better fitness convention. Use of the `below_limit` selector with the `domination_count` fitness assessor has the effect of keeping all designs that are dominated by fewer then a limiting number of other designs subject to the shrinkage limit. Using it with the `layer_rank` fitness assessor has the effect of keeping all those designs whose layer is below a certain threshold again subject to the shrinkage limit.

New as of JEGA v2.0 is the introduction of niche pressure operators. These operators are meant primarily for use with the moga. The job of a niche pressure operator is to encourage diversity along the Pareto frontier as the algorithm runs. This is typically accomplished by discouraging clustering of design points in the performance space. In JEGA, the application of niche pressure occurs as a secondary selection operation. The nicher is given a chance to perform a pre-selection operation prior to the operation of the selection (replacement) operator, and is then called to perform niching on the set of designs that were selected by the selection operator.

Currently, the only niche pressure operators available are the `radial` nicher and the `distance` nicher. The radial niche pressure applicator works by enforcing a minimum Euclidean distance between designs in the performance space at each generation. The algorithm proceeds by starting at the (or one of the) extreme designs along objective dimension 0 and marching through the population removing all designs that are too close to the current design. One exception to the rule is that the algorithm will never remove an extreme design which is defined as a design that is maximal or minimal in all but 1 objective dimension (for a classical 2 objective problem, the extreme designs are those at the tips of the non-dominated frontier). The `distance` nicher enforces a minimimum distance in each dimension.

The designs that are removed by the nicher are not discarded. They are buffered and re-inserted into the population during the next pre-selection operation. This way, the selector is still the only operator that discards designs and the algorithm will not waste time "re-filling" gaps created by the nicher.

The radial nicher requires as input a vector of fractions with length equal to the number of objectives. The elements of the vector are interpreted as percentages of the non-dominated range for each objective defining a minimum distance to all other designs. All values should be in the range (0, 1). The minimum allowable distance between any two designs in the performance space is the Euclidian (simple square-root-sum-of-squares calculation) distance defined by these percentages. The distance nicher has a similar input vector requirement, only the distance is the minimum distance in each dimension.

Also new as of JEGA v2.0 is the introduction of the MOGA specific `metric_tracker` converger. This converger is conceptually similar to the best and average fitness tracker convergers in that it tracks the progress of the population over a certain number of generations and stops when the progress falls below a certain threshold. The implementation is quite different however. The `metric_tracker` converger tracks 3 metrics specific to the non-dominated frontier from generation to generation. All 3 of these metrics are computed as percent changes between the generations. In order to compute these metrics, the converger stores a duplicate of the non-dominated frontier at each generation for comparison to the non-dominated frontier of the next generation.

The first metric is one that indicates how the expanse of the frontier is changing. The expanse along a given objective is defined by the range of values existing within the non-dominated set. The expansion metric is computed by tracking the extremes of the non-dominated frontier from one generation to the next. Any movement of the extreme values is noticed and the maximum percentage movement is computed as:

```
Em = max over j of abs((range(j, i) - range(j, i-1)) / range(j, i-1))  j=1,nof
```

where Em is the max expansion metric, j is the objective function index, i is the current generation number, and nof is the total number of objectives. The range is the difference between the largest value along an objective and the smallest when considering only non-dominated designs.

The second metric monitors changes in the density of the non-dominated set. The density metric is computed as the number of non-dominated points divided by the hypervolume of the non-dominated region of space. Therefore, changes in the density can be caused by changes in the number of non-dominated points or by changes in size of the non-dominated space or both. The size of the non-dominated space is computed as:

```
Vps(i) = product over j of range(j, i)   j=1,nof
```

where Vps(i) is the hypervolume of the non-dominated space at generation i and all other terms have the same meanings as above.

The density of the a given non-dominated space is then:

```
Dps(i) = Pct(i) / Vps(i)
```

where Pct(i) is the number of points on the non-dominated frontier at generation i.

The percentage increase in density of the frontier is then calculated as

```
Cd = abs((Dps(i) - Dps(i-1)) / Dps(i-1))
```

where Cd is the change in density metric.

The final metric is one that monitors the "goodness" of the non-dominated frontier. This metric is computed by considering each design in the previous population and determining if it is dominated by any designs in the current population. All that are determined to be dominated are counted. The metric is the ratio of the number that are dominated to the total number that exist in the previous population.

As mentioned above, each of these metrics is a percentage. The tracker records the largest of these three at each generation. Once the recorded percentage is below the supplied percent change for the supplied number of generations consecutively, the algorithm is converged.

The specification for convergence in a moga can either be `metric_tracker` or can be omitted all together. If omitted, no convergence algorithm will be used and the algorithm will rely on stopping criteria only. If `metric_tracker` is specified, then a `percent_change` and `num_generations` must be supplied as with the other metric tracker convergers (average and best fitness trackers). The `percent_change` is the threshold beneath which convergence is attained whereby it is compared to the metric value computed as described above. The `num_generations` is the number of generations over which the metric value should be tracked. Convergence will be attained if the recorded metric is below `percent_change` for `num_generations` consecutive generations.

The MOGA specific controls are described in Table 5.20 below. Note that MOGA and SOGA create additional output files during execution. "finaldata.dat" is a file that holds the final set of Pareto optimal solutions after any post-processing is complete. "discards.dat" holds solutions that were discarded from the population during the course of evolution. It can often be useful to plot objective function values from these files to visually see the

Pareto front and ensure that finaldata.dat solutions dominate discards.dat solutions. The solutions are written to these output files in the format "Input1...InputN..Output1...OutputM". If MOGA is used in a hybrid optimization strategy (which requires one optimal solution from each individual optimization method to be passed to the subsequent optimization method as its starting point), the solution in the Pareto set closest to the "utopia" point is given as the best solution. This solution is also reported in the DAKOTA output. This "best" solution in the Pareto set has minimum distance from the utopia point. The utopia point is defined as the point of extreme (best) values for each objective function. For example, if the Pareto front is bounded by (1,100) and (90,2), then (1,2) is the utopia point. There will be a point in the Pareto set that has minimum L2-norm distance to this point, for example (10,10) may be such a point. In SOGA, the solution that minimizes the single objective function is returned as the best solution. If moga is used in a strategy which may require passing multiple solutions to the next level (such as the `surrogate_based_global` method or `hybrid` strategy), the `orthogonal_distance` postprocessor type may be used to specify the distances between each solution value to winnow down the solutions in the full Pareto front to a subset which will be passed to the next iteration.

Description	Keyword	Associated Data	Status	Default
Fitness type	`fitness_type`	`layer_rank` | `domination_count`	Required group	None
Niche pressure type	`niching_type`	`radial` | `distance`	Optional group	No niche pressure
Niching distance	`radial` | `distance`	list of real	Optional	0.01 for all objectives
Convergence type	`metric_tracker`	none	Optional group	Stopping criteria only
Percent change limit for metric_tracker converger	`percent_change`	real	Optional	0.1
Number generations for metric_tracker converger	`num_generations`	integer	Optional	10
Post_processor type	`postprocessor_type`	`orthogonal_distance`	Optional	No post-processing of solutions
Post_processor distance	`orthogonal_distance`	`list` of real	Optional	0.01 for all objectives

Table 20.18: Specification detail for MOGA method controls

20.9.4 Single-objective Evolutionary Algorithms

The specification for controls specific to Single-objective Evolutionary algorithms are described here. These controls will be appropriate to use if the user has specified `soga` as the method.

The initialization, crossover, and mutation controls were all described above. There are no SOGA specific aspects to these controls. The `replacement_type` for a SOGA may be `roulette_wheel`, `unique_roulette_wheel`, `elitist`, or `favor_feasible`. The `favor_feasible` replacement type first considers feasibility as a selection criteria. If that does not produce a "winner" then it moves on to considering fitness

value. Because of this, any fitness assessor used with the `favor_feasible` selector must only account objectives in the creation of fitness. Therefore, there is such a fitness assessor and it's use is enforced when the \ favor_feasible selector is chosen. In that case, and if the output level is set high enough, a message will be presented indicating that the `weighted_sum_only` fitness assessor will be used. As of JEGA v2.0 and beyond, the fitness assessment operator must be specified with SOGA although the `merit_function` is currently the only one (note that the `weighted_sum_only` assessor exists but cannot be selected). The roulette wheel selectors no longer assume a fitness function. The `merit_function` fitness assessor uses an exterior penalty function formulation to penalize infeasible designs. The specification allows the input of a `constraint_penalty` which is the multiplier to use on the constraint violations.

The SOGA controls allow two additional convergence types. The `convergence_type` called `average_fitness_tracker` keeps track of the average fitness in a population. If this average fitness does not change more than `percent_change` over some number of generations, `num_generations`, then the solution is reported as converged and the algorithm terminates. The `best_fitness_tracker` works in a similar manner, only it tracks the best fitness in the population. Convergence occurs after `num_generations` has passed and there has been less than `percent_change` in the best fitness value. The percent change can be as low as 0% in which case there must be no change at all over the number of generations. Both also respect the stopping criteria.

The SOGA specific controls are described in Table 5.21 below.

Description	Keyword	Associated Data	Status	Default
Fitness type	`fitness_type`	`merit_function`	Optional group	merit_function
Constraint penalty in merit function	`constraint_penalty`	`real`	Optional	1.0
Replacement type	`replacement_type`	`favor_feasible` | `unique_roulette_wheel` | `roulette_wheel`	Required group	None
Convergence type	`convergence_type`	`best_fitness_tracker` | `average_fitness_tracker`	Optional	None
Number of generations (for convergence test)	`num_generations`	integer	Optional	10
Percent change in fitness	`percent_change`	real	Optional	0.1

Table 20.19: Specification detail for SOGA method controls

Chapter 21

Least Squares Methods

DAKOTA's least squares branch currently contains three methods for solving nonlinear least squares problems: NL2SOL, a trust-region method that adaptively chooses between two Hessian approximations (Gauss-Newton and Gauss-Newton plus a quasi-Newton approximation to the rest of the Hessian), NLSSOL, a sequential quadratic programming (SQP) approach that is from the same algorithm family as NPSOL, and Gauss-Newton, which supplies the Gauss-Newton Hessian approximation to the full-Newton optimizers from OPT++.

The important difference of these algorithms from general-purpose optimization methods is that the response set is defined by calibration terms (e.g. separate terms for each residual), rather than an objective function. Thus, a finer granularity of data is used by least squares solvers as compared to that used by optimizers. This allows the exploitation of the special structure provided by a sum of squares objective function. Refer to Calibration terms and constraint functions (least squares data set) for additional information on the least squares response data set.

21.1 NL2SOL Method

NL2SOL is available as `nl2sol` and addresses unconstrained and bound-constrained problems. It uses a trust-region method (and thus can be viewed as a generalization of the Levenberg-Marquardt algorithm) and adaptively chooses between two Hessian approximations, the Gauss-Newton approximation alone and the Gauss-Newton approximation plus a quasi-Newton approximation to the rest of the Hessian. Even on small-residual problems, the latter Hessian approximation can be useful when the starting guess is far from the solution. On problems that are not over-parameterized (i.e., that do not involve more optimization variables than the data support), NL2SOL usually exhibits fast convergence.

NL2SOL has a variety of internal controls as described in AT&T Bell Labs CS TR 153 (`http://cm.bell-labs.com/cm/cs/cstr/153.ps.gz`). A number of existing DAKOTA controls (method independent controls and responses controls) are mapped into these NL2SOL internal controls. In particular, DAKOTA's `convergence_tolerance`, `max_iterations`, `max_function_evaluations`, and `fd_gradient_step_size` are mapped directly into NL2SOL's `rfctol`, `mxiter`, `mxfcal`, and `dltfdj` controls, respectively. In addition, DAKOTA's `fd_hessian_step_size` is mapped into both `delta0` and `dltfdc`, and DAKOTA's `output` verbosity is mapped into NL2SOL's `auxprt` and `outlev` (for `normal/verbose/debug output, NL2SOL` prints initial guess, final solution, solution statistics, nondefault values, and changes to the active bound constraint set on every iteration; for `quiet output`, NL2SOL prints only the initial guess and final solution; and for `silent output`, NL2SOL output is suppressed).

Several NL2SOL convergence tolerances are adjusted in response to `function_precision`, which gives the relative precision to which responses are computed. These tolerances may also be specified explicitly: `convergence_tolerance` (NL2SOL's `rfctol`, as mentioned previously) is the relative-function convergence tolerance (on the accuracy desired in the sum-of-squares function); `x_conv_tol` (NL2SOL's `xctol`) is the X-convergence tolerance (scaled relative accuracy of the solution variables); `absolute_conv_tol` (NL2SOL's `afctol`) is the absolute function convergence tolerance (stop when half the sum of squares is less than `absolute_conv_tol`, which is mainly of interest on zero-residual test problems); `singular_conv_tol` (NL2SOL's `sctol`) is the singular convergence tolerance, which works in conjunction with `singular_radius` (NL2SOL's `lmaxs`) to test for underdetermined least-squares problems (stop when the relative reduction yet possible in the sum of squares appears less then `singular_conv_tol` for steps of scaled length at most `singular_radius`); `false_conv_tol` (NL2SOL's `xftol`) is the false-convergence tolerance (stop with a suspicion of discontinuity when a more favorable stopping test is not satisfied and a step of scaled length at most `false_conv_tol` is not accepted). Finally, the `initial_trust_radius` specification (NL2SOL's `lmax0`) specifies the initial trust region radius for the algorithm.

The internal NL2SOL defaults can be obtained for many of these controls by specifying the value -1. For both the `singular_radius` and the `initial_trust_radius`, this results in the internal use of steps of length 1. For other controls, the internal defaults are often functions of machine epsilon (as limited by `function_precision`). Refer to CS TR 153 for additional details on these formulations.

Whether and how NL2SOL computes and prints a final covariance matrix and regression diagnostics is affected by several keywords. `covariance` (NL2SOL's `covreq`) specifies the desired covariance approximation:

- 0 = default = none
- 1 or -1 ==> $\sigma^2 H^{-1} J^T J H^{-1}$
- 2 or -2 ==> $\sigma^2 H^{-1}$
- 3 or -3 ==> $\sigma^2 (J^T J)^{-1}$
- Negative values ==> estimate the final Hessian H by finite differences of function values only (using `fd_hessian_step_size`)
- Positive values ==> differences of gradients (using `fd_hessian_step_size`)

When `regression_diagnostics` (NL2SOL's `rdreq`) is specified and a positive-definite final Hessian approximation H is computed, NL2SOL computes and prints a regression diagnostic vector RD such that if omitting the i-th observation would cause alpha times the change in the solution that omitting the j-th observation would cause, then RD[i] = |alpha| RD[j]. The finite-difference step-size tolerance affecting H is `fd_hessian_step_size` (NL2SOL's `delta0` and `dltfdc`, as mentioned previously).

Table 5.22 provides the specification detail for the NL2SOL method dependent controls.

21.2 NLSSOL Method

NLSSOL is available as `nlssol_sqp` and supports unconstrained, bound-constrained, and generally-constrained problems. It exploits the structure of a least squares objective function through the periodic use of Gauss-Newton Hessian approximations to accelerate the SQP algorithm. DAKOTA provides access to the NLSSOL library through the **NLSSOLLeastSq** class. The method independent and method dependent controls are identical to those of NPSOL as described in NPSOL method independent controls and NPSOL method dependent controls.

Description	Keyword	Associated Data	Status	Default
Relative precision in least squares terms	`function_precision`	real	Optional	1e-10
Absolute function convergence tolerance	`absolute_conv_tol`	real	Optional	-1. (use NL2SOL internal default)
Convergence tolerance for change in parameter vector	`x_conv_tol`	real	Optional	-1. (use NL2SOL internal default)
Singular convergence tolerance	`singular_conv_tol`	real	Optional	-1. (use NL2SOL internal default)
Step limit for `sctol`	`singular_radius`	real	Optional	-1. (use NL2SOL internal default of 1)
False convergence tolerance	`false_conv_tol`	real	Optional	-1. (use NL2SOL internal default)
Initial trust region radius	`initial_trust_radius`	real	Optional	-1. (use NL2SOL internal default of 1)
Covariance post-processing	`covariance`	integer	Optional	0 (no covariance)
Regression diagnostics post-processing	`regression_diagnostics`	none	Optional	no regression diagnostics

Table 21.1: Specification detail for NL2SOL method dependent controls.

21.3 Gauss-Newton Method

The Gauss-Newton algorithm is available as `optpp_g_newton` and supports unconstrained, bound-constrained, and generally-constrained problems. The code for the Gauss-Newton approximation (objective function value, gradient, and approximate Hessian defined from residual function values and gradients) is provided outside of OPT++ within **SNLLLeastSq::nlf2_evaluator_gn()**. When interfaced with the unconstrained, bound-constrained, and nonlinear interior point full-Newton optimizers from the OPT++ library, it provides a Gauss-Newton least squares capability which -- on zero-residual test problems -- can exhibit quadratic convergence rates near the solution. (Real problems almost never have zero residuals, i.e., perfect fits.)

Mappings for the method independent and dependent controls are the same as for the OPT++ optimization methods and are as described in OPT++ method independent controls and OPT++ method dependent controls. In particular, since OPT++ full-Newton optimizers provide the foundation for Gauss-Newton, the specifications from Table 5.6 are also applicable for `optpp_g_newton`.

Chapter 22

Surrogate-Based Minimization Methods

In surrogate-based optimization (SBO) and surrogate-based nonlinear least squares (SBNLS), minimization occurs using a set of one or more approximations, defined from a surrogate model, that are built and periodically updated using data from a "truth" model. The surrogate model can be a global data fit (e.g., regression or interpolation of data generated from a design of computer experiments), a multipoint approximation, a local Taylor Series expansion, or a model hierarchy approximation (e.g., a low-fidelity simulation model), whereas the truth model involves a high-fidelity simulation model. The goals of surrogate-based methods are to reduce the total number of truth model simulations and, in the case of global data fit surrogates, to smooth noisy data with an easily navigated analytic function.

22.1 Surrogate-Based Local Method

In the surrogate-based local (SBL) method, a trust region approach is used to manage the minimization process to maintain acceptable accuracy between the surrogate model and the truth model (by limiting the range over which the surrogate model is trusted). The process involves a sequence of minimizations performed on the surrogate model and bounded by the trust region. At the end of each approximate minimization, the candidate optimum point is validated using the truth model. If sufficient decrease has been obtained in the truth model, the trust region is re-centered around the candidate optimum point and the trust region will either shrink, expand, or remain the same size depending on the accuracy with which the surrogate model predicted the truth model decrease. If sufficient decrease has not been attained, the trust region center is not updated and the entire trust region shrinks by a user-specified factor. The cycle then repeats with the construction of a new surrogate model, a minimization, and another test for sufficient decrease in the truth model. This cycle continues until convergence is attained.

The `surrogate_based_local` method must specify an optimization or least squares sub-method either by pointer using `approx_method_pointer` (e.g., 'NLP1') or by name using `approx_method_name` (e.g., 'npsol_sqp'). The former identifies a full sub-method specification for the sub-problem minimizer (which allows non-default minimizer settings), whereas the latter supports a streamlined specification (that employs default minimizer settings). For both cases, the `surrogate_based_local` method specification is responsible for using its `model_pointer` (see Method Independent Controls) to select a `surrogate` model (see Surrogate Model Controls). Any `model_pointer` identified in an approximate sub-method specification is ignored.

In addition to the method independent controls for `max_iterations` and `convergence_tolerance` described in Table 5.1, SBL algorithm controls include `soft_convergence_limit` (a soft convergence control for the SBL iterations which limits the number of consecutive iterations with improvement less than the con-

vergence tolerance) and `truth_surrogate_bypass` (a flag for bypassing all lower level surrogates when performing truth verifications on a top level surrogate). Table 5.23 summarizes these SBL inputs.

Description	Keyword	Associated Data	Status	Default
Surrogate-based local method	`surrogate_-based_local`	none	Required group	N/A
Approximate sub-problem minimization method pointer	`approx_-method_-pointer`	string	Required (1 of 2 selections)	N/A
Approximate sub-problem minimization method name	`approx_-method_name`	string	Required (1 of 2 selections)	N/A
Soft convergence limit for SBL iterations	`soft_-convergence_-limit`	integer	Optional	5
Flag for bypassing lower level surrogates in truth verifications	`truth_-surrogate_-bypass`	none	Optional	no bypass

Table 22.1: Specification detail for surrogate-based local minimization method

The `trust_region` optional group specification can be used to specify the initial size of the trust region (using `initial_size`) relative to the total variable bounds, the minimum size of the trust region (using `minimum_-size`), the contraction factor for the trust region size (using `contraction_factor`) used when the surrogate model is performing poorly, and the expansion factor for the trust region size (using `expansion_factor`) used when the the surrogate model is performing well. Two additional commands are the trust region size contraction threshold (using `contract_threshold`) and the trust region size expansion threshold (using `expand_-threshold`). These two commands are related to what is called the trust region ratio, which is the actual decrease in the truth model divided by the predicted decrease in the truth model in the current trust region. The command `contract_threshold` sets the minimum acceptable value for the trust region ratio, i.e., values below this threshold cause the trust region to shrink for the next SBL iteration. The command `expand_-threshold` determines the trust region value above which the trust region will expand for the next SBL iteration. Table 5.24 summarizes these trust region inputs.

For SBL problems with nonlinear constraints, a number of algorithm formulations exist as described in [Eldred and Dunlavy, 2006] and as summarized in the Advanced Examples section of the Models chapter of the Users Manual [Adams et al., 2010]. First, the "primary" functions (that is, the objective functions or calibration terms) in the approximate subproblem can be selected to be surrogates of the original primary functions (`original_-primary`), a single objective function (`single_objective`) formed from the primary function surrogates, or either an augmented Lagrangian merit function (`augmented_lagrangian_objective`) or a Lagrangian merit function (`lagrangian_objective`) formed from the primary and secondary function surrogates. The former option may imply the use of a nonlinear least squares method, a multiobjective optimization method, or a single objective optimization method to solve the approximate subproblem, depending on the definition of the primary functions. The latter three options all imply the use of a single objective optimization method regardless of primary function definition. Second, the surrogate constraints in the approximate subproblem can be selected to be surrogates of the original constraints (`original_constraints`) or linearized approximations to the surrogate constraints (`linearized_constraints`), or constraints can be omitted from the subproblem (`no_constraints`). Following optimization of the approximate subproblem, the candidate iterate is evaluated

Description	Keyword	Associated Data	Status	Default
Trust region group specification	trust_region	none	Optional group	N/A
Trust region initial size (relative to bounds)	initial_size	real	Optional	0.4
Trust region minimum size	minimum_size	real	Optional	1.e-6
Shrink trust region if trust region ratio is below this value	contract_-threshold	real	Optional	0.25
Expand trust region if trust region ratio is above this value	expand_-threshold	real	Optional	0.75
Trust region contraction factor	contraction_-factor	real	Optional	0.25
Trust region expansion factor	expansion_-factor	real	Optional	2.0

Table 22.2: Specification detail for trust region controls in surrogate-based local methods

using a merit function, which can be selected to be a simple penalty function with penalty ramped by SBL iteration number (penalty_merit), an adaptive penalty function where the penalty ramping may be accelerated in order to avoid rejecting good iterates which decrease the constraint violation (adaptive_penalty_merit), a Lagrangian merit function which employs first-order Lagrange multiplier updates (lagrangian_merit), or an augmented Lagrangian merit function which employs both a penalty parameter and zeroth-order Lagrange multiplier updates (augmented_lagrangian_merit). When an augmented Lagrangian is selected for either the subproblem objective or the merit function (or both), updating of penalties and multipliers follows the approach described in [Conn et al., 2000]. Following calculation of the merit function for the new iterate, the iterate is accepted or rejected and the trust region size is adjusted for the next SBL iteration. Iterate acceptance is governed either by a trust region ratio (tr_ratio) formed from the merit function values or by a filter method (filter); however, trust region resizing logic is currently based only on the trust region ratio. For infeasible iterates, constraint relaxation can be used for balancing constraint satisfaction and progress made toward an optimum. The command constraint_relax followed by a method name specifies the type of relaxation to be used. Currently, homotopy [Perez et al., 2004] is the only available method for constraint relaxation, and this method is dependent on the presence of the NPSOL library within the DAKOTA executable. Table 5.25 summarizes these constraint management inputs.

22.2 Surrogate-Based Global Method

The surrogate_based_global method differs from the surrogate_based_local method in a few ways. First, surrogate_based_global is not a trust region method. Rather, surrogate_based_-global works in an iterative scheme where optimization is performed on a global surrogate using the same bounds during each iteration. In one iteration, the optimal solutions of the surrogate model are found, and then a selected set of these optimal surrogate solutions are passed to the next iteration. At the next iteration, these

Description	Keyword	Associated Data	Status	Default
Approximate subproblem formulation	approx_-subproblem	original_-primary | single_-objective | augmented_-lagrangian_-objective | lagrangian_-objective original_-constraints | linearized_-constraints | no_-constraints	Optional group	original_-primary original_-constraints
SBL merit function	merit_-function	penalty_-merit | adaptive_-penalty_-merit | lagrangian_-merit | augmented_-lagrangian_-merit	Optional group	augmented_-lagrangian_-merit
SBL iterate acceptance logic	acceptance_-logic	tr_ratio | filter	Optional group	filter
SBL constraint relaxation method for infeasible iterates	constraint_-relax	homotopy	Optional group	no relaxation

Table 22.3: Specification detail for constraint management in surrogate-based local methods

surrogate points are evaluated with the "truth" model, and then these points are added back to the set of points upon which the next surrogate is constructed. In this way, the optimization acts on a more accurate surrogate during each iteration, presumably driving to optimality quickly. This approach has no guarantee of convergence. It was originally designed for MOGA (a multi-objective genetic algorithm). Since genetic algorithms often need thousands or tens of thousands of points to produce optimal or near-optimal solutions, the use of surrogates can be helpful for reducing the truth model evaluations. Instead of creating one set of surrogates for the individual objectives and running the optimization algorithm on the surrogate once, the idea is to select points along the (surrogate) Pareto frontier, which can be used to supplement the existing points. In this way, one does not need to use many points initially to get a very accurate surrogate. The surrogate becomes more accurate as the iterations progress. Note that the user has the option of appending the optimal points from the surrogate model to the current set of truth points or using the optimal points from the surrogate model to replace the optimal set of points from the previous iteration. Although appending to the set is the default behavior, at this time we strongly recommend using the option `replace_points` because it appears to be more accurate and robust. Finally, the number of best solutions that will be passed from one iteration to another is governed by the iterator control `final_solutions`. If this is not specified, the surrogate-based global method will take all of the solutions available (e.g. all of the solutions in the Pareto front).

As for the `surrogate_based_local` method, the `surrogate_based_global` specification must identify a sub-method using either `approx_method_pointer` or `approx_method_name` and must identify a surrogate model (see Surrogate Model Controls) using its `model_pointer` (see Method Independent Controls). The only other algorithm control at this time is the method independent control for `max_iterations` described in Table 5.1. Table 5.26 summarizes the method dependent surrogate based global inputs.

Description	Keyword	Associated Data	Status	Default
Surrogate-based global method	`surrogate_based_global`	none	Required group	N/A
Approximate sub-problem minimization method pointer	`approx_method_pointer`	string	Required (1 of 2 selections)	N/A
Approximate sub-problem minimization method name	`approx_method_name`	string	Required (1 of 2 selections)	N/A
Replace points used in surrogate construction with best points from previous iteration	`replace_points`	none	Optional	Points appended, not replaced

Table 22.4: Specification detail for the surrogate-based global method

We have two cautionary notes before using the surrogate-based global method:

- One might first try a single minimization method coupled with a surrogate model prior to using the surrogate-based global method. This is essentially equivalent to setting `max_iterations` to 1 and will allow one to get a sense of what surrogate types are the most accurate to use for the problem. (Also note that one can specify that surrogates be built for all primary functions and constraints or for only a subset of these functions and constraints. This allows one to use a "truth" model directly for some of the response functions, perhaps due to them being much less expensive than other functions. This is outlined in Surrogate Model Controls.)

- We initially recommend a small number of maximum iterations, such as 3-5, to get a sense of how the optimization is evolving as the surrogate gets updated. If it appears to be changing significantly, then a larger number (used in combination with restart) may be needed.

22.3 Efficient Global Method

The Efficient Global Optimization (EGO) method was first developed by Jones, Schonlau, and Welch [Jones et al., 1998]. In EGO, a stochastic response surface approximation for the objective function is developed based on some sample points from the "true" simulation. The particular response surface used is a Gaussian process (GP). The GP allows one to calculate the prediction at a new input location as well as the uncertainty associated with that prediction. The key idea in EGO is to maximize the Expected Improvement Function (EIF). The EIF is used to select the location at which a new training point should be added to the Gaussian process model by maximizing the amount of improvement in the objective function that can be expected by adding that point. A point could be expected to produce an improvement in the objective function if its predicted value is better than the current best solution, or if the uncertainty in its prediction is such that the probability of it producing a better solution is high. Because the uncertainty is higher in regions of the design space with few observations, this provides a balance between exploiting areas of the design space that predict good solutions, and exploring areas where more information is needed. EGO trades off this "exploitation vs. exploration." The general procedure for these EGO-type methods is:

- Build an initial Gaussian process model of the objective function
- Find the point that maximizes the EIF. If the EIF value at this point is sufficiently small, stop.
- Evaluate the objective function at the point where the EIF is maximized. Update the Gaussian process model using this new point. Return to the previous step.

Note that several major differences exist between our implementation and that of [Jones et al., 1998]. First, rather than using a branch and bound method to find the point which maximizes the EIF, we use the DIRECT global optimization method (see DIRECT and NCSU Methods). Second, we support both global optimization and global nonlinear least squares as well as general nonlinear constraints through abstraction and subproblem recasting within the **SurrBasedMinimizer** and **EffGlobalMinimizer** classes.

The efficient global method is in prototype form. Currently, we do not expose any specification controls for the underlying Gaussian process model used or for the optimization of the expected improvement function (which is currently performed by the NCSU DIRECT algorithm using its internal defaults). The efficient global algorithm is specified by the keyword `efficient_global` along with an optional `seed` specification, as shown in in Table 5.27 below. By default, EGO uses the Surfpack GP (Kriging) model, but the DAKOTA implementation may be selected instead. If `use_derivatives` is specified the GP model will be built using available derivative data (Surfpack GP only).

Description	Keyword	Associated Data	Status	Default
Efficient global method	`efficient_global`	none	Required group	N/A
Random seed	`seed`	integer	Optional	Time based seed: nonrepeatable
GP selection	`gaussian_process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_derivatives`	none	Optional	Use function values only

Table 22.5: Specification detail for the efficient global method

Chapter 23

Uncertainty Quantification Methods

DAKOTA provides a variety of methods for propagating uncertainty. Aleatory uncertainty refers to inherent variability, irreducible uncertainty, or randomness, and is addressed with the probabilistic methods described in Aleatory Uncertainty Quantification Methods. Epistemic uncertainty refers to subjective uncertainty, reducible uncertainty, model form uncertainty, or uncertainty due to lack of knowledge, and is addressed using the non-probabilistic approaches described in Epistemic Uncertainty Quantification Methods. In general, we refer to both classes of uncertainty quantification methods in DAKOTA as nondeterministic methods. In the descriptions below, we described the issues and specification controls that are common to both aleatory and epistemic uncertainty quantification.

DAKOTA's nondeterministic methods make use of a few method independent controls. `max_iterations` and `convergence_tolerance` are used by iterated local and global reliability, stochastic expansions undergoing automated refinement, and optimization-based epistemic methods. `output` level controls are also employed by several methods:

- optional output of probability density functions (PDFs), for settings of `normal` or higher (sampling methods, including numerical statistics for stochastic expansion methods)
- optional output of sample evaluations performed on approximations, for settings of `verbose` or higher (stochastic expansion methods)
- optional tabular output of numerical integration points and weights, for settings of `verbose` or higher (stochastic expansion methods)
- optional output of local sensitivities computed from global approximations, for settings of `normal` or higher (stochastic expansion methods)
- optional output of statistical quantities of interest (QOI) for iterations prior to convergence of an automated refinement process (stochastic expansion methods with a `debug` setting)

Each of the uncertainty quantification techniques is standardized on support for `response_levels`, `probability_levels`, `reliability_levels`, and `gen_reliability_levels` specifications along with their optional `num_response_levels`, `num_probability_levels`, `num_reliability_levels` and `num_gen_reliability_levels` keys, except for a few exceptions where certain level mappings cannot be supported: `global_reliability`, `importance`, `local_evidence`, and `global_evidence` do not support mappings involving `reliability_levels`, and `local_interval_est` and `global_interval_est` do not support any level mappings. The keys define the distribution of the levels among the different response functions. For example, the following specification

```
num_response_levels = 2 4 3
response_levels = 1. 2. .1 .2 .3 .4 10. 20. 30.
```

would assign the first two response levels (1., 2.) to response function 1, the next four response levels (.1, .2, .3, .4) to response function 2, and the final three response levels (10., 20., 30.) to response function 3. If the `num_response_levels` key were omitted from this example, then the response levels would be evenly distributed among the response functions (three levels each in this case).

The `response_levels` specification provides the target response values for generating probabilities, reliabilities, or generalized reliabilities (forward mapping). The selection among these possible results for the forward mapping is performed with the `compute` keyword followed by either `probabilities`, `reliabilities`, or `gen_reliabilities`. For example, specifying a `response_level` of 52.3 followed with `compute probabilities` will result in the calculation that the (uncertain) output value is less than or equal to 52.3, given the uncertain distributions on the inputs. Conversely, the `probability_levels`, `reliability_levels`, and `gen_reliability_levels` specifications provide target levels for which response values will be computed (inverse mapping). For example, specifying a `probability_level` of 0.95 will result in the calculation of a response value which corresponds to the 95th percentile of the output distribution. Specifications of `response_levels`, `probability_levels`, `reliability_levels`, and `gen_reliability_levels` may be combined within the calculations for each response function. The mapping results (probabilities, reliabilities, or generalized reliabilities for the forward mapping and response values for the inverse mapping) define the final statistics of the nondeterministic analysis that can be accessed for use at a higher level (via the primary and secondary mapping matrices for nested models; see Nested Model Controls).

Sets of response-probability pairs computed with the forward/inverse mappings define either a cumulative distribution function (CDF) or a complementary cumulative distribution function (CCDF) for each response function. In the case of evidence-based epistemic methods, this is generalized to define either cumulative belief and plausibility functions (CBF and CPF) or complementary cumulative belief and plausibility functions (CCBF and CCPF) for each response function, where a forward mapping involves computing the belief and plausibility probability level for a specified response level and an inverse mapping involves computing the belief and plausibility response level for either a specified probability level or a specified generalized reliability level (two results for each level mapping in the evidence-based epistemic case, instead of the one result for each level mapping in the aleatory case). The selection of a CDF/CBF/CPF or CCDF/CCBF/CCPF can be performed with the `distribution` keyword followed by either `cumulative` for the CDF/CBF/CPF option or `complementary` for the CCDF/CCBF/CCPF option. This selection also defines the sign of the reliability or generalized reliability indices. Table 5.28 provides the specification detail for the forward/inverse mappings used by each of the nondeterministic analysis methods.

Different nondeterministic methods have differing support for uncertain variable distributions. Tables 5.29, 5.30, and 5.31 summarize the uncertain variables that are available for use by the different methods, where a "-" indicates that the distribution is not supported by the method, a "U" means the uncertain input variables of this type must be uncorrelated, a "C" denotes that correlations are supported involving uncertain input variables of this type, and an "A" means the method must be in `all_variables` mode to support the given variable type. Additional notes include:

- we have four variants for stochastic expansions (SE), listed as Wiener, Askey, Extended, and Piecewise which draw from different sets of basis polynomials. The term stochastic expansion indicates polynomial chaos and stochastic collocation collectively, although the Piecewise option is only currently supported for stochastic collocation. Refer to Polynomial chaos expansion method and Stochastic collocation method for additional information on these three options.
- methods supporting the epistemic interval distributions have differing approaches: `sampling` and the `lhs` option of `global_interval_est` model the interval basic probability assignments (BPAs) as

Description	Keyword	Associated Data	Status	Default
Distribution type	distribution	cumulative | complementary	Optional group	cumulative (CDF)
Response levels	response_levels	list of reals	Optional group	No CDF/CCDF probabilities/reliabilities to compute
Number of response levels	num_response_levels	list of integers	Optional	response_levels evenly distributed among response functions
Target statistics for response levels	compute	probabilities | reliabilities | gen_reliabilities	Optional	probabilities
Probability levels	probability_levels	list of reals	Optional group	No CDF/CCDF response levels to compute
Number of probability levels	num_probability_levels	list of integers	Optional	probability_levels evenly distributed among response functions
Reliability levels	reliability_levels	list of reals	Optional group	No CDF/CCDF response levels to compute
Number of reliability levels	num_reliability_levels	list of integers	Optional	reliability_levels evenly distributed among response functions
Generalized reliability levels	gen_reliability_levels	list of reals	Optional group	No CDF/CCDF response levels to compute
Number of generalized reliability levels	num_gen_reliability_levels	list of integers	Optional	gen_reliability_levels evenly distributed among response functions

Table 23.1: Specification detail for forward/inverse level mappings

continuous histogram bin distributions for purposes of generating samples; `local_interval_est` and the `ego` option of `global_interval_est` ignore the BPA details and models these variables as simple bounded regions defined by the cell extremes; and `local_evidence` and `global_evidence` model the interval specifications as true BPAs.

23.1 Aleatory Uncertainty Quantification Methods

Aleatory uncertainty is also known as inherent variability, irreducible uncertainty, or randomness. An example of aleatory uncertainty is the distribution of height in a population, as it is characterized by the availability of sufficient data to accurately model the form of the variation. For this reason, aleatory uncertainty is typically modeled using probabilistic approaches through the specification of probability distributions to represent the uncertain input variables and the propagation of this uncertainty using probability theory. The probabilistic approaches supported in DAKOTA include sampling, local and global reliability, polynomial chaos, and stochastic collocation, which may be used to propagate random variables described by Normal Distribution, Lognormal Distribution, Uniform Distribution, Loguniform Distribution, Triangular Distribution, Exponential Distribution, Beta Distribution, Gamma Distribution, Gumbel Distribution, Frechet Distribution, Weibull Distribution, Histogram Bin Distribution, Poisson Distribution, Binomial Distribution, Negative Binomial Distribution, Geometric Distribution, Hypergeometric Distribution, and/or Histogram Point Distribution.

23.1.1 Nondeterministic sampling method

The nondeterministic sampling method is selected using the `sampling` specification. This method draws samples from the specified uncertain variable probability distributions and propagates them through the model to obtain statistics on the output response functions of interest. DAKOTA provides access to nondeterministic sampling methods through the combination of the **NonDSampling** base class and the **NonDLHSSampling** derived class.

CDF/CCDF probabilities are calculated for specified response levels using a simple binning approach. Response levels are calculated for specified CDF/CCDF probabilities and generalized reliabilities by indexing into a sorted samples array (the response levels computed are not interpolated and will correspond to one of the sampled values). CDF/CCDF reliabilities are calculated for specified response levels by computing the number of sample standard deviations separating the sample mean from the response level. Response levels are calculated for specified CDF/CCDF reliabilities by projecting out the prescribed number of sample standard deviations from the sample mean.

The `seed` integer specification specifies the seed for the random number generator which is used to make sampling studies repeatable, and `rng` specifies which random number generator is used. The `fixed_seed` flag is relevant if multiple sampling sets will be generated during the course of a strategy (e.g., surrogate-based optimization, optimization under uncertainty). Specifying this flag results in the reuse of the same seed value for each of these multiple sampling sets, which can be important for reducing variability in the sampling results. However, this behavior is not the default as the repetition of the same sampling pattern can result in a modeling weakness that an optimizer could potentially exploit (resulting in actual reliabilities that are lower than the estimated reliabilities). In either case (`fixed_seed` or not), the study is repeatable if the user specifies a `seed` and the study is random is the user omits a `seed` specification.

The number of samples to be evaluated is selected with the `samples` integer specification. The algorithm used to generate the samples can be specified using `sample_type` followed by either `random`, for pure random Monte Carlo sampling, or `lhs`, for Latin Hypercube sampling.

Distribution Type	Sampling	Local Reliability	Global Reliability	Wiener SE	Askey SE	Extended SE	Piecewise SE	Local Interval	Global Interval	Local Evidence	Global Evidence
Normal	C	C	C	C	C	C	-	-	-	-	-
Bounded Normal	C	U	U	U	U	U	U	-	-	-	-
Lognormal	C	C	C	C	C	U	-	-	-	-	-
Bounded Lognormal	C	U	U	U	U	U	U	-	-	-	-
Uniform	C	C	C	C	U	U	U	-	-	-	-
Loguniform	C	U	U	U	U	U	U	-	-	-	-
Triangular	C	U	U	U	U	U	U	-	-	-	-
Exponential	C	C	C	C	U	U	-	-	-	-	-
Beta	C	U	U	U	U	U	U	-	-	-	-
Gamma	C	C	C	C	U	U	-	-	-	-	-
Gumbel	C	C	C	C	C	U	-	-	-	-	-
Frechet	C	C	C	C	C	U	-	-	-	-	-
Weibull	C	C	C	C	C	U	-	-	-	-	-
Continuous Histogram Bin	C	U	U	U	U	U	U	-	-	-	-

Table 23.2: Summary of Distribution Types supported by Nondeterministic Methods, Part I (Continuous Aleatory Types)

Distribution Type	Sampling	Local Reliability	Global Reliability	Wiener SE	Askey SE	Extended SE	Piecewise SE	Local Interval	Global Interval	Local Evidence	Global Evidence
Poisson	C	-	-	-	-	-	-	-	-	-	-
Binomial	C	-	-	-	-	-	-	-	-	-	-
Negative Binomial	C	-	-	-	-	-	-	-	-	-	-
Geometric	C	-	-	-	-	-	-	-	-	-	-
Hypergeometric	C	-	-	-	-	-	-	-	-	-	-
Discrete Histogram Point	C	-	-	-	-	-	-	-	-	-	-

Table 23.3: Summary of Distribution Types supported by Nondeterministic Methods, Part II (Discrete Aleatory Types)

Distribution Type	Sampling	Local Reliability	Global Reliability	Wiener SE	Askey SE	Extended SE	Piecewise SE	Local Interval	Global Interval	Local Evidence	Global Evidence
Interval	U	-	U,A	U,A	U,A	U,A	U,A	U	U	U	U
Continuous Design	U,A	-	U,A	U,A	U,A	U,A	U,A	-	-	-	-
Discrete Design Range, Int Set, Real Set	U,A	-	-	-	-	-	-	-	-	-	-
Continuous State	U,A	-	U,A	U,A	U,A	U,A	U,A	-	-	-	-
Discrete State Range, Int Set, Real Set	U,A	-	-	-	-	-	-	-	-	-	-

Table 23.4: Summary of Distribution Types supported by Nondeterministic Methods, Part III (Epistemic, Design, and State Types)

If the user wants to increment a particular set of samples with more samples to get better estimates of mean, variance, and percentiles, one can select `incremental_random` or `incremental_lhs` as the `sample_type`. Note that a preliminary sample of size N must have already been performed, and a `dakota.rst` restart file must be available from this original sample. For example, say a user performs an initial study using `lhs` as the `sample_type`, and generates 50 samples. If the user creates a new input file where `samples` is now specified to be 100, the `sample_type` is defined to be `incremental_lhs` or `incremental_random`, and `previous_samples` is specified to be 50, the user will get 50 new LHS samples which maintain both the correlation and stratification of the original LHS sample. The N new samples will be combined with the N original samples to generate a combined sample of size 2N. The syntax for running the second sample set is: `dakota -i input2.in -r dakota.rst`, where `input2.in` is the file which specifies incremental sampling. Note that the number of samples in the second set MUST currently be 2 times the number of previous samples, although incremental sampling based on any power of two may be supported in future releases.

The nondeterministic sampling method also supports a design of experiments mode through the `all_variables` flag. Normally, `sampling` generates samples only for the uncertain variables, and treats any design or state variables as constants. The `all_variables` flag alters this behavior by instructing the sampling algorithm to treat any continuous design or continuous state variables as parameters with uniform probability distributions between their upper and lower bounds. Samples are then generated over all of the continuous variables (design, uncertain, and state) in the variables specification. This is similar to the behavior of the design of experiments methods described in Design of Computer Experiments Methods, since they will also generate samples over all continuous design, uncertain, and state variables in the variables specification. However, the design of experiments methods will treat all variables as being uniformly distributed between their upper and lower bounds, whereas the `sampling` method will sample the uncertain variables within their specified probability distributions.

Finally, the nondeterministic sampling method supports two types of sensitivity analysis. In this context of sampling, we take sensitivity analysis to be global, not local as when calculating derivatives of output variables with respect to input variables. Our definition is similar to that of [Saltelli et al., 2004]: "The study of how uncertainty in the output of a model can be apportioned to different sources of uncertainty in the model input." As a default, DAKOTA provides correlation analyses when running LHS. Correlation tables are printed with the simple, partial, and rank correlations between inputs and outputs. These can be useful to get a quick sense of how correlated the inputs are to each other, and how correlated various outputs are to inputs. In addition, we have the capability to calculate sensitivity indices through variance based decomposition using the keyword `variance_based_decomp`. Variance based decomposition is a way of using sets of samples to understand how the variance of the output behaves, with respect to each input variable. A larger value of the sensitivity index, S_i, means that the uncertainty in the input variable i has a larger effect on the variance of the output. More details on the calculations and interpretation of the sensitivity indices can be found in [Saltelli et al., 2004] and [Weirs et al., 2010]. Note that `variance_based_decomp` is extremely computationally intensive since replicated sets of sample values are evaluated. If the user specified a number of samples, N, and a number of nondeterministic variables, M, variance-based decomposition requires the evaluation of N∗(M+2) samples. To obtain sensitivity indices that are reasonably accurate, we recommend that N, the number of samples, be at least one hundred and preferably several hundred or thousands. Because of the computational cost, `variance_based_decomp` is turned off as a default. Table 5.32 provides details of the nondeterministic sampling specifications beyond those of Table 5.28.

23.1.2 Local reliability methods

Local reliability methods are selected using the `local_reliability` specification and are implemented within the **NonDLocalReliability** class. These methods compute approximate response function distribution statistics based on specified uncertain variable probability distributions. Each of the local reliability methods can compute forward and inverse mappings involving response, probability, reliability, and generalized reliability

Description	Keyword	Associated Data	Status	Default
Nondeterministic sampling method	`sampling`	none	Required group	N/A
Sampling type	`sample_type`	`random` | `lhs` | `incremental_random` | `incremental_lhs`	Optional group	`lhs`
Number of samples	`samples`	integer	Optional	minimum required
Previous samples for incremental approaches	`previous_samples`	integer	Optional	0 (no previous_samples)
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple runs
Random number generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
All variables flag	`all_variables`	none	Optional	sampling only over uncertain variables
Variance based decomposition (VBD)	`variance_based_decomp`	none	Optional	No VBD analysis
VBD tolerance for omitting small indices	`drop_tolerance`	none	Optional	All VBD indices displayed

Table 23.5: Specification detail for nondeterministic sampling method

levels.

The Mean Value method (MV, also known as MVFOSM in [Haldar and Mahadevan, 2000]) is the simplest, least-expensive method in that it estimates the response means, response standard deviations, and all CDF/CCDF forward/inverse mappings from a single evaluation of response functions and gradients at the uncertain variable means. This approximation can have acceptable accuracy when the response functions are nearly linear and their distributions are approximately Gaussian, but can have poor accuracy in other situations.

All other reliability methods perform an internal nonlinear optimization to compute a most probable point (MPP) of failure. A sign convention and the distance of the MPP from the origin in the transformed standard normal space ("u-space") define the reliability index, as explained in the section on Reliability Methods in the Uncertainty Quantification chapter of the Users Manual [Adams et al., 2010]. The reliability can then be converted to a probability using either first- or second-order integration, may then be refined using importance sampling, and finally may be converted to a generalized reliability index. The forward reliability analysis algorithm of computing reliabilities/probabilities for specified response levels is called the Reliability Index Approach (RIA), and the inverse reliability analysis algorithm of computing response levels for specified probability levels is called the Performance Measure Approach (PMA). The different RIA/PMA algorithm options are specified using the `mpp_search` specification which selects among different limit state approximations that can be used to reduce computational expense during the MPP searches. The `x_taylor_mean` MPP search option performs a single Taylor series approximation in the space of the original uncertain variables ("x-space") centered at the uncertain variable means, searches for the MPP for each response/probability level using this approximation, and performs a validation response evaluation at each predicted MPP. This option is commonly known as the Advanced Mean Value (AMV) method. The `u_taylor_mean` option is identical to the `x_taylor_mean` option, except that the approximation is performed in u-space. The `x_taylor_mpp` approach starts with an x-space Taylor series at the uncertain variable means, but iteratively updates the Taylor series approximation at each MPP prediction until the MPP converges. This option is commonly known as the AMV+ method. The `u_taylor_mpp` option is identical to the `x_taylor_mpp` option, except that all approximations are performed in u-space. The order of the Taylor-series approximation is determined by the corresponding `responses` specification and may be first or second-order. If second-order (methods named AMV^2 and AMV^2+ in [Eldred and Bichon, 2006]), the series may employ analytic, finite difference, or quasi Hessians (BFGS or SR1). The `x_two_point` MPP search option uses an x-space Taylor series approximation at the uncertain variable means for the initial MPP prediction, then utilizes the Two-point Adaptive Nonlinear Approximation (TANA) outlined in [Xu and Grandhi, 1998] for all subsequent MPP predictions. The `u_two_point` approach is identical to `x_two_point`, but all the approximations are performed in u-space. The `x_taylor_mpp` and `u_taylor_mpp`, `x_two_point` and `u_two_point` approaches utilize the `max_iterations` and `convergence_tolerance` method independent controls to control the convergence of the MPP iterations (the maximum number of MPP iterations per level is limited by `max_iterations`, and the MPP iterations are considered converged when $\| \mathbf{u}^{(k+1)} - \mathbf{u}^{(k)} \|_2 <$ `convergence_tolerance`). And, finally, the `no_approx` option performs the MPP search on the original response functions without the use of any approximations. The optimization algorithm used to perform these MPP searches can be selected to be either sequential quadratic programming (uses the `npsol_sqp` optimizer) or nonlinear interior point (uses the `optpp_q_newton` optimizer) algorithms using the `sqp` or `nip` keywords.

In addition to the MPP search specifications, one may select among different integration approaches for computing probabilities at the MPP by using the `integration` keyword followed by either `first_order` or `second_order`. Second-order integration employs the formulation of [Hohenbichler and Rackwitz, 1988] (the approach of [Breitung, 1984] and the correction of [Hong 1999] are also implemented, but are not active). Combining the `no_approx` option of the MPP search with first- and second-order integrations results in the traditional first- and second-order reliability methods (FORM and SORM). These integration approximations may be subsequently refined using importance sampling. The `refinement` specification allows the seletion of basic importance sampling (`import`), adaptive importance sampling (`adapt_import`), or multimodal adaptive importance sampling (`mm_adapt_import`), along with the specification of number of samples (`samples`) and

random seed (seed). Additional details on these methods are available in [Eldred et al., 2004b] and [Eldred and Bichon, 2006] and in the Uncertainty Quantification Capabilities chapter of the Users Manual [Adams et al., 2010].

Table 5.33 provides details of the local reliability method specifications beyond those of Table 5.28.

Description	Keyword	Associated Data	Status	Default
Reliability method	local_-reliability	none	Required group	N/A
MPP search type	mpp_search	x_taylor_-mean | u_taylor_-mean | x_taylor_mpp | u_taylor_mpp | x_two_point | u_two_point | no_approx	Optional group	No MPP search (MV method)
MPP search algorithm	sqp, nip	none	Optional	NPSOL's SQP algorithm
Integration method	integration	first_order | second_order	Optional group	First-order integration
Refinement method	refinement	import | adapt_import | mm_adapt_-import	Optional group	No refinement
Refinement samples	samples	integer	Optional	0
Refinement seed	seed	integer	Optional group	randomly generated seed

Table 23.6: Specification detail for local reliability methods

23.1.3 Global reliability methods

Global reliability methods are selected using the global_reliability specification and are implemented within the **NonDGlobalReliability** class. These methods do not support forward/inverse mappings involving reliability_levels, since they never form a reliability index based on distance in u-space. Rather they use a Gaussian process model to form an approximation to the limit state (based either in x-space via the x_-gaussian_process specification or in u-space via the u_gaussian_process specification), followed by probability estimation based on multimodal adaptive importance sampling (see [Bichon et al.,2007"]) and [\ref Bichon2008 "Bichon et al., 2008]). These probability estimates may then be transformed into generalized reliability levels if desired. At this time, inverse reliability analysis (mapping probability or generalized reliability levels into response levels) is not yet operational. The Gaussian process model approximation to the limit state is formed over the uncertain variables by default, but may be extended to also capture the effect of design and state variables via the all_variables flag. By default, the Surfpack GP (Kriging) model is used, but the DAKOTA implementation may be selected instead. If use_derivatives is specified the GP model will be built using available derivative data (Surfpack GP only).

Table 5.34 provides details of the global reliability method specifications beyond those of Table 5.28.

Description	Keyword	Associated Data	Status	Default
Global reliability method	`global_-reliability`	none	Required group	N/A
Approximation type	`x_gaussian_-process` | `u_gaussian_-process`	none	Required	N/A
GP selection	`gaussian_-process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_-derivatives`	none	Optional	Use function values only
All variables flag	`all_-variables`	none	Optional	Contour estimation only over uncertain variables
Random seed for initial GP construction	`seed`	integer	Optional	Time based seed: nonrepeatable

Table 23.7: Specification detail for global reliability methods

23.1.4 Polynomial chaos expansion method

The polynomial chaos expansion (PCE) is a general framework for the approximate representation of random response functions in terms of finite-dimensional series expansions in standardized random variables

$$R = \sum_{i=0}^{P} \alpha_i \Psi_i(\xi)$$

where α_i is a deterministic coefficient, Ψ_i is a multidimensional orthogonal polynomial and ξ is a vector of standardized random variables. An important distinguishing feature of the methodology is that the functional relationship between random inputs and outputs is captured, not merely the output statistics as in the case of many nondeterministic methodologies. DAKOTA provides access to PCE methods through the **NonDPolynomialChaos** class. Refer to the Uncertainty Quantification Capabilities chapter of the Users Manual [Adams et al., 2010] for additional information on the PCE algorithm.

To select the basis Ψ_i of the expansion, three approaches may be employed, as previously shown in Tables 5.29, 5.30, and 5.31: Wiener, Askey, and Extended. The Wiener option uses a Hermite orthogonal polynomial basis for all random variables and employs the same nonlinear variable transformation as the local and global reliability methods (and therefore has the same variable support). The Askey option, however, employs an extended basis of Hermite, Legendre, Laguerre, Jacobi, and generalized Laguerre orthogonal polynomials. The Extended option avoids the use of any nonlinear variable transformations by augmenting the Askey approach with numerically-generated orthogonal polynomials for non-Askey probability density functions. The selection of Wiener versus Askey versus Extended is partially automated and partially under the user's control. The Extended option is the default and supports only Gaussian correlations (see Tables 5.29, 5.30, and 5.31). This default can be overridden by the user by supplying the keyword `askey` to request restriction to the use of Askey bases only or by supplying the keyword `wiener` to request restriction to the use of exclusively Hermite bases. If needed to support prescribed correlations (not under user control), the Extended and Askey options will fall back to the Wiener option *on a per variable basis*. If the prescribed correlations are also unsupported by Wiener expansions, then DAKOTA will exit with an error. Additional details include:

- Askey polynomial selections include Hermite for normal (optimal) as well as bounded normal, lognormal, bounded lognormal, gumbel, frechet, and weibull (sub-optimal); Legendre for uniform (optimal) as well as loguniform, triangular, and bin-based histogram (sub-optimal); Laguerre for exponential (optimal); Jacobi for beta (optimal); and generalized Laguerre for gamma (optimal).
- Extended polynomial selections replace each of the sub-optimal Askey basis selections with numerically-generated polynomials that are orthogonal to the prescribed probability density functions (for bounded normal, lognormal, bounded lognormal, loguniform, triangular, gumbel, frechet, weibull, and bin-based histogram).

The `p_refinement` keyword specifies the usage of automated polynomial order refinement, which can be either `uniform` or `dimension_adaptive`. The `dimension_adaptive` option is supported for the tensor-product quadrature and Smolyak sparse grid options (see Table 5.36 below), and `uniform` is supported for tensor and sparse grids as well as regression approaches (`collocation_points` or `collocation_ratio`, see Table 5.37 below). Each of these refinement cases makes use of the `max_iterations` and `convergence_tolerance` method independent controls (see Table 5.1); the former control limits the number of refinement iterations, and the latter control terminates refinement when the two-norm of the change in the response covariance matrix (or, in goal-oriented approaches, the two-norm of change in the statistical quantities of interest (QOI)) falls below the tolerance. The `dimension_adaptive` case can be further specified to utilize `sobol`, `decay`, or `generalized` refinement controls. The former two cases employ anisotropic tensor/sparse grids in which the anisotropic dimension preference (leading to anisotropic integrations/expansions with differing refinement levels for different random dimensions) is determined using either total Sobol' indices from variance-based decomposition (`sobol` case: high indices result in high dimension preference) or using spectral coefficient decay rates from a rate estimation technique similar to Richardson extrapolation (`decay` case: low decay rates result in high dimension preference). In these two cases as well as the `uniform` refinement case, the `quadrature_order` or `sparse_grid_level` are ramped by one on each refinement iteration until either of the two convergence controls is satisfied. For the `uniform` refinement case with regression approaches, the `expansion_order` is ramped by one on each iteration while the oversampling ratio (either defined by `collocation_ratio` or inferred from `collocation_points` based on the initial expansion) is held fixed. Finally, the `generalized` `dimension_adaptive` case is the default adaptive approach; it refers to the generalized sparse grid algorithm, in which candidate index sets are evaluated for their impact on the statistical QOI, the most influential sets are selected and used to generate additional candidates, and the index set frontier of a sparse grid is evolved in an unstructured and goal-oriented manner (refer to User's Manual PCE descriptions for additional specifics).

The `variance_based_decomp` and `drop_tolerance` are also the same as those described in Nondeterministic sampling method, but since PCE outputs main, interaction, and total effects by default, the `univariate_effects` option has been added to allow suppression of the interaction effects since the output volume of these results can be prohibitive for high dimensional problems.

As for Nondeterministic sampling method and Global reliability methods, the `all_variables` flag can be used to form expansions over all continuous variables, rather than just the default aleatory uncertain continuous variables. For continuous design, continuous state, and continuous epistemic uncertain variables included in `all_variables` mode, Legendre chaos bases are used to model the bounded intervals for these variables. However, these variables are not assumed to have any particular probability distribution, only that they are independent variables. Moreover, when probability integrals are evaluated, only the aleatory random variable domain is integrated, leaving behind a polynomial relationship between the statistics and the remaining design/state/epistemic variables.

Table 5.35 shows these general PCE controls and it, along with Tables 5.36, 5.37, and 5.38 to follow, provide the details of the polynomial chaos expansion specifications beyond those of Table 5.28.

To obtain the coefficients α_i of the expansion, six options are provided and the specification details for these

Description	Keyword	Associated Data	Status	Default
Polynomial chaos expansion method	polynomial_chaos	none	Required group	N/A
Alternate basis of orthogonal polynomials	askey | wiener	none	Optional	Extended basis of orthogonal polynomials (Askey + numerically generated)
Automated polynomial order refinement	p_refinement	uniform | dimension_adaptive	Optional group	No p-refinement
Dimension-adaptive refinement control	sobol | decay | generalized	none	Optional group	generalized
Variance-based decomposition (VBD)	variance_based_decomp	none	Optional group	VBD indices not computed/printed
Restriction of VBD indices to main/total	univariate_effects	none	Optional	Unrestricted: interaction effects included
VBD tolerance for omitting small indices	drop_tolerance	real	Optional	All VBD indices displayed
All variables flag	all_variables	none	Optional	Expansion only over uncertain variables

Table 23.8: Specification detail for polynomial chaos expansion method: general controls.

options are provided in Tables 5.36 and 5.37:

1. multidimensional integration by a tensor-product of Gaussian quadrature rules (specified with `quadrature_order`, and, optionally, `dimension_preference`). The default rule selection is to employ `non_nested` Gauss rules including Gauss-Hermite (for normals or transformed normals), Gauss-Legendre (for uniforms or transformed uniforms), Gauss-Jacobi (for betas), Gauss-Laguerre (for exponentials), generalized Gauss-Laguerre (for gammas), and numerically-generated Gauss rules (for other distributions when using an Extended basis). For the case of `p_refinement` or the case of an explicit `nested` override, Gauss-Hermite rules are replaced with Genz-Keister nested rules and Gauss-Legendre rules are replaced with Gauss-Patterson nested rules, both of which exchange lower integrand precision for greater point reuse. By specifying a `dimension_preference`, where higher preference leads to higher order polynomial resolution, the tensor grid may be rendered anisotropic. The dimension specified to have highest preference will be set to the specified `quadrature_order` and all other dimensions will be reduced in proportion to their reduced preference; any non-integral portion is truncated. To synchronize with tensor-product integration, a tensor-product expansion is used, where the order p_i of the expansion in each dimension is selected to be half of the integrand precision available from the rule in use, rounded down. In the case of non-nested Gauss rules with integrand precision $2m_i - 1$, p_i is one less than the quadrature order m_i in each dimension (a one-dimensional expansion contains the same number of terms, $p+1$, as the number of Gauss points). The total number of terms, *N*, in a tensor-product expansion involving *n* uncertain input variables is
$$N \;=\; 1 + P \;=\; \prod_{i=1}^{n}(p_i + 1)$$
In some advanced use cases (e.g., multifidelity UQ), multiple grid resolutions can be employed; for this reason, the `quadrature_order` specification supports an array input.

2. multidimensional integration by the Smolyak sparse grid method (specified with `sparse_grid_level` and, optionally, `dimension_preference`). The underlying one-dimensional integration rules are the same as for the tensor-product quadrature case; however, the default rule selection is `nested` for sparse grids (Genz-Keister for normals/transformed normals and Gauss-Patterson for uniforms/transformed uniforms). This default can be overridden with an explicit `non_nested` specification (resulting in Gauss-Hermite for normals/transformed normals and Gauss-Legendre for uniforms/transformed uniforms). As for tensor quadrature, the `dimension_preference` specification enables the use of anisotropic sparse grids (refer to the PCE description in the User's Manual for the anisotropic index set constraint definition). Similar to anisotropic tensor grids, the dimension with greatest preference will have resolution at the full `sparse_grid_level` and all other dimension resolutions will be reduced in proportion to their reduced preference. For PCE with either isotropic or anisotropic sparse grids, a summation of tensor-product expansions is used, where each anisotropic tensor-product quadrature rule underlying the sparse grid construction results in its own anisotropic tensor-product expansion as described in case 1. These anisotropic tensor-product expansions are summed into a sparse PCE using the standard Smolyak summation (again, refer to the User's Manual for additional details). As for `quadrature_order`, the `sparse_grid_level` specification admits an array input for enabling specification of multiple grid resolutions used by certain advanced solution methodologies.

3. multidimensional integration by Stroud cubature rules [Stroud, 1971] and extensions [Xiu, 2008], as specified with `cubature_integrand`. A total-order expansion is used, where the isotropic order *p* of the expansion is half of the integrand order, rounded down. The total number of terms *N* for an isotropic total-order expansion of order *p* over *n* variables is given by
$$N \;=\; 1 + P \;=\; 1 + \sum_{s=1}^{p} \frac{1}{s!} \prod_{r=0}^{s-1}(n + r) \;=\; \frac{(n+p)!}{n!p!}$$

Since the maximum integrand order is currently five for normal and uniform and two for all other types, at most second- and first-order expansions, respectively, will be used. As a result, cubature is primarily useful for global sensitivity analysis, where the Sobol' indices will provide main effects and, at most, two-way interactions. In addition, the random variable set must be independent and identically distributed (*iid*), so the use of `askey` or `wiener` transformations may be required to create *iid* variable sets in the transformed space (as well as to allow usage of the higher order cubature rules for normal and uniform). Note that global sensitivity analysis often assumes uniform bounded regions, rather than precise probability distributions, so the *iid* restriction would not be problematic in that case.

Description	Keyword	Associated Data	Status	Default
Quadrature order for PCE coefficient estimation	`quadrature_order`	list of integers (one per grid resolution)	Required (1 of 7 selections)	N/A
Sparse grid level for PCE coefficient estimation	`sparse_grid_level`	list of integers (one per grid resolution)	Required (1 of 7 selections)	N/A
Cubature integrand order for PCE coefficient estimation	`cubature_integrand`	integer (1, 2, 3, or 5 for normal or uniform; 1 or 2 for exponential, beta, or gamma, 2 for all other distribution types)	Required (1 of 7 selections)	N/A
Dimension preference for anisotropic tensor and sparse grids	`dimension_preference`	list of reals	Optional	isotropic grids
Sparse grid growth rule restriction	`restricted` | `unrestricted`	none	Optional group	`restricted` (except for generalized sparse grids)
Integration rule point nesting for `quadrature_order` or `sparse_grid_level`	`nested` | `non_nested`	None	Optional	quadrature: `non_nested` unless automated refinement; sparse grids: `nested`

Table 23.9: Specification detail for polynomial chaos expansion method: quadrature, cubature, and sparse grid approaches.

4. multidimensional integration by Latin hypercube sampling (specified with `expansion_samples`). In this case, the expansion order p cannot be inferred from the numerical integration specification and it is necessary to provide either `expansion_order` or `expansion_terms` to specify either p or N, respectively, for a total-order expansion, where the latter specification allows the use of a partial-order expansion (truncation of a complete order expansion, while supported, is not generally recommended).

5. linear regression (specified with either `collocation_points` or `collocation_ratio`). A total-order expansion is used and must be specified using either `expansion_order` or `expansion_terms` as described in the previous option. Given p or N, the total number of collocation points (including any

sample reuse) must be at least N, and an oversampling is generally advisable. To more easily satisfy this requirement (i.e., to avoid requiring the user to calculate N from n and p), `collocation_ratio` allows for specification of a constant oversampling factor applied to N (e.g., `collocation_ratio = 2`. produces samples = $2N$). In addition, the default linear relationship with N can be overridden using a real-valued exponent specified using `ratio_order`. In this case, the number of samples becomes cN^o where c is the `collocation_ratio` and o is the `ratio_order`. The `use_derivatives` flag informs the regression approach to include derivative matching equations (limited to gradients at present) in the least squares solutions, enabling the use of fewer collocation points for a given expansion order and dimension (number of points required becomes $\frac{cN^o}{n+1}$). When admissible, a constrained least squares approach is employed in which response values are first reproduced exactly and error in reproducing response derivatives is minimized. Two collocation grid options are supported: the default is Latin hypercube sampling ("point collocation"), and an alternate approach of "probabilistic collocation" is also available through inclusion of the `tensor_grid` keyword. In this alternate case, the collocation grid is defined using a subset of tensor-product quadrature points: the order of an isotropic tensor-product quadrature grid is selected as the minimum required to meet or exceed the point requirement, and then this set of points is filtered down to the requirement based on the subset of points having highest product integration weight. In our experience, the alternate probabilistic collocation approach using a structured grid tends to suffer from ill-conditioning more quickly than the default point collocation approach using an unstructured grid.

6. coefficient import from a file (specified with `expansion_import_file`). A total-order expansion is assumed and must be specified using either `expansion_order` or `expansion_terms`.

If n is small (e.g., two or three), then tensor-product Gaussian quadrature is quite effective and can be the preferred choice. For moderate n (e.g., five), tensor-product quadrature quickly becomes too expensive and the sparse grid and point collocation approaches are preferred. For large n (e.g., more than ten), point collocation may begin to suffer from ill-conditioning and sparse grids are generally recommended. Random sampling for coefficient estimation is generally not recommended due to its slow convergence rate, although it does hold the advantage that the simulation budget is more flexible than that required by the other approaches. For incremental studies, approaches 4 and 5 support reuse of previous samples through the `incremental_lhs` (refer to Nondeterministic sampling method for description of incremental LHS) and `reuse_samples` (refer to Global approximations for description of the "all" option of sample reuse) specifications, respectively.

In the quadrature and sparse grid cases, growth rates for nested and non-nested rules can be synchronized for consistency. For a non-nested Gauss rule used within a sparse grid, linear one-dimensional growth rules of $m = 2l + 1$ are used to enforce odd quadrature orders, where l is the grid level and m is the number of points in the rule. The precision of this Gauss rule is then $i = 2m - 1 = 4l + 1$. For nested rules, order growth with level is typically exponential; however, the default behavior is to restrict the number of points to be the lowest order rule that is available that meets the one-dimensional precision requirement implied by either a level l for a sparse grid ($i = 4l+1$) or an order m for a tensor grid ($i = 2m-1$). This behavior is known as "restricted growth" or "delayed sequences." To override this default behavior in the case of sparse grids, the `unrestricted` keyword can be used; it cannot be overridden for tensor grids using nested rules since it also provides a mapping to the available nested rule quadrature orders. An exception to the default usage of restricted growth is the `dimension_adaptive` `p_refinement` `generalized` sparse grid case described previously, since the ability to evolve the index sets of a sparse grid in an unstructured manner eliminates the motivation for restricting the exponential growth of nested rules.

Additional specifications include the level mappings described in Uncertainty Quantification Methods and the `sample_type`, `samples`, `seed`, `fixed_seed`, and `rng` specifications described in Nondeterministic sampling method, where the `sample_type` options are restricted to `random` and `lhs`. Each of these sampling specifications refer to sampling on the PCE approximation for the purposes of generating approximate statistics, which should be distinguished from simulation sampling for generating the chaos coefficients as described in

Description	Keyword	Associated Data	Status	Default
Number simulation samples to estimate coeffs	expansion_samples	integer	Required (1 of 7 selections)	N/A
Use incremental LHS for expansion_samples	incremental_lhs	none	Optional	no sample reuse in coefficient estimation
Number collocation points to estimate coeffs	collocation_points	integer	Required (1 of 7 selections)	N/A
Collocation point oversampling ratio to estimate coeffs	collocation_ratio	real	Required (1 of 7 selections)	N/A
Order of collocation oversampling relationship	ratio_order	real	Optional	1.
Derivative usage with collocation_{points or ratio}	use_derivatives	none	Optional	estimate by function values alone
Reuse points flag with collocation_{points or ratio}	reuse_points	none	Optional	no sample reuse in coefficient estimation
Tensor grid flag with collocation__{points or ratio}	tensor_grid	none	Optional	regression with LHS sample set ("point collocation")
PCE coeffs import file	expansion_import_file	string	Required (1 of 7 selections)	N/A
Expansion order	expansion_order	integer	Required (1 of 2 selections) for any above required spec	N/A
Expansion terms	expansion_terms	integer	Required (1 of 2 selections) for any above required spec	N/A

Table 23.10: Specification detail for polynomial chaos expansion method: expansion sampling, regression, and expansion import options.

options 4 and 5 above. The `sample_refinement` specification is similar to that of Local reliability methods, with the difference that the number of refinement samples is not under the user's control (these evaluations are approximation-based, so management of this expense is less critical). This option allows for refinement of probability and generalized reliability results using importance sampling. Table 5.38 provide the details of these sampling specifications for polynomial chaos.

Description	Keyword	Associated Data	Status	Default
Sampling type (random sampling for and on expansion)	`sample_type`	`random` | `lhs`	Optional group	`lhs`
Number of samples on PCE for generating statistics	`samples`	integer	Optional (required if sampling-based statistics are requested)	0
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple PCE runs
Random number generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
Importance sampling refinement	`sample_-refinement`	`import` | `adapt_import` | `mm_adapt_-import`	Optional group	No sampling refinement

Table 23.11: Specification detail for polynomial chaos expansion method: sampling controls.

The advanced use case of multifidelity UQ automatically becomes active if the model selected for iteration by the method specification is a multifidelity surrogate model (refer to Hierarchical approximations). In this case, an expansion will first be formed for the model discrepancy (the difference between response results if `additive correction` or the ratio of results if `multiplicative correction`), using the first `quadrature_-order` or `sparse_grid_level` value along with any specified refinement strategy. Second, an expansion will be formed for the low fidelity surrogate model, using the second `quadrature_order` or `sparse_-grid_level` value (if present; the first is reused if not present) along with any specified refinement strategy. Then the two expansions are combined (added or multiplied) into an expansion that approximates the high fidelity model, from which the final set of statistics are generated. For polynomial chaos expansions, this high fidelity expansion can differ significantly in form from the low fidelity and discrepancy expansions, particularly in the `multiplicative` case where it is expanded to include all of the basis products.

23.1.5 Stochastic collocation method

The stochastic collocation (SC) method is very similar to the PCE method described above, with the key difference that the orthogonal polynomial basis functions are replaced with interpolation polynomial bases. The interpolation polynomials may be either local or global and either value-based or gradient-enhanced. In the local case, valued-based are piecewise linear splines and gradient-enhanced are piecewise cubic splines, and in the global case, valued-based are Lagrange interpolants and gradient-enhanced are Hermite interpolants. A value-based expansion

takes the form

$$R = \sum_{i=1}^{N_p} r_i L_i(\xi)$$

where N_p is the total number of collocation points, r_i is a response value at the ith collocation point, L_i is the ith multidimensional interpolation polynomial, and ξ is a vector of standardized random variables. The ith interpolation polynomial assumes the value of 1 at the ith collocation point and 0 at all other collocation points, involving either a global Lagrange polynomial basis or local piecewise splines. It is easy to see that the approximation reproduces the response values at the collocation points and interpolates between these values at other points. A gradient-enhanced expansion (selected via the `use_derivatives` keyword) involves both type 1 and type 2 basis functions as follows:

$$R = \sum_{i=1}^{N_p} [r_i H_i^{(1)}(\xi) + \sum_{j=1}^{n} \frac{dr_i}{d\xi_j} H_{ij}^{(2)}(\xi)]$$

where the ith type 1 interpolant produces 1 for the value at the ith collocation point, 0 for values at all other collocation points, and 0 for derivatives (when differentiated) at all collocation points, and the ij^{th} type 2 interpolant produces 0 for values at all collocation points, 1 for the jth derivative component at the ith collocation point, and 0 for the jth derivative component at all other collocation points. Again, this expansion reproduces the response values at each of the collocation points, and when differentiated, also reproduces each component of the gradient at each of the collocation points. Since this technique includes the derivative interpolation explicitly, it eliminates issues with matrix ill-conditioning that can occur in the gradient-enhanced PCE approach based on least squares regression. However, the calculation of high-order global polynomials with the desired interpolation properties can be similarly numerically challenging such that the use of local cubic splines is recommended due to numerical stability.

Thus, in PCE, one forms coefficients for known orthogonal polynomial basis functions, whereas SC forms multidimensional interpolation functions for known coefficients. DAKOTA provides access to SC methods through the **NonDStochCollocation** class. Refer to the Uncertainty Quantification Capabilities chapter of the Users Manual [Adams et al., 2010] for additional information on the SC algorithm.

As for Polynomial chaos expansion method, the orthogonal polynomials used in defining the Gauss points that make up the interpolation grid are governed by the Wiener, Askey, or Extended options. The Wiener option uses interpolation points from Gauss-Hermite (non-nested) or Genz-Keister (nested) integration rules for all random variables and employs the same nonlinear variable transformation as the local and global reliability methods (and therefore has the same variable support). The Askey option, however, employs interpolation points from Gauss-Hermite (Genz-Keister if nested), Gauss-Legendre (Gauss-Patterson if nested), Gauss-Laguerre, Gauss-Jacobi, and generalized Gauss-Laguerre quadrature. The Extended option avoids the use of any nonlinear variable transformations by augmenting the Askey approach with Gauss points from numerically-generated orthogonal polynomials for non-Askey probability density functions. As for PCE, the Wiener/Askey/Extended selection defaults to Extended, can be overridden by the user using the keywords `askey` or `wiener`, and automatically falls back from Extended/Askey to Wiener on a per variable basis as needed to support prescribed correlations. Unlike PCE, however, SC also supports the option of `piecewise` local basis functions. These are piecewise linear splines, or in the case of gradient-enhanced interpolation via the `use_derivatives` specification, piecewise cubic Hermite splines. Both of these basis selections provide local support only over the range from the interpolated point to its nearest 1D neighbors (within a tensor grid or within each of the tensor grids underlying a sparse grid), which exchanges the fast convergence of global bases for smooth functions for robustness in the representation of nonsmooth response functions (that can induce Gibbs oscillations when using high-order global basis functions).

Piecewise bases can be formulated using either a `nodal` basis approach or a `hierarchical` basis approach, where the latter enables local error estimation via hierarchical surpluses (currently under development). When local basis functions are used, the usage of nonequidistant collocation points (e.g., the Gauss point selections described above) are not well motivated, so equidistant Newton-Cotes points are employed in this case, and all random variable types are transformed to standard uniform probability space (refer to Tables 5.29, 5.30, and 5.31 for variable types for which this transformation is supported). The global gradient-enhanced interpolants (Hermite inteprolation polynomials) are also restricted to uniform or transformed uniform random variables (due to the need to compute collocation weights by integration of the basis polynomials) and share the variable support shown in these tables for Piecewise SE.

Automated expansion refinement can be selected as either `p_refinement` or `h_refinement`, and either refinement specification can be either `uniform` or `dimension_adaptive`. The `dimension_adaptive` case can be further specified as either `sobol` or `generalized` (`decay` not supported). Each of these automated refinement approaches makes use of the `max_iterations` and `convergence_tolerance` iteration controls (see Table 5.1). The `h_refinement` specification involves use of the same piecewise interpolants (linear or cubic Hermite splines) described above for the `piecewise` specification option (it is not necessary to redundantly specify `piecewise` in the case of `h_refinement`). In future releases, the `hierarchical` approach for `piecewise` bases will enable local `h_refinement` in addition to the current `uniform` and `dimension_adaptive` options.

The `variance_based_decomp` and `all_variables` specifications are identical to those described in Polynomial chaos expansion method. The `all_variables` flag expands the dimensionality of the interpolation to include continuous design, state, and epistemic uncertain variables, in addition to the default continuous aleatory uncertain variables. Interpolation points for these dimensions are based on Gauss-Legendre rules if non-nested, Gauss-Patterson rules if nested, and Newton-Cotes points in the case of piecewise bases. Again, when probability integrals are evaluated, only the aleatory random variable domain is integrated, leaving behind a polynomial relationship between the statistics and the remaining design/state/epistemic variables.

Table 5.39 shows these general SC controls, and it along with Table 5.40 to follow provide the details of the SC specifications beyond those of Tables 5.28 and 5.38.

To form the multidimensional interpolants L_i of the expansion, two options are provided and the specification details for these options are provided in Table 5.40.

1. interpolation on a tensor-product of Gaussian quadrature points (specified with `quadrature_order` and, optionally, `dimension_preference` for anisotropic tensor grids). As for PCE, non-nested Gauss rules are employed by default, although the presence of `p_refinement` or `h_refinement` will result in default usage of nested rules for normal or uniform variables after any variable transformations have been applied (both defaults can be overridden using explicit `nested` or `non_nested` specifications).
2. interpolation on a Smolyak sparse grid (specified with `sparse_grid_level` and, optionally, `dimension_preference` for anisotropic sparse grids) defined from Gaussian rules. As for sparse PCE, nested rules are employed unless overridden with the `non_nested` option, and the growth rules are restricted unless overridden by the `unrestricted` keyword.

If n is small, then tensor-product Gaussian quadrature is again the preferred choice. For larger n, tensor-product quadrature quickly becomes too expensive and the sparse grid approach is preferred. For self-consistency in growth rates, nested rules employ restricted exponential growth (with the exception of the `dimension_adaptive` `p_refinement` `generalized` case) for consistency with the linear growth used for non-nested Gauss rules (integrand precision $i = 4l + 1$ for sparse grid level l and $i = 2m - 1$ for tensor grid order m).

Additional specifications include the level mappings described in Uncertainty Quantification Methods, the `sample_type`, `samples`, `seed`, `fixed_seed`, and `rng` specifications described in Nondeterministic sam-

Description	Keyword	Associated Data	Status	Default
Stochastic collocation method	`stoch_-collocation`	none	Required group	N/A
Alternate basis selections for defining collocation points/weights	`askey` | `wiener` | `piecewise`	none	Optional	Gauss points/weights from extended set of global orthogonal polynomials (Askey + numerically generated)
Piecewise basis formulation	`nodal` | `hierarchical`	none	Optional	`nodal`
Automated expansion refinement	`p_refinement` | `h_refinement`	`uniform` | `dimension_-adaptive`	Optional group	No refinement
Dimension-adaptive refinement control	`sobol` | `generalized`	none	Optional group	`generalized`
Derivative usage flag	`use_-derivatives`	none	Optional	interpolation based on function values alone
Variance-based decomposition (VBD)	`variance_-based_decomp`	none	Optional group	VBD indices not computed/printed
Restriction of VBD indices to main/total	`univariate_-effects`	none	Optional	Unrestricted: interaction effects included
VBD tolerance for omitting small indices	`drop_-tolerance`	real	Optional	All VBD indices displayed
All variables flag	`all_-variables`	none	Optional	Expansion only over uncertain variables

Table 23.12: Specification detail for stochastic collocation method: general controls.

Description	Keyword	Associated Data	Status	Default
Quadrature order for collocation points	`quadrature_order`	list of integers (one per grid resolution)	Required (1 of 2 selections)	N/A
Sparse grid level for collocation points	`sparse_grid_level`	list of integers (one per grid resolution)	Required (1 of 2 selections)	N/A
Dimension preference for anisotropic tensor and sparse grids	`dimension_preference`	list of reals	Optional	isotropic grids
Sparse grid growth rule restriction	`restricted` | `unrestricted`	none	Optional group	`restricted` (except for generalized sparse grids)
Integration rule point nesting	`nested` | `non_nested`	None	Optional	quadrature: `non_nested` unless automated refinement; sparse grids: `nested`

Table 23.13: Specification detail for stochastic collocation method: grid controls.

pling method, and the `sample_refinement` specification described in Polynomial chaos expansion method. Each of the sampling specifications is identical to those for PCE shown in Table 5.38 and refer to sampling on the interpolant for the purposes of generating approximate statistics, which should not be confused with collocation evaluations used for forming the interpolant. Finally, the process for multifidelity UQ is identical to that described in Polynomial chaos expansion method, with the exception that the high fidelity expansion generated from combining the low fidelity and discrepancy expansions retains the polynomial form of the low fidelity expansion (only the coefficients are updated).

23.2 Epistemic Uncertainty Quantification Methods

Epistemic uncertainty is also referred to as subjective uncertainty, reducible uncertainty, model form uncertainty, or uncertainty due to lack of knowledge. Examples of epistemic uncertainty are little or no experimental data for an unknown physical parameter, or the existence of complex physics or behavior that is not included in the simulation model of a system. Epistemic uncertainty can be modeled probabilistically but is often modeled using non-probabilistic approaches such as interval propagation, evidence theory, possibility theory, information gap theory, etc. In DAKOTA, epistemic uncertainty analysis is performed using interval analysis or Dempster-Shafer theory of evidence. Epistemic (or mixed aleatory-epistemic) uncertainty may also be propagated through the use of the Nondeterministic sampling method, although in this case, the output statistics are limited to response intervals (any epistemic component suppresses all probabilistic results). Mixed uncertainty can also be addressed through use of nested UQ (refer to the Users Manual [Adams et al., 2010] for NestedModel discussion and examples); in this case, epistemic and aleatory analyses can be segregated and intervals on probabilistic results can be reported. A subtle distinction exists between `sampling` for epistemic intervals and the `lhs` option of `global_interval_est`: the former allows mixed aleatory-epistemic uncertainty within a single level, whereas the latter supports only epistemic variables and relies on nesting to address mixed uncertainty. In each

of these cases, the Interval Uncertain Variable specification is used to describe the epistemic uncertainty using either simple intervals or basic probability assignments. Note that for mixed UQ problems with both aleatory and epistemic variables, if the user defines the epistemic variables as intervals and aleatory variables as probability distribution types, the method `sampling` (in a simple, single-level study) will result in intervals only on the output. Although the aleatory variables will be sampled according to their distributions, the output will only be reported as an interval given the presence of interval variables. There is also the option to perform nested sampling, where one separates the epistemic and aleatory uncertain variables, samples over epistemic variables in the outer loop and then samples the aleatory variables in the inner llop, resulting in intervals on statistics. The calculation of intervals on statistics can also be performed by using nested approaches with interval estimation or evidence methods in the outer loop and aleatory UQ methods on the inner loop such as stochastic expansion or reliability methods. More detail about these "intervals on statistics" approaches can be found in [Eldred and Swiler, 2009] and [Eldred et al., 2011].

23.2.1 Local Interval Estimation

In interval analysis, one assumes that nothing is known about an epistemic uncertain variable except that its value lies somewhere within an interval. In this situation, it is NOT assumed that the value has a uniform probability of occuring within the interval. Instead, the interpretation is that any value within the interval is a possible value or a potential realization of that variable. In interval analysis, the uncertainty quantification problem is one of determining the resulting bounds on the output (defining the output interval) given interval bounds on the inputs. Again, any output response that falls within the output interval is a possible output with no frequency information assigned to it.

We have the capability to perform interval analysis using either local methods (`local_interval_est`) or global methods (`global_interval_est`). If the problem is amenable to local optimization methods (e.g. can provide derivatives or use finite difference method to calculate derivatives), then one can use local methods to calculate these bounds. `local_interval_est` allows the user to specify either `sqp` which is sequential quadratic programming, or `nip` which is a nonlinear interior point method. Table 5.41 provides the specification for the local interval method.

Description	Keyword	Associated Data	Status	Default
Nondeterministic interval estimation	`local_-interval_est`	none	Required group	N/A
Estimation method	`sqp` | `nip`	none	Required group	N/A

Table 23.14: Specification detail for local interval estimation used in epistemic uncertainty

23.2.2 Global Interval Estimation

As mentioned above, when performing interval analysis, one assumes that nothing is known about an epistemic uncertain variable except that its value lies somewhere within an interval. The goal of uncertainty quantification in this context is to determine the resulting bounds on the output (defining the output interval) given interval bounds on the inputs.

In the global approach, one uses either a global optimization method or a sampling method to assess the bounds. `global_interval_est` allows the user to specify either `lhs`, which performs Latin Hypercube Sampling and takes the minimum and maximum of the samples as the bounds (no optimization is performed) or `ego`. In the case of `ego`, the efficient global optimization (EGO) method is used to calculate bounds (see the EGO method on this page for more explanation). By default, the Surfpack GP (Kriging) model is used, but the DAKOTA

implementation may be selected instead. If `use_derivatives` is specified the GP model will be built using available derivative data (Surfpack GP only). When using `lhs` or `ego`, one can specify a seed for the number of LHS samples, the random number generator, and the number of samples. Table 5.42 provides the specification for the global interval methods.

Description	Keyword	Associated Data	Status	Default
Nondeterministic interval estimation	`global_interval_est`	none	Required group	N/A
Estimation method	`lhs` | `ego`	none	Required group	N/A
EGO GP selection	`gaussian_process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_derivatives`	none	Optional	Use function values only
Random seed generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples	`samples`	integer	Optional	10,000 for LHS, approximately numVars^2 for EGO

Table 23.15: Specification detail for global interval estimation used in epistemic uncertainty

23.2.3 Local Evidence Theory (Dempster-Shafer) Methods

The above section discussed a pure interval approach. This section discusses Dempster-Shafer evidence theory. In this approach, one does not assign a probability distribution to each uncertain input variable. Rather, one divides each uncertain input variable into one or more intervals. The input parameters are only known to occur within intervals: nothing more is assumed. Each interval is defined by its upper and lower bounds, and a Basic Probability Assignment (BPA) associated with that interval. The BPA represents a probability of that uncertain variable being located within that interval. The intervals and BPAs are used to construct uncertainty measures on the outputs called "belief" and "plausibility." Belief represents the smallest possible probability that is consistent with the evidence, while plausibility represents the largest possible probability that is consistent with the evidence. For more information about the Dempster-Shafer theory of evidence, see Oberkampf and Helton, 2003 and Helton and Oberkampf, 2004.

Similar to the interval approaches, one may use global or local methods to determine plausbility and belief measures for the outputs. Note that to calculate the plausibility and belief cumulative distribution functions, one has to look at all combinations of intervals for the uncertain variables. Within each interval cell combination, the minimum and maximum value of the objective function determine the belief and plausibility, respectively. In terms of implementation, global methods use LHS sampling or global optimization to calculate the minimum and maximum values of the objective function within each interval cell, while local methods use gradient-based optimization methods to calculate these minima and maxima.

Finally, note that the nondeterministic general settings apply to the interval and evidence methods, but one needs to be careful about the interpretation and translate probabilistic measures to epistemic ones. For example, if the user specifies distribution of type complementary, a complementary plausibility and belief function will be generated for the evidence methods (as opposed to a complementary distribution function in the `sampling` case). If the user specifies a set of responses levels, both the belief and plausibility will be calculated for each response

level. Likewise, if the user specifies a probability level, the probability level will be interpreted both as a belief and plausibility, and response levels corresponding to the belief and plausibility levels will be calculated. Finally, if generalized reliability levels are specified, either as inputs (`gen_reliability_levels`) or outputs (`response_levels` with `compute gen_reliabilities`), then these are directly converted to/from probability levels and the same probability-based mappings described above are performed.

Table 5.43 provides the specification for the `local_evidence` method. Note that two local optimization methods are available: `sqp` (sequential quadratic programming or `nip` (nonlinear interior point method).

Description	Keyword	Associated Data	Status	Default
Nondeterministic local evidence method	`local_-evidence`	none	Required group	N/A
Estimation method	`sqp` | `nip`	none	Required group	N/A

Table 23.16: Specification detail for local evidence theory method for epistemic uncertainty

23.2.4 Global Evidence Theory (Dempster-Shafer) Methods

Evidence theory has been explained above in the Local Evidence Theory section. The basic idea is that one specifies an "evidence structure" on uncertain inputs and propagates that to obtain belief and plausibility functions on the response functions. The inputs are defined by sets of intervals and Basic Probability Assignments (BPAs). Evidence propagation is computationally expensive, since the minimum and maximum function value must be calculated for each "interval cell combination." These bounds are aggregated into belief and plausibility.

Table 5.44 provides the specification for the `global_evidence` method. `global_evidence` allows the user to specify either `lhs` or `ego`. `lhs` performs Latin Hypercube Sampling and takes the minimum and maximum of the samples as the bounds per "interval cell combination." In the case of `ego`, the efficient global optimization (EGO) method is used to calculate bounds (see the EGO method on this page for more explanation). By default, the Surfpack GP (Kriging) model is used, but the DAKOTA implementation may be selected instead. If `use_-derivatives` is specified the GP model will be built using available derivative data (Surfpack GP only). When using `lhs` or `ego`, one can specify a seed for the number of LHS samples, the random number generator, and the number of samples.

Note that to calculate the plausibility and belief cumulative distribution functions, one has to look at all combinations of intervals for the uncertain variables. In terms of implementation, if one is using LHS sampling as outlined above, this method creates a large sample over the response surface, then examines each cell to determine the minimum and maximum sample values within each cell. To do this, one needs to set the number of samples relatively high: the default is 10,000 and we recommend at least that number. If the model you are running is a simulation that is computationally quite expensive, we recommend that you set up a surrogate model within the DAKOTA input file so that `global_evidence` performs its sampling and calculations on the surrogate and not on the original model. If one uses optimization methods instead to find the minimum and maximum sample values within each cell, this can also be computationally expensive.

Description	Keyword	Associated Data	Status	Default
Nondeterministic global evidence method	`global_-evidence`	none	Required group	N/A
Estimation method	`lhs` | `ego`	none	Required group	N/A
GP selection	`gaussian_-process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_-derivatives`	none	Optional	Use function values only
Random seed generator	`rng`	`mt19937` or `rnum2`	Optional	Mersenne twister (`mt19937`)
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples	`samples`	integer	Optional	10,000 for LHS, approximately numVars^2 for EGO

Table 23.17: Specification detail for global evidence theory method for epistemic uncertainty

Chapter 24

Nondeterministic Calibration Methods

This iterator branch is an important focus for upcoming releases. Capabilities for inferring statistical models from data are important for a variety of current and emerging mission areas.

24.1 Bayesian Calibration Methods

Currently, we are in the process of incorporating Bayesian calibration methods in DAKOTA. These methods take prior information on parameter values (in the form of prior distributions) and observational data (e.g. from experiments) and produce posterior distributions on the parameter values. When the computational simulation is then executed with samples from the posterior parameter distributions, the results that are produced are consistent with ("agree with") the experimental data. In the case of calibrating parameters from a computational simulation model, we require a "likelihood function" that specifies the likelihood of observing a particular observation given the model and its associated parameterization. We assume a Gaussian likelihood function currently. The algorithms that produce the posterior distributions on model parameters are Monte Carlo Markov Chain (MCMC) sampling algorithms. MCMC methods require many samples, often tens of thousands, so in the case of model calibration, often emulators of the computational simulation are used.

We have two Bayesian calibration methods under development in DAKOTA. GPMSA (Gaussian Process Models for Simulation Analysis) is a code that has been developed at Los Alamos National Laboratory. It uses a Gaussian process model as the emulator for the expensive computational simulation. GPMSA also has extensive features for calibration, such as the capability to include a "model discrepancy" term and the capability to model functional data such as time series data. The other method is using components from the QUESO library (Quantification of Uncertainty for Estimation, Simulation, and Optimization) developed at the University of Texas. We are using a DRAM (Delayed Rejected Adaptive Metropolis) algorithm for the MCMC sampling from the QUESO library.

These methods are under active development and are not included in the current DAKOTA release.

Chapter 25

Solution Verification Methods

Solution verification procedures estimate the order of convergence of the simulation response data during the course of a refinement study. This branch of methods is new and currently only contains one algorithm: Richardson extrapolation.

25.1 Richardson Extrapolation

This method utilizes state variables as active variables (continuous state only at this time; discrete state to follow later) for parameterizing the refinement of the discretizations, and employs the `max_iterations` and `convergence_tolerance` method independent controls to manage the iterative procedures. The refinement path is determined from the `initial_state` of the `continuous_state` variables specification (refer to State Variables) in combination with the `refinement_rate`, where each of the state variables is treated as an independent refinement factor and each of the initial state values is repeatedly divided by the refinement rate value to define new discretization states. Three algorithm options are currently provided:

1. The `estimate_order` option is the simplest option. For each of the refinement factors, it evaluates three points along the refinement path and uses these results to perform an estimate of the convergence order for each response function.

1. The `converge_order` option is initialized using the `estimate_order` aproach, and additional refinements are performed along the refinement path until the convergence order estimates converge (two-norm of the change in response orders is less than the convergence tolerance).

1. The `converge_qoi` option is similar to the `converge_order` option, except that the convergence criterion is that the two-norm of the response discretization errors (computed from extrapolation) must be less than the convergence tolerance.

In each of these cases, convergence order for a response quantity of interest (QOI) is estimated from

$$p = ln(\frac{QOI_3 - QOI_2}{QOI_2 - QOI_1})/ln(r)$$

where r is the uniform refinement rate specified by `refinement_rate`. Table 5.45 provides the specification details for the `richardson_extrap` method.

Description	Keyword	Associated Data	Status	Default
Richardson extrapolation	`richardson_extrap`	none	Required group	N/A
Order estimation approach	`estimate_order` | `converge_order` | `converge_qoi`	none	Required	N/A
Refinement rate	`refinement_rate`	real	Optional	2.

Table 25.1: Specification detail for Richardson extrapolation method for solution verification

Chapter 26

Design of Computer Experiments Methods

Design and Analysis of Computer Experiments (DACE) methods compute response data sets at a selection of points in the parameter space. Three libraries are provided for performing these studies: DDACE, FSUDace, and PSUADE. The design of experiments methods do not currently make use of any of the method independent controls.

26.1 DDACE

The Distributed Design and Analysis of Computer Experiments (DDACE) library provides the following DACE techniques: grid sampling (`grid`), pure random sampling (`random`), orthogonal array sampling (`oas`), latin hypercube sampling (`lhs`), orthogonal array latin hypercube sampling (`oa_lhs`), Box-Behnken (`box_behnken`), and central composite design (`central_composite`). It is worth noting that there is some overlap in sampling techniques with those available from the nondeterministic branch. The current distinction is that the nondeterministic branch methods are designed to sample within a variety of probability distributions for uncertain variables, whereas the design of experiments methods treat all variables as having uniform distributions. As such, the design of experiments methods are well-suited for performing parametric studies and for generating data sets used in building global approximations (see Global approximations), but are not currently suited for assessing the effect of uncertainties. If a design of experiments over both design/state variables (treated as uniform) and uncertain variables (with probability distributions) is desired, then `sampling` can support this with its `all_variables` option (see Nondeterministic sampling method). DAKOTA provides access to the DDACE library through the **DDACEDesignCompExp** class.

In terms of method dependent controls, the specification structure is straightforward. First, there is a set of design of experiments algorithm selections separated by logical OR's (`grid` or `random` or `oas` or `lhs` or `oa_lhs` or `box_behnken` or `central_composite`). Second, there are optional specifications for the random seed to use in generating the sample set (`seed`), for fixing the seed (`fixed_seed`) among multiple sample sets (see Nondeterministic sampling method for discussion), for the number of samples to perform (`samples`), and for the number of symbols to use (`symbols`). The `seed` control is used to make sample sets repeatable, and the `symbols` control is related to the number of replications in the sample set (a larger number of symbols equates to more stratification and fewer replications). The `main_effects` control prints Analysis-of-Variance main effects results (e.g. ANOVA tables with p-values per variable). The `main_effects` control is only operational

with the orthogonal arrays or Latin Hypercube designs, not for Box Behnken or Central Composite designs. The `quality_metrics` control is available for the DDACE library. This control turns on calculation of volumetric quality measures which measure the uniformity of the point samples. More details on the quality measures are given under the description of the FSU sampling methods. The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in contributing to the output variance. More details on variance based decomposition are given in Nondeterministic sampling method. Design of experiments specification detail is given in Table 5.46.

Description	Keyword	Associated Data	Status	Default
Design of experiments method	`dace`	none	Required group	N/A
dace algorithm selection	`grid` | `random` | `oas` | `lhs` | `oa_lhs` | `box_behnken` | `central_composite`	none	Required	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple DACE runs
Number of samples	`samples`	integer	Optional	minimum required
Number of symbols	`symbols`	integer	Optional	default for sampling algorithm
Main effects	`main_effects`	none	Optional	No main_effects
Quality metrics	`quality_metrics`	none	Optional	No quality_metrics
Variance based decomposition	`variance_based_decomp`	none	Optional	No variance_based_decomp

Table 26.1: Specification detail for design of experiments methods

26.2 FSUDace

The Florida State University Design and Analysis of Computer Experiments (FSUDace) library provides the following DACE techniques: quasi-Monte Carlo sampling (`fsu_quasi_mc`) based on the Halton sequence (`halton`) or the Hammersley sequence (`hammersley`), and Centroidal Voronoi Tessellation (`fsu_cvt`). All three methods generate sets of uniform random variables on the interval [0,1]. If the user specifies lower and upper bounds for a variable, the [0,1] samples are mapped to the [lower, upper] interval. The quasi-Monte Carlo and CVT methods are designed with the goal of low discrepancy. Discrepancy refers to the nonuniformity of the sample points within the hypercube. Discrepancy is defined as the difference between the actual number and the expected number of points one would expect in a particular set B (such as a hyper-rectangle within the unit hypercube), maximized over all such sets. Low discrepancy sequences tend to cover the unit hypercube reasonably

uniformly. Quasi-Monte Carlo methods produce low discrepancy sequences, especially if one is interested in the uniformity of projections of the point sets onto lower dimensional faces of the hypercube (usually 1-D: how well do the marginal distributions approximate a uniform?) CVT does very well volumetrically: it spaces the points fairly equally throughout the space, so that the points cover the region and are isotropically distributed with no directional bias in the point placement. There are various measures of volumetric uniformity which take into account the distances between pairs of points, regularity measures, etc. Note that CVT does not produce low-discrepancy sequences in lower dimensions, however: the lower-dimension (such as 1-D) projections of CVT can have high discrepancy.

The quasi-Monte Carlo sequences of Halton and Hammersley are deterministic sequences determined by a set of prime bases. Generally, we recommend that the user leave the default setting for the bases, which are the lowest primes. Thus, if one wants to generate a sample set for 3 random variables, the default bases used are 2, 3, and 5 in the Halton sequence. To give an example of how these sequences look, the Halton sequence in base 2 starts with points 0.5, 0.25, 0.75, 0.125, 0.625, etc. The first few points in a Halton base 3 sequence are 0.33333, 0.66667, 0.11111, 0.44444, 0.77777, etc. Notice that the Halton sequence tends to alternate back and forth, generating a point closer to zero then a point closer to one. An individual sequence is based on a radix inverse function defined on a prime base. The prime base determines how quickly the [0,1] interval is filled in. Generally, the lowest primes are recommended.

The Hammersley sequence is the same as the Halton sequence, except the values for the first random variable are equal to 1/N, where N is the number of samples. Thus, if one wants to generate a sample set of 100 samples for 3 random variables, the first random variable has values 1/100, 2/100, 3/100, etc. and the second and third variables are generated according to a Halton sequence with bases 2 and 3, respectively. For more information about these sequences, see [Halton, 1960, Halton and Smith, 1964, and Kocis and Whiten, 1997].

The specification for specifying quasi-Monte Carlo (`fsu_quasi_mc`) is given below in Table 5.47. The user must specify if the sequence is (`halton`) or (`hammersley`). The user must also specify the number of samples to generate for each variable (`samples`). Then, there are three optional lists the user may specify. The first list determines where in the sequence the user wants to start. For example, for the Halton sequence in base 2, if the user specifies sequence_start = 2, the sequence would not include 0.5 and 0.25, but instead would start at 0.75. The default `sequence_start` is a vector with 0 for each variable, specifying that each sequence start with the first term. The `sequence_leap` control is similar but controls the "leaping" of terms in the sequence. The default is 1 for each variable, meaning that each term in the sequence be returned. If the user specifies a sequence_leap of 2 for a variable, the points returned would be every other term from the QMC sequence. The advantage to using a leap value greater than one is mainly for high-dimensional sets of random deviates. In this case, setting a leap value to the next prime number larger than the largest prime base can help maintain uniformity when generating sample sets for high dimensions. For more information about the efficacy of leaped Halton sequences, see [Robinson and Atcitty, 1999]. The final specification for the QMC sequences is the prime base. It is recommended that the user not specify this and use the default values. For the Halton sequence, the default bases are primes in increasing order, starting with 2, 3, 5, etc. For the Hammersley sequence, the user specifies (s-1) primes if one is generating an s-dimensional set of random variables.

The `fixed_sequence` control is similar to `fixed_seed` for other sampling methods. If `fixed_sequence` is specified, the user will get the same sequence (meaning the same set of samples) for subsequent calls of the QMC sampling method (for example, this might be used in a surrogate based optimization method or a parameter study where one wants to fix the uncertain variables). The `latinize` command takes the QMC sequence and "latinizes" it, meaning that each original sample is moved so that it falls into one strata or bin in each dimension as in Latin Hypercube sampling. The default setting is NOT to latinize a QMC sample. However, one may be interested in doing this in situations where one wants better discrepancy of the 1-dimensional projections (the marginal distributions). The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in

contributing to the output variance. More details on variance based decomposition are given in Nondeterministic sampling method.

Finally, `quality_metrics` calculates four quality metrics relating to the volumetric spacing of the samples. The four quality metrics measure different aspects relating to the uniformity of point samples in hypercubes. Desirable properties of such point samples are: are the points equally spaced, do the points cover the region, and are they isotropically distributed, with no directional bias in the spacing. The four quality metrics we report are h, chi, tau, and d. h is the point distribution norm, which is a measure of uniformity of the point distribution. Chi is a regularity measure, and provides a measure of local uniformity of a set of points. Tau is the second moment trace measure, and d is the second moment determinant measure. All of these values are scaled so that smaller is better (the smaller the metric, the better the uniformity of the point distribution). Complete explanation of these measures can be found in [Gunzburger and Burkardt, 2004.].

Description	Keyword	Associated Data	Status	Default
FSU Quasi-Monte Carlo	`fsu_quasi_mc`	none	Required group	N/A
Sequence type	`halton` | `hammersley`	none	Required group	N/A
Number of samples	`samples`	integer	Optional	(0) for standalone sampling, (minimum required) for surrogates
Sequence starting indices	`sequence_-start`	integer list (one integer per variable)	Optional	Vector of zeroes
Sequence leaping indices	`sequence_-leap`	integer list (one integer per variable)	Optional	Vector of ones
Prime bases for sequences	`prime_base`	integer list (one integer per variable)	Optional	Vector of the first s primes for s-dimensions in Halton, First (s-1) primes for Hammersley
Fixed sequence flag	`fixed_-sequence`	none	Optional	sequence not fixed: sampling patterns are variable among multiple QMC runs
Latinization of samples	`latinize`	none	Optional	No latinization
Variance based decomposition	`variance_-based_decomp`	none	Optional	No variance_-based_decomp
Quality metrics	`quality_-metrics`	none	Optional	No quality_metrics

Table 26.2: Specification detail for FSU Quasi-Monte Carlo sequences

The FSU CVT method (`fsu_cvt`) produces a set of sample points that are (approximately) a Centroidal Voronoi Tessellation. The primary feature of such a set of points is that they have good volumetric spacing; the points

tend to arrange themselves in a pattern of cells that are roughly the same shape. To produce this set of points, an almost arbitrary set of initial points is chosen, and then an internal set of iterations is carried out. These iterations repeatedly replace the current set of sample points by an estimate of the centroids of the corresponding Voronoi subregions. [Du, Faber, and Gunzburger, 1999].

The user may generally ignore the details of this internal iteration. If control is desired, however, there are a few variables with which the user can influence the iteration. The user may specify `max_iterations`, the number of iterations carried out; `num_trials`, the number of secondary sample points generated to adjust the location of the primary sample points; and `trial_type`, which controls how these secondary sample points are generated. In general, the variable with the most influence on the quality of the final sample set is `num_trials`, which determines how well the Voronoi subregions are sampled. Generally, `num_trials` should be "large", certainly much bigger than the number of sample points being requested; a reasonable value might be 10,000, but values of 100,000 or 1 million are not unusual.

CVT has a seed specification similar to that in DDACE: there are optional specifications for the random seed to use in generating the sample set (`seed`), for fixing the seed (`fixed_seed`) among multiple sample sets (see Nondeterministic sampling method for discussion), and for the number of samples to perform (`samples`). The `seed` control is used to make sample sets repeatable. Finally, the user has the option to specify the method by which the trials are created to adjust the centroids. The `trial_type` can be one of three types: `random`, where points are generated randomly; `halton`, where points are generated according to the Halton sequence; and `grid`, where points are placed on a regular grid over the hyperspace.

Finally, latinization is available for CVT as with QMC. The `latinize` control takes the CVT sequence and "latinizes" it, meaning that each original sample is moved so that it falls into one strata or bin in each dimension as in Latin Hypercube sampling. The default setting is NOT to latinize a CVT sample. However, one may be interested in doing this in situations where one wants better discrepancy of the 1-dimensional projections (the marginal distributions). The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in contributing to the output variance. More details on variance based decomposition are given in Nondeterministic sampling method. The `quality_metrics` control is available for CVT as with QMC. This command turns on calculation of volumetric quality measures which measure the "goodness" of the uniformity of the point samples. More details on the quality measures are given under the description of the QMC methods.

The specification detail for the FSU CVT method is given in Table 5.48.

26.3 PSUADE

The Problem Solving Environment for Uncertainty Analysis and Design Exploration (PSUADE) is a Lawrence Livermore National Laboratory tool for metamodeling, sensitivity analysis, uncertainty quantification, and optimization. Its features include non-intrusive and parallel function evaluations, sampling and analysis methods, an integrated design and analysis framework, global optimization, numerical integration, response surfaces (MARS and higher order regressions), graphical output with Pgplot or Matlab, and fault tolerance [C.H. Tong, 2005].

The Morris One-At-A-Time (MOAT) method, originally proposed by Morris [M.D. Morris, 1991], is a screening method, designed to explore a computational model to distinguish between input variables that have negligible, linear and additive, or nonlinear or interaction effects on the output. The computer experiments performed consist of individually randomized designs which vary one input factor at a time to create a sample of its elementary effects. The PSUADE implementation of MOAT is selected with method keyword `psuade_moat`. The number of samples (`samples`) must be a positive integer multiple of (number of continuous design variable + 1) and will be automatically adjusted if misspecified. The number of partitions (`partitions`) applies to each variable

Description	Keyword	Associated Data	Status	Default
FSU CVT sampling	`fsu_cvt`	none	Required group	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple CVT runs
Number of samples	`samples`	integer	Required	(0) for standalone sampling, (minimum required) for surrogates
Number of trials	`num_trials`	integer	Optional	10000
Trial type	`trial_type`	`random` | `grid` | `halton`	Optional	`random`
Latinization of samples	`latinize`	none	Optional	No latinization
Variance based decomposition	`variance_-based_decomp`	none	Optional	No variance_-based_decomp
Quality metrics	`quality_-metrics`	none	Optional	No quality_metrics

Table 26.3: Specification detail for FSU Centroidal Voronoi Tesselation sampling

being studied and must be odd (the number of MOAT levels per variable is partitions + 1). This will also be adjusted at runtime as necessary. For information on practical use of this method, see [Saltelli, et al., 2004]. The specification detail for the PSUADE MOAT method is given in Table 5.49.

Description	Keyword	Associated Data	Status	Default
PSUADE MOAT method	`psuade_moat`	none	Required group	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples	`samples`	integer	Optional	10∗(num_cdv + 1)
Number of partitions	`partitions`	integer	Optional	3

Table 26.4: Specification detail for PSUADE methods

Chapter 27

Parameter Study Methods

DAKOTA's parameter study methods compute response data sets at a selection of points in the parameter space. These points may be specified as a vector, a list, a set of centered vectors, or a multi-dimensional grid. Capability overviews and examples of the different types of parameter studies are provided in the Users Manual [Adams et al., 2010]. DAKOTA implements all of the parameter study methods within the **ParamStudy** class.

With the exception of output verbosity (a setting of `silent` will suppress some parameter study diagnostic output), DAKOTA's parameter study methods do not make use of the method independent controls. Therefore, the parameter study documentation which follows is limited to the method dependent controls for the vector, list, centered, and multidimensional parameter study methods.

27.1 Vector parameter study

DAKOTA's vector parameter study computes response data sets at selected intervals along a vector in parameter space. It is often used for single-coordinate parameter studies (to study the effect of a single variable on a response set), but it can be used more generally for multiple coordinate vector studies (to investigate the response variations along some n-dimensional vector such as an optimizer search direction). This study is selected using the `vector_parameter_study` specification followed by either a `final_point` or a `step_vector` specification.

The vector for the study can be defined in several ways (refer to dakota.input.summary). First, a `final_point` specification, when combined with the initial values from the variables specification (in Variables Commands, see `initial_point` and `initial_state` for design and state variables as well as inferred initial values for uncertain variables), uniquely defines an n-dimensional vector's direction and magnitude through its start and end points. The values included in the `final_point` specification are the actual variable values for discrete sets, not the underlying set index value. The intervals along this vector are then specified with a `num_steps` specification, for which the distance between the initial values and the `final_point` is broken into `num_steps` intervals of equal length. For continuous and discrete range variables, distance is measured in the actual values of the variables, but for discrete set variables (either integer or real sets for design, uncertain, or state types), distance is instead measured in index offsets. Since discrete sets may have nonuniform offsets in their enumerated set values but have uniform offsets in their index values, defining steps in terms of set indices allows for meaningful parameter study specifications for these variable types. This study performs function evaluations at both ends, making the total number of evaluations equal to `num_steps+1`. The study has stringent requirements on performing appropriate steps with any discrete range and discrete set variables. A `num_steps` specification

must result in discrete range and set index steps that are integral: no remainder is currently permitted in the integer step calculation and no rounding to integer steps will occur. The `final_point` specification detail is given in Table 5.50.

Description	Keyword	Associated Data	Status	Default
Vector parameter study	`vector_-parameter_-study`	none	Required group	N/A
Termination point of vector	`final_point`	list of reals (actual values; no set indices)	Required group	N/A
Number of steps along vector	`num_steps`	integer	Required	N/A

Table 27.1: final_point specification detail for the vector parameter study

The other technique for defining a vector in the study is the `step_vector` specification. This parameter study begins at the initial values and adds the increments specified in `step_vector` to obtain new simulation points. For discrete set types (design, uncertain, or state; real or integer), the steps are set index offsets, not steps between the actual set values. This increment process is performed `num_steps` times, and since the initial values are included, the total number of simulations is again equal to `num_steps+1`. The `step_vector` specification detail is given in Table 5.51.

Description	Keyword	Associated Data	Status	Default
Vector parameter study	`vector_-parameter_-study`	none	Required group	N/A
Step vector	`step_vector`	list of reals (index offset components are cast to integers)	Required group	N/A
Number of steps along vector	`num_steps`	integer	Required	N/A

Table 27.2: step_vector specification detail for the vector parameter study

27.2 List parameter study

DAKOTA's list parameter study allows for evaluations at user selected points of interest which need not follow any particular structure. This study is selected using the `list_parameter_study` method specification followed by a `list_of_points` specification.

The number of real values in the `list_of_points` specification must be a multiple of the total number of variables (including continuous and discrete types) contained in the variables specification. This parameter study simply performs simulations for the first parameter set (the first `n` entries in the list), followed by the next parameter set (the next `n` entries), and so on, until the list of points has been exhausted. Since the initial values from the variables specification will not be used, they need not be specified. For discrete set types, the actual values should be specified, not the set indices, although the values will be validated for membership within the set value specifications. The list parameter study specification detail is given in Table 5.52.

Description	Keyword	Associated Data	Status	Default
List parameter study	list_-parameter_-study	none	Required group	N/A
List of points to evaluate	list_of_-points	list of reals (actual values; no set indices)	Required	N/A

Table 27.3: Specification detail for the list parameter study

27.3 Centered parameter study

DAKOTA's centered parameter study computes response data sets along multiple coordinate-based vectors, one per parameter, centered about the initial values from the variables specification. This is useful for investigation of function contours with respect to each parameter individually in the vicinity of a specific point (e.g., post-optimality analysis for verification of a minimum), thereby avoiding the cost associated with a multidimensional grid. It is selected using the `centered_parameter_study` method specification followed by `step_vector` and `steps_per_variable` specifications. The `step_vector` specification provides the size of the increments for each variable (employed sequentially, not all at once as for Vector parameter study) in either actual values (continuous and discrete range) or index offsets (discrete set). The `steps_per_variable` specification provides the number of increments per variable (again, employed sequentially) in each of the plus and minus directions. The centered parameter study specification detail is given in Table 5.53.

Description	Keyword	Associated Data	Status	Default
Centered parameter study	centered_-parameter_-study	none	Required group	N/A
Step vector	step_vector	list of reals (index offset components are cast to integers)	Required group	N/A
Number of steps per variable	steps_per_-variable	list of integers	Required	N/A

Table 27.4: Specification detail for the centered parameter study

27.4 Multidimensional parameter study

DAKOTA's multidimensional parameter study computes response data sets for an n-dimensional grid of points. Each continuous and discrete range variable is partitioned into equally spaced intervals between its upper and lower bounds, each discrete set variable is partitioned into equally spaced index intervals, and each combination of the values defined by the boundaries of these partitions is evaluated.

This study is selected using the `multidim_parameter_study` method specification followed by a `partitions` specification, where the `partitions` list specifies the number of partitions for each variable. The number of entries in the partitions list must be equal to the total number of variables contained in the variables specification. As for the vector and centered studies, remainders within the integer division of the step calculations are not permitted for discrete range or set types and therefore no integer rounding occurs, so the partitions specification must be carefully selected in the presence of these types. Since the initial values from the variables

specification will not be used, they need not be specified. The multidimensional parameter study specification detail is given in Table 5.54.

Description	Keyword	Associated Data	Status	Default
Multidimensional parameter study	`multidim_-parameter_-study`	none	Required group	N/A
Partitions per variable	`partitions`	list of integers	Required	N/A

Table 27.5: Specification detail for the multidimensional parameter study

Chapter 28

Model Commands

Chapter 29

Model Description

A DAKOTA model specifies how variables are mapped into a set of responses in support of an iterative method and the model specification in an input file dictates the components to be used in constructing a particular model instance. This specification selects a **Model** from the model hierarchy, which includes **SingleModel**, **DataFitSurrModel**, **HierarchSurrModel**, and **NestedModel** derived classes. Depending on the type of derived model, different sub-specifications are needed to construct different components of the model.

Several examples follow. The first example shows a minimal specification for a single model, which is the default model when no models are specified by the user.

```
model,
single
```

This example does not provide any pointer strings and therefore relies on the default behavior of constructing the model with the last variables, interface, and responses specifications parsed.

The next example displays a surrogate model specification which selects a quadratic polynomial from among the global approximation methods. It uses a pointer to a design of experiments method for generating the data needed for building the global approximation, reuses any old data available for the current approximation region, and employs the first-order multiplicative approach to correcting the approximation at the center of the current approximation region.

```
model,
id_model = 'M1'
variables_pointer = 'V1'
responses_pointer = 'R1'
surrogate global
  polynomial quadratic
  dace_method_pointer = 'DACE'
  reuse_samples region
  correction multiplicative first_order
```

This example demonstrates the use of identifiers and pointers. It provides the optional model independent specifications for model identifier, variables pointer, and responses pointer (see Model Independent Controls) as well as model dependent specifications for global surrogates (see Global approximations).

Finally, an advanced nested model example would be

```
model,
```

```
id_model = 'M1'
variables_pointer = 'V1'
responses_pointer = 'R1'
nested
  optional_interface_pointer = 'OI1'
    optional_interface_responses_pointer = 'OIR1'
  sub_method_pointer = 'SM1'
    primary_variable_mapping   = ''  ''  'X'     'Y'
    secondary_variable_mapping = ''  ''  'mean'  'mean'
    primary_response_mapping   = 1. 0. 0. 0. 0. 0. 0. 0. 0.
    secondary_response_mapping = 0. 0. 0. 1. 3. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 3. 0.
```

This example also supplies model independent controls for model identifier, variables pointer, and responses pointer (see Model Independent Controls), and supplies model dependent controls for specifying details of the nested mapping (see Nested Model Controls).

Chapter 30

Model Specification

As alluded to in the examples above, the model specification has the following structure:

```
model,
<model independent controls>
<model selection>
  <model dependent controls>
```

The `<model independent controls>` are those controls which are valid for all models. Referring to dakota.input.summary, these controls are defined externally from and prior to the model selection blocks. The model selection blocks are all required group specifications separated by logical OR's, where the model selection must be `single` OR `surrogate` OR `nested`. If a `surrogate` model is specified, a secondary selection must be made for its type: `global`, `multipoint`, `local`, or `hierarchical`. The `<model dependent controls>` are those controls which are only meaningful for a specific model. These controls are defined within each model selection block. Defaults for model independent and model dependent controls are defined in **DataModel**. The following sections provide additional detail on the model independent controls followed by the model selections and their corresponding model dependent controls.

Chapter 31

Model Independent Controls

The model independent controls include a model identifier string, pointers to variables and responses specifications, and a model type specification. The model identifier string is supplied with `id_model` and is used to provide a unique identifier string for use within method specifications (refer to `model_pointer` in Method Independent Controls).

The type of model can be `single`, `nested`, or `surrogate`. Each of these model specifications supports `variables_pointer` and `responses_pointer` strings for identifying the variables and responses specifications used in constructing the model (by cross-referencing with `id_variables` and `id_responses` strings from particular variables and responses keyword specifications). These pointers are valid for each model type since each model contains a set of variables that is mapped into a set of responses -- only the specifics of the mapping differ. Additional pointers are used for each model type for constructing the components of the variable to response mapping. As a strategy specification identifies one or more methods and a method specification identifies a model, a model specification identifies variables, interface, and responses specifications. This top-down flow specifies all of the object interrelationships.

For each of these pointer specifications, if a pointer string is specified and no corresponding id string is available, DAKOTA will exit with an error message. If the pointer is optional and no pointer string is specified, then the last specification parsed will be used. It is appropriate to omit optional cross-referencing whenever the relationships are unambiguous due to the presence of only one specification.

Table 6.1 provides the specification detail for the model independent controls involving identifiers, model type controls, and pointers.

Description	Keyword	Associated Data	Status	Default
Model set identifier	`id_model`	string	Optional	method use of last model parsed
Model type	`single` | `surrogate` | `nested`	none	Required group	N/A (`single` if no model specification)
Variables set pointer	`variables_pointer`	string	Optional	model use of last variables parsed
Responses set pointer	`responses_pointer`	string	Optional	model use of last responses parsed

Table 31.1: Specification detail for the model independent controls: identifiers, model types, and pointers

Chapter 32

Single Model Controls

In the `single` model case, a single interface is used to map the variables into responses. The optional `interface_pointer` specification identifies this interface by cross-referencing with the `id_interface` string input from a particular interface keyword specification.

Table 6.2 provides the specification detail for single models.

Description	Keyword	Associated Data	Status	Default
Interface set pointer	`interface_-pointer`	string	Optional	model use of last interface parsed

Table 32.1: Specification detail for single models

Chapter 33

Surrogate Model Controls

In the `surrogate` model case, the specification first allows a mixture of surrogate and actual response mappings through the use of the optional `id_surrogates` specification. This identifies the subset of the response functions by number that are to be approximated (the default is all functions). The valid response function identifiers range from 1 through the total number of response functions (see Function Specification). Next, the specification selects a global, multipoint, local, or hierarchical approximation. Table 6.3 provides the specification detail for surrogate models.

Description	Keyword	Associated Data	Status	Default
Surrogate response ids	`id_-surrogates`	list of integers	Optional	All response functions are approximated
Surrogate type selection	`global` | `multipoint` | `local` | `hierarchical`	none	Required group	N/A

Table 33.1: Specification detail for the surrogate models

Each of these surrogate types provides an approximate representation of a "truth" model which is used to perform the parameter to response mappings. This approximation is built and updated using data from the truth model. This data is generated in some cases using a design of experiments iterator applied to the truth model (global approximations with a `dace_method_pointer`). In other cases, truth model data from a single point (local, hierarchical approximations), from a few previously evaluated points (multipoint approximations), or from the restart database (global approximations with `reuse_samples`) can be used. Surrogate models are used extensively in the surrogate-based optimization and least squares methods (see **SurrBasedMinimizer** and its derived classes and Surrogate-Based Minimization Methods), in which the goals are to reduce expense by minimizing the number of truth function evaluations and to smooth out noisy data with a global data fit. However, the use of surrogate models is not restricted to optimization techniques; uncertainty quantification and optimization under uncertainty methods are other primary users.

The following sections present the `global`, `multipoint`, `local`, and `hierarchical` specification groups in further detail.

33.1 Global approximations

The global surrogate model requires specification of one of the following approximation types: `polynomial`, `gaussian_process` (Kriging), `neural_network`, `mars`, `moving_least_squares`, or `radial_basis`. These specifications will create a polynomial, a Gaussian process (Kriging interpolation), layered perceptron artificial neural network approximation, multivariate adaptive regression spline, moving least squares, or radial basis function approximation, respectively. All these approximations are implemented in SurfPack except an alternate Gaussian process model, which is implemented directly in DAKOTA.

In the polynomial case, the order of the polynomial (linear, quadratic, or cubic) must be specified.

DAKOTA has two Gaussian process (GP) or Kriging models, one that resides in DAKOTA proper and the other in DAKOTA's Surfpack subpackage. Choice of the GP implmentation is made by specifying `gaussian_process` (alternately `kriging`) followed by either `dakota` or `surfpack`. Historically these models were drastically different, but in DAKOTA 5.1, they became quite similar. They now differ in that the Surfpack GP has a richer set of features/options and tends to be more accurate than the DAKOTA version. Due to how the Surfpack GP handles ill-conditioned correlation matrices (which significantly contributes to its greater accuracy), the `Surfpack` GP can be a factor of two or three slower than DAKOTA's. As of DAKOTA 5.2, the Surfpack implementation is the default in all contexts except Bayesian calibration. The DAKOTA GP is provided for continuity, but may be deprecated in the future.

The trend function of both GP models is selected using the `trend` keyword, with options `constant`, `linear`, or `reduced_quadratic` (default). The Surfpack GP has the additional option of (a full) `quadratic`. The `reduced_quadratic` trend function includes the main effects, but not mixed/interaction terms. NOTE: prior to DAKOTA 5.1, `reduced_quadratic` was selected for the DAKOTA GP via the `quadratic` option.

The DAKOTA GP always determines its correlation parameters via maximum likelihood estimate (MLE) using a global optimization procedure. Ill-conditioning induced by poorly spaced sample points is a common challenge in the construction of Gaussian process models. DAKOTA's GP deals with ill-conditioning in two ways. First, when it encounters a non-invertible correlation matrix it iteratively increases the size of a "nugget," but in such cases the resulting approximation smooths rather than interpolates the data. Second, it has a `point_selection` option (default off) that uses a greedy algorithm to select a well-spaced subset of points prior to the construction of the GP. In this case, the GP will only interpolate the selected subset. Typically, one should not need point selection in trust-region methods because a small number of points are used to develop a surrogate within each trust region. Point selection is most beneficial when constructing with a large number of points, typically more than order one hundred, though this depends on the number of variables and spacing of the sample points.

By default, the Surfpack GP also selects its correlation parameters using a global optimizer, but its search is concentrated in a smaller feasible region of correlation parameters. However, one may force the Surfpack GP to use a different search algorithm by specifying `optimization_method` followed by any of `'global'`, `'local'`, `'sampling'`, or `'none'`. The none option and starting location of the local search default to the center of the small feasible region of correlation lengths. However, one may also directly specify `correlation_lengths` as a list of N real numbers where N is the number of input dimensions. As of DAKOTA 5.2, the Surfpack GP handles ill-conditioning by selecting the subset of points that tightly meets a constraint on the condition number and provides the maximum amount of useful information for the correlation lengths chosen by MLE. For each set of correlation lengths examined during the MLE optimization, the optimal subset of points for those correlation lengths is used. The subset of points used to build the Surfpack GP is the one associated with the most likely set of correlation lengths. This differs from the `point_selection` option of the DAKOTA GP which initially chooses a well-spaced subset of points and finds the correlation parameters that are most likely for that one subset. It is similar to the DAKOTA GP's point selection in that the Surfpack GP will only interpolate the selected subset of points.

The `use_derivatives` keyword will cause the Surfpack GP to be constructed from a combination of function value and gradient information (if available). This is only beneficial when a function value plus a gradient can be computed at a computational cost comparable to a function value only. This rules out gradients computed by finite differences (which also tend to be insufficiently accurate). However, the accurate and inexpensive derivatives calculated by analytical, automatic differentiation, or continuous adjoint techniques can be cost effective and beneficial provided that they are not infinite or nearly so. Although a function value plus gradient can be often be obtained more cheaply than two function value only evaluations, each scalar derivative equation costs as much as a scalar function value equation during the construction of the emulator. Since the cost to build a GP is cubic in the number of equations, the cost to build a gradient-enhanced GP/Kriging model will be roughly (N+1)^3 times greater than one that is not gradient-enhanced, where N is the number of input dimensions. Also, a poorly spaced sample design has a greater adverse affect on the conditioning of the correlation matrix when derivative information is included. This can result in significantly more points being discarded and hence providing no benefit to the derivative enhanced GP/Kriging model's accuracy.

Tables 6.4 and 6.5 summarize the global approximation selection specifications.

Table 6.6 summarizes the remaining (optional) global approximation specifications, including controls for the number of points requested to build the surrogate, the source of the points (`dace_method_pointer` and `reuse_samples`), the use of derivative information in constructions (`use_derivatives`), calculation of fitness metrics, and the `correction` method.

The number of points used in building a global approximation is determined by three point counts:

- Model points: Each global surrogate type has an associated minimum reasonable and recommended number of build points, selected via `minimum_points` and `recommended_points` (default), respectively, within a global surrogate model specification. Alternately an integer `total_points` may be specified to request a specific number of points in each surrogate build (a lower bound of minimum points is enforced). This count, then, can be summarized as Model points = (minimum | recommended | max(minimum, total)).

- Reused points: any samples available for reuse via the `reuse_samples` keyword (described below) are always used to build. This count defaults to zero.

- Iterator points: `samples` specified within the DACE method specification (identified via `dace_method_pointer`) always provide a lower bound on the number of fresh DACE samples used to build the surrogate. This count defaults to zero.

The number of new points to be evaluated by the DACE method is defined by New points = max(Model points - Reused points, Iterator points), using the logic that the number of points needed is Model points minus Reused points, but respecting a lower bound DACE samples specification when present. The total number of points used in the surrogate build is then New points + Reused points. The DACE iterator will only be invoked if it has new samples to perform, and if new samples are required and no DACE iterator has been provided, an error will result.

The `dace_method_pointer` links to design of experiments iterator used to generate truth model data for building a global data fit. The `reuse_samples` specification can be used to employ old data (either from previous function evaluations performed in the run or from function evaluations read from a restart database or text file) in the building of new global approximations. The default is no reuse of old data (since this can induce directional bias), and the settings of `all`, `region`, and `points_file` result in reuse of all available data, reuse of all data available in the current trust region, and reuse of all data from a specified text file, respectively. Both `dace_method_pointer` and `reuse_samples` are optional specifications, which gives the user maximum flexibility in using design of experiments data, restart/text file data, or both. Note that the default format for a samples data file has changed in DAKOTA 5.2 (see User's Manual).

Description	Keyword	Associated Data	Status	Default
Global approximations	global	none	Required group (1 of 4 selections)	N/A
Polynomial	polynomial	linear | quadratic | cubic	Required group (1 of 7 selections)	N/A
DAKOTA Gaussian process	gaussian_-process	dakota	Required (1 of 7 selections)	N/A
GP trend function	trend	constant | linear | reduced_-quadratic	Optional	reduced_-quadratic trend
GP point selection	point_-selection	none	Optional	no point selection
Surfpack Gaussian process (Kriging)	gaussian_-process	surfpack	Required group (1 of 7 selections)	N/A
Surfpack GP trend function	trend	constant | linear | reduced_-quadratic | quadratic	Optional	reduced_-quadratic trend
Surfpack GP correlation lengths	correlation_-lengths	list of reals	Optional for none/local	internally computed correlation_-lengths
Surfpack GP optimization method	optimization_-method	'global' | 'local' | 'sampling' | 'none'	Optional	global
Surfpack GP maximum trials	max_trials	integer	Optional	
Surfpack GP gradient enhancement	use_-derivatives	none	Optional	no gradient enhancement
Artificial neural network	neural_-network	none	Required (1 of 7 selections)	N/A
ANN number nodes	nodes	integer	Optional	
ANN range	range	real	Optional	
ANN random weight	random_-weight	integer	Optional	

Table 33.2: Specification detail for global approximations: global approximation type, Part I

Description	Keyword	Associated Data	Status	Default
Multivariate adaptive regression splines	mars	none	Required (1 of 7 selections)	N/A
MARS maximum bases	max_bases	integer	Optional	
MARS interpolation	interpolation	linear | cubic	Optional	
Moving least squares	moving_least_squares	none	Required (1 of 7 selections)	N/A
MLS polynomial order	poly_order	integer	Optional	
MLS weight function	weight_function	integer	Optional	
Radial basis functions	radial_basis_functions	none	Required group (1 of 7 selections)	N/A
RBF number of bases	bases	integer	Optional	
RBF maximum points	max_pts	integer	Optional	
RBF minimum partitions	min_partition	integer	Optional	
RBF maximum subsets	bases	integer	Optional	

Table 33.3: Specification detail for global approximations: global approximation type, Part II

The `use_derivatives` flag specifies that any available derivative information should be used in global approximation builds. This capability is currently supported in **SurrBasedLocalMinimizer**, Pecos::SurrogateDataPoint, and **Approximation::build()**, and will soon be supported for `global_kriging` models, but is not yet supported in any global approximation derived class redefinitions of build().

To assess the goodness of fit of a global surrogate, a variety of diagnostics are available for the following global approximation methods: polynomial regressions, kriging, mars, moving least squares, neural networks, and radial basis functions. The diagnostics are specified by the keyword `diagnostics`, followed by a list of strings specifying the actual metrics. The diagnostic metrics available are: `sum_squared`, `mean_squared`, `root_mean_squared`, `max_squared`, `sum_scaled`, `max_scaled`, `sum_abs`, `mean_abs`, `max_abs`, `press`, `cv`, and `rsquared`. Most of these diagnostics refer to some operation on the residuals (the difference between the surrogate model and the truth model at the data points upon which the surrogate is built). For example, `sum_squared` refers to the sum of the squared residuals, `mean_abs` refers to the mean of the absolute value of the residuals, and `max_scaled` refers to the maximum scaled value of the residual, where the scaled value is taken by dividing the residual by the true function value at that point. In addition, `press` refers to a leave-one-out cross validation metric, and `cv` refers to a general cross validation metric. `rsquared` refers to the R-squared value typically used in regression analysis (the proportion of the variability in the response that can be accounted for by the surrogate model).

The `correction` specification specifies that the approximation will be corrected to match truth data, either matching truth values in the case of `zeroth_order` matching, matching truth values and gradients in the case of `first_order` matching, or matching truth values, gradients, and Hessians in the case of `second_order` matching. For `additive` and `multiplicative` corrections, the correction is local in that the truth data is matched at a single point, typically the center of the approximation region. The `additive` correction adds a scalar offset (`zeroth_order`), a linear function (`first_order`), or a quadratic function (`second_order`) to the approximation to match the truth data at the point, and the `multiplicative` correction multiplies the approximation by a scalar (`zeroth_order`), a linear function (`first_order`), or a quadratic function (`second_order`) to match the truth data at the point. The `additive first_order` case is due to [Lewis and Nash, 2000] and the `multiplicative first_order` case is commonly known as beta correction [Haftka, 1991]. For the `combined` correction, the use of both additive and multiplicative corrections allows the satisfaction of an additional matching condition, typically the truth function values at the previous correction point (e.g., the center of the previous trust region). The `combined` correction is then a multipoint correction, as opposed to the local `additive` and `multiplicative` corrections. Each of these correction capabilities is described in detail in [Eldred et al., 2004a].

33.2 Multipoint approximations

Multipoint approximations use data from previous design points to improve the accuracy of local approximations. Currently, the Two-point Adaptive Nonlinearity Approximation (TANA-3) method of [Xu and Grandhi, 1998] is supported. This method requires response value and gradient information from two points, and uses a first-order Taylor series if only one point is available. The truth model to be used to generate the value/gradient data used in the approximation is identified through the required `actual_model_pointer` specification. Table 6.7 summarizes the multipoint approximation specifications.

Description	Keyword	Associated Data	Status	Default
Number of build points	total_points | minimum_points | recommended_points	integer | none | none	Optional	recommended_points
Design of experiments method pointer	dace_method_pointer	string	Optional	no design of experiments data
Sample reuse in global approximation	reuse_samples	all | region | none	Optional group	all if points_file defined, none otherwise
File import of samples for global approximation builds	points_file	string	Optional	no point import from a file
Data file in annotated format	annotated	boolean	Optional	annotated
Data file in freeform format	freeform	boolean	Optional	annotated
Use of gradient data in global approximation builds	use_gradients	none	Optional	gradient data not used in global approximation builds
Print diagnostic metrics about the surrogate goodness of fit	diagnostics	string list (as many as desired): sum_squared, mean_squared, root_mean_squared, max_squared, sum_scaled, max_scaled, sum_abs, mean_abs, max_abs, press, cv, and rsquared	Optional	No diagnostics printed
Surrogate correction approach	correction	additive or multiplicative or combined, zeroth_order or first_order or second_order	Optional group	no surrogate correction

Table 33.4: Specification detail for global approximations: build and correction controls

Description	Keyword	Associated Data	Status	Default
Multipoint approximation	multipoint	none	Required group (1 of 4 selections)	N/A
Two-point adaptive nonlinear approximation	tana	none	Required	N/A
Pointer to the truth model specification	actual_-model_-pointer	string	Required	N/A

Table 33.5: Specification detail for multipoint approximations

33.3 Local approximations

Local approximations use value, gradient, and possibly Hessian data from a single point to form a series expansion for approximating data in the vicinity of this point. The currently available local approximation is the taylor_-series selection. The order of the Taylor series may be either first-order or second-order, which is automatically determined from the gradient and Hessian specifications in the responses specification (see Gradient Specification and Hessian Specification) for the truth model.

The truth model to be used to generate the value/gradient/Hessian data used in the series expansion is identified through the required actual_model_pointer specification. The use of a model pointer (as opposed to an interface pointer) allows additional flexibility in defining the approximation. In particular, the derivative specification for the truth model may differ from the derivative specification for the approximation , and the truth model results being approximated may involve a model recursion (e.g., the values/gradients from a nested model). Table 6.8 summarizes the local approximation interface specifications.

Description	Keyword	Associated Data	Status	Default
Local approximation	local	none	Required group (1 of 4 selections)	N/A
Taylor series local approximation	taylor_-series	none	Required	N/A
Pointer to the truth model specification	actual_-model_-pointer	string	Required	N/A

Table 33.6: Specification detail for local approximations

33.4 Hierarchical approximations

Hierarchical approximations use corrected results from a low fidelity model as an approximation to the results of a high fidelity "truth" model. These approximations are also known as model hierarchy, multifidelity, variable fidelity, and variable complexity approximations. The required low_fidelity_model_pointer specification points to the low fidelity model specification. This model is used to generate low fidelity responses which are then corrected and returned to an iterator. The required high_fidelity_model_pointer specification points to the specification for the high fidelity truth model. This model is used only for verifying low fidelity results and updating low fidelity corrections. The correction specification specifies which correction technique will be applied to the low fidelity results in order to match the high fidelity results at one or more points. In the

hierarchical case (as compared to the global case), the `correction` specification is required, since the omission of a correction technique would effectively eliminate the purpose of the high fidelity model. If it is desired to use a low fidelity model without corrections, then a hierarchical approximation is not needed and a `single` model should be used. Refer to Global approximations for additional information on available correction approaches. Table 6.9 summarizes the hierarchical approximation specifications.

Description	Keyword	Associated Data	Status	Default
Hierarchical approximation	`hierarchical`	none	Required group (1 of 4 selections)	N/A
Pointer to the low fidelity model specification	`low_-fidelity_-model_-pointer`	string	Required	N/A
Pointer to the high fidelity model specification	`high_-fidelity_-model_-pointer`	string	Required	N/A
Surrogate correction approach	`correction`	`additive` or `multiplicative` or `combined`, `zeroth_order` or `first_order` or `second_order`	Required group	N/A

Table 33.7: Specification detail for hierarchical approximations

Chapter 34

Nested Model Controls

Instead of appealing directly to a primary interface, a nested model maps parameters to responses by executing a sub-iterator for each evaluation of the nested model; the sub-iterator in turn operates on a sub-model. The sub-iterator responses may be combined with those from an optional interface specification.

In the `nested` model case, a `sub_method_pointer` must be provided in order to specify the nested iterator, and `optional_interface_pointer` and `optional_interface_responses_pointer` provide an optional group specification for the optional interface portion of nested models (where `optional_interface_pointer` points to the interface specification and `optional_interface_responses_pointer` points to a responses specification describing the data to be returned by this interface). This interface is used to provide non-nested data, which is then combined with data from the nested iterator using the `primary_response_mapping` and `secondary_response_mapping` inputs (see mapping discussion below).

Table 6.10 provides the specification detail for nested model pointers.

Description	Keyword	Associated Data	Status	Default
Optional interface set pointer	`optional_interface_pointer`	string	Optional group	no optional interface
Responses pointer for nested model optional interfaces	`optional_interface_responses_pointer`	string	Optional	reuse of top-level responses specification
Sub-method pointer for nested models	`sub_method_pointer`	string	Required	N/A

Table 34.1: Specification detail for nested models

Nested models may employ mappings for both the variable inputs to the sub-model and the response outputs from the sub-model. In the former case, the `primary_variable_mapping` and `secondary_variable_mapping` specifications are used to map from the active top-level variables into the sub-model variables, and in the latter case, the `primary_response_mapping` and `secondary_response_mapping` specifications are used to compute the sub-model response contributions to the top-level responses. For the variable mappings, the primary and secondary specifications provide lists of strings which are used to target specific sub-model vari-

ables and their sub-parameters, respectively. The primary strings are matched to continuous or discrete variable labels such as `'cdv_1'` (either user-supplied or default labels), and the secondary strings are matched to either real or integer random variable distribution parameters such as `'mean'` or `'num_trials'` (the form of the uncertain distribution parameter keyword that is appropriate for a single variable instance) or continuous or discrete design/state variable sub-parameters such as `'lower_bound'` or `'upper_bound'` (again, keyword form appropriate for a single variable instance). No coercion of types is supported, so real-valued top-level variables should map to either real-valued sub-model variables or real-valued sub-parameters and integer-valued top-level variables should map to either integer-valued sub-model variables or integer-valued sub-parameters. As long as these real versus integer constraints are satisfied, mappings are free to cross variable types (design, aleatory uncertain, epistemic uncertain, state) and domain types (continuous, discrete). Both `primary_variable_mapping` and `secondary_variable_mapping` specifications are optional, which is designed to support the following three possibilities:

1. If both primary and secondary variable mappings are specified, then an active top-level variable value will be inserted into the identified sub-parameter (the secondary mapping) for the identified sub-model variable (the primary mapping).
2. If a primary mapping is specified but a secondary mapping is not, then an active top-level variable value will be inserted into the identified sub-model variable value (the primary mapping).
3. If a primary mapping is not specified (corresponding secondary mappings, if specified, are ignored), then an active top-level variable value will be inserted into a corresponding sub-model variable, based on matching of variable types (e.g., top-level and sub-model variable specifications both allocate a set of `'continuous_design'` variables which are active at the top level). Multiple sub-model variable types may be updated in this manner, provided that they are all active in the top-level variables. Since there is a direct variable correspondence for these default insertions, sub-model bounds and labels are also updated from the top-level bounds and labels in order to eliminate the need for redundant input file specifications. Thus, it is typical for the sub-model variables specification to only contain the minimal required information, such as the number of variables, for these insertion targets. The sub-model must allocate enough space for each of the types that will accept default insertions, and the leading set of matching sub-model variables are updated (i.e., the sub-model may allocate more than needed and the trailing set will be unmodified).

These different variable mapping possibilities may be used in any combination by employing empty strings (″) for particular omitted mappings (the number of strings in user-supplied primary and secondary variable mapping specifications must equal the total number of active top-level variables, including both continuous and discrete types). The ordering of the active variables is the same as shown in dakota.input.summary and as presented in Variables Commands.

If inactive variables are present at the outer level, then the default type 3 mapping is used for these variables; that is, outer loop inactive variables are inserted into inner loop variables (active or inactive) based on matching of variable types, top-level bounds and labels are also propagated, the inner loop must allocate sufficient space to receive the outer loop values, and the leading subset within this inner loop allocation is updated. This capability is important for allowing nesting beyond two levels, since an active variable at the outer-most loop may become inactive at the next lower level, but still need to be further propagated down the recursion.

For the response mappings, the primary and secondary specifications provide real-valued multipliers to be applied to sub-iterator response results. The idea is that the responses from the inner loop are mapped to the outer loop. For example, if the nested model is an uncertainty quantification, aleatory statistics from the inner loop such as the mean inner loop response are mapped to the outer level, where they are treated epistemically, so that intervals on the mean (for example) are calculated. The response mapping defines a vector which multiplies the values from the inner loop to the outer loop. Each row of the mapping corresponds to one outer loop response, where

each column of the mapping corresponds to a value from the inner loop. Depending on the number of responses and the particular attributes calculated on the inner loop, there will be a vector of inner loop response values that need to be accounted for in the mapping. The sub-iterator response results are defined as follows for different sub-iterator types:

- optimization: the final objective function(s) and nonlinear constraints
- nonlinear least squares: the final least squares terms and nonlinear constraints
- aleatory uncertainty quantification (UQ): for each response function, a mean statistic, a standard deviation statistic, and all probability/reliability/generalized reliability/response level results for any user-specified `response_levels`, `probability_levels`, `reliability_levels`, and/or `gen_reliability_levels`, in that order.
- epistemic and mixed aleatory/epistemic UQ using interval estimation methods: lower and upper interval bounds for each response function.
- epistemic and mixed aleatory/epistemic UQ using evidence methods: for each response function, lower and upper interval bounds (belief and plausibility) for all probability/reliability/generalized reliability/response level results computed from any user-specified `response_levels`, `probability_levels`, `reliability_levels`, and/or `gen_reliability_levels`, in that order.
- parameter studies and design of experiments: for optimization and least squares response data sets, the best solution found (lowest constraint violation if infeasible, lowest composite objective function if feasible). For generic response data sets, a best solution metric is not defined, so sub-iterator response results are not defined in this case.

The primary values map sub-iterator response results into top-level objective functions, least squares terms, or generic response functions, depending on the declared top-level response set. The secondary values map sub-iterator response results into top-level nonlinear inequality and equality constraints. Refer to **NestedModel::response_mapping()** for additional details.

An example of variable and response mappings is provided below:

```
primary_variable_mapping   = ''  ''  'X'     'Y'
secondary_variable_mapping = ''  ''  'mean'  'mean'
primary_response_mapping   = 1. 0. 0. 0. 0. 0. 0. 0. 0.
secondary_response_mapping = 0. 0. 0. 1. 3. 0. 0. 0. 0.
                             0. 0. 0. 0. 0. 0. 1. 3. 0.
```

The variable mappings correspond to 4 top-level variables, the first two of which employ the default mappings from active top-level variables to sub-model variables of the same type (option 3 above) and the latter two of which are inserted into the mean distribution parameters of sub-model variables `'X'` and `'Y'` (option 1 above). In this particular example, there are 9 inner loop response attributes and 3 outer loop response functions (one primary response function and 2 secondary functions, such as one objective and two constraints). Each row of the response mapping is a vector which is multiplied (e.g. dot-product) against the 9 sub-iterator values to determine the outer loop function. For example, the primary response mapping only picks up the first value from the inner loop. This first value is the mean of the first response function on the inner loop. The response mappings correspond to 9 sub-iterator response functions (e.g., there are 3 response functions, each with a set of UQ final statistics: each has a mean, a standard deviation, and a level mapping in this example. If no probability or reliability levels are specified, the responses would only have a mean and standard deviation). The primary response mapping maps the first sub-iterator response function (mean) into a single objective function, least squares term, or generic response function

(as dictated by the top-level response specification), and the secondary response mapping maps the fourth sub-iterator response function plus 3 times the fifth sub-iterator response function (mean plus 3 standard deviations) into one top-level nonlinear constraint and the seventh sub-iterator response function plus 3 times the eighth sub-iterator response function (mean plus 3 standard deviations) into another top-level nonlinear constraint (these top-level nonlinear constraints may be inequality or equality, as dictated by the top-level response specification). Note that in many cases, each particular sub-iterator response will be mapped to a unique outer loop response (for example, in the nested UQ case where one wants to determine an interval on each inner loop statistic). In these cases, the response mapping will be the identity matrix. That is, the primary response mapping will have N rows and N columns corresponding to the N sub-iterator response values, and the mapping matrix will have a value of one along the diagonal and zeros elsewhere.

Table 6.11 provides the specification detail for the model independent controls involving nested model mappings.

Description	Keyword	Associated Data	Status	Default
Primary variable mappings for nested models	`primary_variable_mapping`	list of strings	Optional	default variable insertions based on variable type
Secondary variable mappings for nested models	`secondary_variable_mapping`	list of strings	Optional	primary mappings into sub-model variables are value-based
Primary response mappings for nested models	`primary_response_mapping`	list of reals	Optional	no sub-iterator contribution to primary functions
Secondary response mappings for nested models	`secondary_response_mapping`	list of reals	Optional	no sub-iterator contribution to secondary functions

Table 34.2: Specification detail for the model independent controls: nested model mappings

Chapter 35

Variables Commands

Chapter 36

Variables Description

The variables section in a DAKOTA input file specifies the parameter set to be iterated by a particular method. This parameter set is made up of design, uncertain, and state variables. Design variables can be continuous or discrete and consist of those variables which an optimizer adjusts in order to locate an optimal design. Each of the design parameters can have an initial point and a descriptive tag. Continuous and discrete range types include lower and upper bounds, and discrete set types include the admissible set values.

Uncertain variables may be categorized as either aleatory or epistemic and either continuous or discrete. Continuous aleatory uncertain variables include normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, and histogram bin distributions. Discrete aleatory uncertain variables include poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point distributions. In addition to aleatory uncertain variables defined by probability distributions, DAKOTA also supports epistemic uncertain variables that are non-probabilistic. The interval type specification is a continuous epistemic type that supports both simple bounded intervals as well as basic probability assignment (BPA) belief structures, where a BPA defines the uncertainty in a variable through providing one or more intervals in which the variable may lie along with varying levels of belief for each interval.

Each uncertain variable specification contains descriptive tags and most contain, either explicitly or implicitly, distribution lower and upper bounds. Distribution lower and upper bounds are explicit portions of the normal, lognormal, uniform, loguniform, triangular, and beta specifications, whereas they are implicitly defined for histogram bin, histogram point, and interval variables (from the extreme values within the bin/point/interval specifications) as well as for binomial (0 to `num_trials`) and hypergeometric (0 to min(`num_drawn`, `num_selected`)) variables. When used with design of experiments and multidimensional parameter studies, distribution bounds are also inferred for normal and lognormal (if optional bounds are unspecified) as well as for exponential, gamma, gumbel, frechet, weibull, poisson, negative binomial, and geometric (which have no bounds specifications); these bounds are [0, $\mu + 3\sigma$] for exponential, gamma, frechet, weibull, poisson, negative binomial, geometric, and unspecified lognormal, and [$\mu - 3\sigma$, $\mu + 3\sigma$] for gumbel and unspecified normal. For other types of parameter studies (vector and centered), an inferred initial starting point is needed for the uncertain variables. All uncertain variables are initialized to their means for these studies, where mean values for bounded normal and bounded lognormal may additionally be repaired to satisfy any specified distribution bounds, mean values for discrete integer range distributions are rounded down to the nearest integer, and mean values for discrete set distributions are rounded to the nearest set value.

In addition to tags and bounds specifications, normal variables include mean and standard deviation specifications, lognormal variables include lambda and zeta, mean and standard deviation, or mean and error factor specifications, triangular variables include mode specifications, exponential variables include beta specifications, beta, gamma,

gumbel, frechet, and weibull variables include alpha and beta specifications, histogram bin variables include abscissa and either ordinate or count specifications, poisson variables include lambda specifications, binomial and negative binomial variables include probability per trial and number of trials specifications, geometric variables include probability per trial specifications, hypergeometric variables include the specification of the total population, selected population, and number drawn, histogram point variables include abscissa and count specifications, and interval variables include basic probability assignments per interval.

State variables can be continuous or discrete and consist of "other" variables which are to be mapped through the simulation interface. Each state variable specification can have an initial state and descriptors. Continuous and discrete range types include lower and upper bounds, and discrete set types include the admissible set values. State variables provide a convenient mechanism for parameterizing additional model inputs, such as mesh density, simulation convergence tolerances and time step controls, and can be used to enact model adaptivity in future strategy developments.

The ordering of variables is important, and a consistent ordering is employed throughout the DAKOTA software. It is the same ordering as shown in dakota.input.summary and as presented in the outline of this chapter. That ordering can be summarized as continuous followed by discrete integer followed by discrete real within each of the following types: design, aleatory uncertain, epistemic uncertain, and state. Ordering of variable types below this granularity (e.g., from normal to histogram bin within continuous aleatory uncertain) is defined somewhat arbitrarily, but is enforced consistently throughout the code.

Several examples follow. In the first example, two continuous design variables are specified:

```
variables,
continuous_design = 2
  initial_point    0.9    1.1
  upper_bounds     5.8    2.9
  lower_bounds     0.5   -2.9
  descriptors   'radius' 'location'
```

In the next example, defaults are employed. In this case, `initial_point` will default to a vector of `0.` values, `upper_bounds` will default to vector values of `DBL_MAX` (the maximum number representable in double precision for a particular platform, as defined in the platform's `float.h` C header file), `lower_bounds` will default to a vector of `-DBL_MAX` values, and `descriptors` will default to a vector of `'cdv_i'` strings, where `i` ranges from one to two:

```
variables,
continuous_design = 2
```

In the following example, the syntax for a normal-lognormal distribution is shown. One normal and one lognormal uncertain variable are completely specified by their means and standard deviations. In addition, the dependence structure between the two variables is specified using the `uncertain_correlation_matrix`.

```
variables,
        normal_uncertain    =  1
          means             =  1.0
          std_deviations    =  1.0
          descriptors       =  'TF1n'
        lognormal_uncertain =  1
          means             =  2.0
          std_deviations    =  0.5
          descriptors       =  'TF2ln'
        uncertain_correlation_matrix =  1.0 0.2
                                        0.2 1.0
```

An example of the syntax for a state variables specification follows:

```
variables,
        continuous_state = 1
          initial_state       4.0
          lower_bounds        0.0
          upper_bounds        8.0
          descriptors        'CS1'
        discrete_state_range = 1
          initial_state       104
          lower_bounds        100
          upper_bounds        110
          descriptors        'DS1'
```

And in a more advanced example, a variables specification containing a set identifier, continuous and discrete design variables, normal and uniform uncertain variables, and continuous and discrete state variables is shown:

```
variables,
id_variables = 'V1'
continuous_design = 2
  initial_point    0.9    1.1
  upper_bounds     5.8    2.9
  lower_bounds     0.5   -2.9
  descriptors   'radius' 'location'
discrete_design_range = 1
  initial_point    2
  upper_bounds     1
  lower_bounds     3
  descriptors   'material'
normal_uncertain = 2
  means          =  248.89, 593.33
  std_deviations =   12.4,   29.7
  descriptors    =  'TF1n'   'TF2n'
uniform_uncertain = 2
  lower_bounds =  199.3,  474.63
  upper_bounds =  298.5,  712.
  descriptors  =  'TF1u'   'TF2u'
continuous_state = 2
  initial_state = 1.e-4  1.e-6
  descriptors   = 'EPSIT1' 'EPSIT2'
discrete_state_set_int = 1
  initial_state = 100
  set_values    = 100 212 375
  descriptors   = 'load_case'
```

Refer to the DAKOTA Users Manual [Adams et al., 2010] for discussion on how different iterators view these mixed variable sets.

Chapter 37

Variables Specification

The variables specification has the following structure:

```
variables,
<set identifier>
<continuous design variables specification>
<discrete design range variables specification>
<discrete design set integer variables specification>
<discrete design set real variables specification>
<normal uncertain variables specification>
<lognormal uncertain variables specification>
<uniform uncertain variables specification>
<loguniform uncertain variables specification>
<triangular uncertain variables specification>
<exponential uncertain variables specification>
<beta uncertain variables specification>
<gamma uncertain variables specification>
<gumbel uncertain variables specification>
<frechet uncertain variables specification>
<weibull uncertain variables specification>
<histogram bin uncertain variables specification>
<poisson uncertain variables specification>
<binomial uncertain variables specification>
<negative binomial uncertain variables specification>
<geometric uncertain variables specification>
<hypergeometric uncertain variables specification>
<histogram point uncertain variables specification>
<uncertain correlation specification>
<interval uncertain variables specification>
<continuous state variables specification>
<discrete state range variables specification>
<discrete state set integer variables specification>
<discrete state set real variables specification>
```

Referring to dakota.input.summary, it is evident from the enclosing brackets that the set identifier specification, the uncertain correlation specification, and each of the variables specifications are all optional. The set identifier and uncertain correlation are stand-alone optional specifications, whereas the variables specifications are optional group specifications, meaning that the group can either appear or not as a unit. If any part of an optional group is specified, then all required parts of the group must appear.

The optional status of the different variable type specifications allows the user to specify only those variables which are present (rather than explicitly specifying that the number of a particular type of variables is zero). However, at least one type of variables that are active for the iterator in use must have nonzero size or an input error message will result. The following sections describe each of these specification components in additional detail.

Chapter 38

Variables Set Identifier

The optional set identifier specification uses the keyword `id_variables` to input a unique string for use in identifying a particular variables set. A model can then identify the use of this variables set by specifying the same string in its `variables_pointer` specification (see Model Independent Controls). For example, a model whose specification contains `variables_pointer = 'V1'` will use a variables specification containing the set identifier `id_variables = 'V1'`.

If the `id_variables` specification is omitted, a particular variables set will be used by a model only if that model omits specifying a `variables_pointer` and if the variables set was the last set parsed (or is the only set parsed). In common practice, if only one variables set exists, then `id_variables` can be safely omitted from the variables specification and `variables_pointer` can be omitted from the model specification(s), since there is no potential for ambiguity in this case. Table 7.1 summarizes the set identifier inputs.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Variables set identifier	`id_variables`	string	Optional	use of last variables parsed

Table 38.1: Specification detail for set identifier

Chapter 39

Design Variables

Design variable types include continuous real, discrete range of integer values (contiguous integers), discrete set of integer values, and discrete set of real values. Within each optional design variables specification group, the number of variables is always required. The following Tables 7.2 through 7.5 summarize the required and optional specifications for each design variable subtype. The `initial_point` specifications provide the point in design space from which an iterator is started and default to either zeros (continuous and discrete range) or middle values (discrete sets). The `descriptors` specifications supply strings which will be replicated through the DAKOTA output to identify the numerical values for these parameters; these default to numbered strings.

For continuous and discrete range variables, the `lower_bounds` and `upper_bounds` restrict the size of the feasible design space and are frequently used to prevent nonphysical designs. Default values are positive and negative machine limits for upper and lower bounds (+/- `DBL_MAX`, `INT_MAX`, `INT_MIN` from the `float.h` and `limits.h` system header files). As for linear and nonlinear inequality constraint bounds (see Method Independent Controls and Objective and constraint functions (optimization data set)), a nonexistent upper bound can be specified by using a value greater than the "big bound size" constant (1.e+30 for continuous variables, 1.e+9 for discrete integer variables) and a nonexistent lower bound can be specified by using a value less than the negation of these constants (-1.e+30 for continuous, -1.e+9 for discrete integer), although not all optimizers currently support this feature (e.g., DOT and CONMIN will treat these large bound values as actual variable bounds, but this should not be problematic in practice).

39.1 Continuous Design Variables

For continuous variables, the `scale_types` specification includes strings specifying the scaling type for each component of the continuous design variables vector in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `scale_types` may be selected from `'none'`, `'value'`, `'auto'`, or `'log'`, to select no, characteristic value, automatic, or logarithmic scaling, respectively. If a single string is specified it will apply to all components of the continuous design variables vector. Each entry in `scales` may be a user-specified nonzero real characteristic value to be used in scaling each variable component. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'` and `'log'`. If a single real value is specified it will apply to all components of the continuous design variables vector.

Description	Keyword	Associated Data	Status	Default
Continuous design variables	continuous_-design	integer	Optional group	no continuous design variables
Initial point	initial_-point	list of reals	Optional	vector values = 0. (repaired to bounds, if required)
Lower bounds	lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Upper bounds	upper_bounds	list of reals	Optional	vector values = +DBL_MAX
Scaling types	scale_types	list of strings	Optional	vector values = 'none'
Scales	scales	list of reals	Optional	vector values = 1. (no scaling)
Descriptors	descriptors	list of strings	Optional	vector of 'cdv_i' where i = 1,2,3...

Table 39.1: Specification detail for continuous design variables

39.2 Discrete Design Range Variables

Description	Keyword	Associated Data	Status	Default
Discrete design range variables	discrete_-design_range	integer	Optional group	no discrete design variables
Initial point	initial_-point	list of integers	Optional	vector values = 0 (repaired to bounds, if required)
Lower bounds	lower_bounds	list of integers	Optional	vector values = INT_MIN
Upper bounds	upper_bounds	list of integers	Optional	vector values = INT_MAX
Descriptors	descriptors	list of strings	Optional	vector of 'ddriv_i' where i = 1,2,3,...

Table 39.2: Specification detail for discrete design range variables

39.3 Discrete Design Integer Set Variables

Discrete set variables are specified with an integer list specifying how many set members there are for each variable and a list of integer or real set values for discrete_design_set_integer (Table 7.4) and discrete_design_set_real (Table 7.5), respectively.

Description	Keyword	Associated Data	Status	Default
Discrete design set of integer variables	`discrete_-design_set_-integer`	integer	Optional group	no discrete design set of integer variables
Initial point	`initial_-point`	list of integers	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	`num_set_-values`	list of integers	Optional	equal distribution
Set values	`set_values`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'ddsiv_i'` where `i = 1,2,3,...`

Table 39.3: Specification detail for discrete design set of integer variables

39.4 Discrete Design Real Set Variables

Description	Keyword	Associated Data	Status	Default
Discrete design set of real variables	`discrete_-design_set_-real`	integer	Optional group	no discrete design set of real variables
Initial point	`initial_-point`	list of reals	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	`num_set_-values`	list of integers	Optional	equal distribution
Set values	`set_values`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'ddsrv_i'` where `i = 1,2,3,...`

Table 39.4: Specification detail for discrete design set of real variables

Chapter 40

Aleatory Uncertain Variables

Aleatory uncertain variables involve continuous or discrete probability distribution specifications. Continuous probability distributions including normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, and histogram bin distributions. Discrete probability distributions include poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point distributions. Each of these specifications is an optional group specification.

These specifications of probability distributions directly support the use of probabilistic uncertainty quantification methods such as sampling, reliability, and stochastic expansion methods. However, the inclusion of lower and upper distribution bounds for all uncertain variable types (either explicitly defined, implicitly defined, or inferred; see Variables Description) also allows the use of these variables within methods that rely on a bounded region to define a set of function evaluations (i.e., design of experiments and some parameter study methods). Each distribution also provides optional uncertain variable descriptors (default values are numbered strings) that supply identifiers that help associate the numerical values with the uncertain parameters as they appear within the DAKOTA output. Tables 7.6 through 7.23 summarize the details of the aleatory uncertain variable specifications.

40.1 Normal Distribution

Within the normal uncertain optional group specification, the number of normal uncertain variables, the means, and standard deviations are required specifications, and the distribution lower and upper bounds and variable descriptors are optional specifications. The normal distribution is widely used to model uncertain variables such as population characteristics. It is also used to model the mean of a sample: as the sample size becomes very large, the Central Limit Theorem states that the mean becomes approximately normal, regardless of the distribution of the original variables.

The density function for the normal distribution is:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_N} e^{-\frac{1}{2}\left(\frac{x-\mu_N}{\sigma_N}\right)^2}$$

where μ_N and σ_N are the mean and standard deviation of the normal distribution, respectively.

Note that if you specify bounds for a normal distribution, the sampling occurs from the underlying distribution with the given mean and standard deviation, but samples are not taken outside the bounds (see "bounded normal" distribution type in [Wyss and Jorgensen, 1998]). This can result in the mean and the standard deviation of the

sample data being different from the mean and standard deviation of the underlying distribution. For example, if you are sampling from a normal distribution with a mean of 5 and a standard deviation of 3, but you specify bounds of 1 and 7, the resulting mean of the samples will be around 4.3 and the resulting standard deviation will be around 1.6. This is because you have bounded the original distribution significantly, and asymetrically, since 7 is closer to the original mean than 1.

Description	Keyword	Associated Data	Status	Default
normal uncertain variables	`normal_-uncertain`	integer	Optional group	no normal uncertain variables
normal uncertain means	`means`	list of reals	Required	N/A
normal uncertain standard deviations	`std_-deviations`	list of reals	Required	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Distribution upper bounds	`upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'nuv_i'` where `i = 1,2,3,...`

Table 40.1: Specification detail for normal uncertain variables

40.2 Lognormal Distribution

If the logarithm of an uncertain variable X has a normal distribution, that is $\log X \sim N(\mu, \sigma)$, then X is distributed with a lognormal distribution. The lognormal is often used to model time to perform some task. It can also be used to model variables which are the product of a large number of other quantities, by the Central Limit Theorem. Finally, the lognormal is used to model quantities which cannot have negative values. Within the lognormal uncertain optional group specification, the number of lognormal uncertain variables, the means, and either standard deviations or error factors must be specified, and the distribution lower and upper bounds and variable descriptors are optional specifications. These distribution bounds can be used to truncate the tails of lognormal distributions, which as for bounded normal, can result in the mean and the standard deviation of the sample data being different from the mean and standard deviation of the underlying distribution (see "bounded lognormal" and "bounded lognormal-n" distribution types in [Wyss and Jorgensen, 1998]).

For the lognormal variables, one may specify either the mean μ and standard deviation σ of the actual lognormal distribution, the mean μ and error factor ϵ of the actual lognormal distribution, or the mean λ ("lambda") and standard deviation ζ ("zeta") of the underlying normal distribution. The conversion equations from lognormal mean μ and either lognormal error factor ϵ or lognormal standard deviation σ to the mean λ and standard deviation ζ of the underlying normal distribution are as follows:

$$\zeta = \frac{ln(\epsilon)}{1.645}$$

$$\zeta^2 = ln(\frac{\sigma^2}{\mu^2} + 1)$$

$$\lambda = ln(\mu) - \frac{\zeta^2}{2}$$

Conversions from λ and ζ back to μ and ϵ or σ are as follows:

$$\mu = e^{\lambda + \frac{\zeta^2}{2}}$$
$$\sigma^2 = e^{2\lambda + \zeta^2}(e^{\zeta^2} - 1)$$
$$\epsilon = e^{1.645\zeta}$$

The density function for the lognormal distribution is:

$$f(x) = \frac{1}{\sqrt{2\pi}\zeta x} e^{-\frac{1}{2}\left(\frac{lnx - \lambda}{\zeta}\right)^2}$$

Description	Keyword	Associated Data	Status	Default
lognormal uncertain variables	`lognormal_-uncertain`	integer	Optional group	no lognormal uncertain variables
lognormal uncertain means	`means`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain standard deviations	`std_-deviations`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain error factors	`error_-factors`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain lambdas	`lambdas`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain zetas	`zetas`	list of reals	Required (1 of 3 selections)	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Optional	vector values = `0.`
Distribution upper bounds	`upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'lnuv_i'` where `i = 1,2,3,...`

Table 40.2: Specification detail for lognormal uncertain variables

40.3 Uniform Distribution

Within the uniform uncertain optional group specification, the number of uniform uncertain variables and the distribution lower and upper bounds are required specifications, and variable descriptors is an optional specification. The uniform distribution has the density function:

$$f(x) = \frac{1}{U_U - L_U}$$

where U_U and L_U are the upper and lower bounds of the uniform distribution, respectively. The mean of the uniform distribution is $\frac{U_U + L_U}{2}$ and the variance is $\frac{(U_U - L_U)^2}{12}$. Note that this distribution is a special case of the more general beta distribution.

Description	Keyword	Associated Data	Status	Default
uniform uncertain variables	`uniform_-uncertain`	integer	Optional group	no uniform uncertain variables
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'uuv_i'` where `i = 1,2,3,...`

Table 40.3: Specification detail for uniform uncertain variables

40.4 Loguniform Distribution

If the logarithm of an uncertain variable X has a uniform distribution, that is $\log X \sim U(L_{LU}, U_{LU})$, then X is distributed with a loguniform distribution. Within the loguniform uncertain optional group specification, the number of loguniform uncertain variables and the distribution lower and upper bounds are required specifications, and variable descriptors is an optional specification. The loguniform distribution has the density function:

$$f(x) = \frac{1}{x(lnU_{LU} - lnL_{LU})}$$

Description	Keyword	Associated Data	Status	Default
loguniform uncertain variables	`loguniform_-uncertain`	integer	Optional group	no loguniform uncertain variables
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'luuv_i'` where `i = 1,2,3,...`

Table 40.4: Specification detail for loguniform uncertain variables

40.5 Triangular Distribution

The triangular distribution is often used when one does not have much data or information, but does have an estimate of the most likely value and the lower and upper bounds. Within the triangular uncertain optional group specification, the number of triangular uncertain variables, the modes, and the distribution lower and upper bounds are required specifications, and variable descriptors is an optional specification.

The density function for the triangular distribution is:

$$f(x) = \frac{2(x - L_T)}{(U_T - L_T)(M_T - L_T)}$$

if $L_T \leq x \leq M_T$, and

$$f(x) = \frac{2(U_T - x)}{(U_T - L_T)(U_T - M_T)}$$

if $M_T \leq x \leq U_T$, and 0 elsewhere. In these equations, L_T is the lower bound, U_T is the upper bound, and M_T is the mode of the triangular distribution.

Description	Keyword	Associated Data	Status	Default
triangular uncertain variables	`triangular_uncertain`	integer	Optional group	no triangular uncertain variables
triangular uncertain modes	`modes`	list of reals	Required	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'tuv_i'` where `i = 1,2,3,...`

Table 40.5: Specification detail for triangular uncertain variables

40.6 Exponential Distribution

The exponential distribution is often used for modeling failure rates. Within the exponential uncertain optional group specification, the number of exponential uncertain variables and the beta parameters are required specifications, and variable descriptors is an optional specification.

The density function for the exponential distribution is given by:

$$f(x) = \frac{1}{\beta} e^{\frac{-x}{\beta}}$$

where $\mu_E = \beta$ and $\sigma_E^2 = \beta^2$. Note that this distribution is a special case of the more general gamma distribution.

Description	Keyword	Associated Data	Status	Default
exponential uncertain variables	`exponential_uncertain`	integer	Optional group	no exponential uncertain variables
exponential uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'euv_i'` where `i = 1,2,3,...`

Table 40.6: Specification detail for exponential uncertain variables

40.7 Beta Distribution

Within the beta uncertain optional group specification, the number of beta uncertain variables, the alpha and beta parameters, and the distribution upper and lower bounds are required specifications, and the variable descriptors is an optional specification. The beta distribution can be helpful when the actual distribution of an uncertain variable is unknown, but the user has a good idea of the bounds, the mean, and the standard deviation of the uncertain variable. The density function for the beta distribution is

$$f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \frac{(x-L_B)^{\alpha-1}(U_B-x)^{\beta-1}}{(U_B-L_B)^{\alpha+\beta-1}}$$

where $\Gamma(\alpha)$ is the gamma function and $B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$ is the beta function. To calculate mean and standard deviation from the alpha, beta, upper bound, and lower bound parameters of the beta distribution, the following expressions may be used.

$$\mu_B = L_B + \frac{\alpha}{\alpha+\beta}(U_B - L_B)$$

$$\sigma_B^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}(U_B-L_B)^2$$

Solving these for α and β gives:

$$\alpha = (\mu_B - L_B)\frac{(\mu_B-L_B)(U_B-\mu_B)-\sigma_B^2}{\sigma_B^2(U_B-L_B)}$$

$$\beta = (U_B - \mu_B)\frac{(\mu_B-L_B)(U_B-\mu_B)-\sigma_B^2}{\sigma_B^2(U_B-L_B)}$$

Note that the uniform distribution is a special case of this distribution for parameters $\alpha = \beta = 1$.

Description	Keyword	Associated Data	Status	Default
beta uncertain variables	`beta_uncertain`	integer	Optional group	no beta uncertain variables
beta uncertain alphas	`alphas`	list of reals	Required	N/A
beta uncertain betas	`betas`	list of reals	Required	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'buv_i'` where `i = 1,2,3,...`

Table 40.7: Specification detail for beta uncertain variables

40.8 Gamma Distribution

The gamma distribution is sometimes used to model time to complete a task, such as a repair or service task. It is a very flexible distribution. Within the gamma uncertain optional group specification, the number of gamma

uncertain variables and the alpha and beta parameters are required specifications, and variable descriptors is an optional specification.

The density function for the gamma distribution is given by:

$$f(x) = \frac{x^{\alpha-1} e^{\frac{-x}{\beta}}}{\beta^{\alpha}\Gamma(\alpha)}$$

where $\mu_{GA} = \alpha\beta$ and $\sigma^2_{GA} = \alpha\beta^2$. Note that the exponential distribution is a special case of this distribution for parameter $\alpha = 1$.

Description	Keyword	Associated Data	Status	Default
gamma uncertain variables	`gamma_-uncertain`	integer	Optional group	no gamma uncertain variables
gamma uncertain alphas	`alphas`	list of reals	Required	N/A
gamma uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'gauv_i'` where `i = 1,2,3,...`

Table 40.8: Specification detail for gamma uncertain variables

40.9 Gumbel Distribution

Within the gumbel optional uncertain group specification, the number of gumbel uncertain variables, and the alpha and beta parameters are required specifications. The Gumbel distribution is also referred to as the Type I Largest Extreme Value distribution. The distribution of maxima in sample sets from a population with a normal distribution will asymptotically converge to this distribution. It is commonly used to model demand variables such as wind loads and flood levels.

The density function for the Gumbel distribution is given by:

$$f(x) = \alpha e^{-\alpha(x-\beta)} exp(-e^{-\alpha(x-\beta)})$$

where $\mu_{GU} = \beta + \frac{0.5772}{\alpha}$ and $\sigma_{GU} = \frac{\pi}{\sqrt{6}\alpha}$.

40.10 Frechet Distribution

With the frechet uncertain optional group specification, the number of frechet uncertain variables and the alpha and beta parameters are required specifications. The Frechet distribution is also referred to as the Type II Largest Extreme Value distribution. The distribution of maxima in sample sets from a population with a lognormal distribution will asymptotically converge to this distribution. It is commonly used to model non-negative demand variables.

The density function for the frechet distribution is:

$$f(x) = \frac{\alpha}{\beta}(\frac{\beta}{x})^{\alpha+1} e^{-(\frac{\beta}{x})^{\alpha}}$$

Description	Keyword	Associated Data	Status	Default
gumbel uncertain variables	`gumbel_-uncertain`	integer	Optional group	no gumbel uncertain variables
gumbel uncertain alphas	`alphas`	list of reals	Required	N/A
gumbel uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'guuv_i'` where `i = 1,2,3,...`

Table 40.9: Specification detail for gumbel uncertain variables

where $\mu_F = \beta\Gamma(1 - \frac{1}{\alpha})$ and $\sigma_F^2 = \beta^2[\Gamma(1 - \frac{2}{\alpha}) - \Gamma^2(1 - \frac{1}{\alpha})]$

Description	Keyword	Associated Data	Status	Default
frechet uncertain variables	`frechet_-uncertain`	integer	Optional group	no frechet uncertain variables
frechet uncertain alphas	`alphas`	list of reals	Required	N/A
frechet uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'fuv_i'` where `i = 1,2,3,...`

Table 40.10: Specification detail for frechet uncertain variables

40.11 Weibull Distribution

The Weibull distribution is commonly used in reliability studies to predict the lifetime of a device. Within the weibull uncertain optional group specification, the number of weibull uncertain variables and the alpha and beta parameters are required specifications. The Weibull distribution is also referred to as the Type III Smallest Extreme Value distribution. It is also used to model capacity variables such as material strength.

The density function for the weibull distribution is given by:

$$f(x) = \frac{\alpha}{\beta}\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-\left(\frac{x}{\beta}\right)^{\alpha}}$$

where $\mu_W = \beta\Gamma(1 + \frac{1}{\alpha})$ and $\sigma_W = \sqrt{\frac{\Gamma(1+\frac{2}{\alpha})}{\Gamma^2(1+\frac{1}{\alpha})} - 1}\mu_W$

40.12 Histogram Bin Distribution

Histogram uncertain variables are typically used to model a set of empirical data. A bin histogram is a continuous aleatory distribution that allows the user to specify bins of non-zero width (where the uncertain variable may lie)

Description	Keyword	Associated Data	Status	Default
weibull uncertain variables	`weibull_-uncertain`	integer	Optional group	no weibull uncertain variables
weibull uncertain alphas	`alphas`	list of reals	Required	N/A
weibull uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'wuv_i'` where `i = 1,2,3,...`

Table 40.11: Specification detail for weibull uncertain variables

along with the relative frequencies that are associated with each bin.

Within the histogram bin uncertain optional group specification, the number of histogram bin uncertain variables is a required specification, the number of pairs is an optional key for apportionment of abscissas/ordinates/counts, specification of abscissas and either ordinates or counts is required, and the variable descriptors is an optional specification. When using a histogram bin variable, one must define at least one bin (with two bounding value pairs).

The `abscissas` specification define abscissa values ("x" coordinates) for the PDF of each histogram variable. When paired with `counts`, the specifications provide sets of (x,c) pairs for each histogram variable where `c` defines a count (i.e., a frequency or relative probability) associated with a bin. If using bins of unequal width and specification of probability densities is more natural, then the `counts` specification can be replaced with a `ordinates` specification ("y" coordinates) in order to support interpretation of the input as (x,y) pairs defining the profile of a "skyline" PDF. Conversion between the two specifications is straightforward: a count/frequency is a cumulative probability quantity defined from the product of the ordinate density value and the `x` bin width. Thus, in the cases of bins of equal width, ordinate and count specifications are equivalent. In addition, ordinates and counts may be relative values; it is not necessary to scale them as all user inputs will be normalized.

To fully specify a bin-based histogram with `n` bins (potentially of unequal width), `n+1` (x,c) or (x,y) pairs must be specified with the following features:

- `x` is the parameter value for the left boundary of a histogram bin and `c` is the corresponding count for that bin. Alternatively, `y` defines the ordinate density value for this bin within a skyline PDF. The right boundary of the bin is defined by the left boundary of the next pair.
- the final pair specifies the right end of the last bin and must have a `c` or `y` value of zero.
- the `x` values must be strictly increasing.
- all `c` or `y` values must be positive, except for the last which must be zero.
- a minimum of two pairs must be specified for each bin-based histogram.

The number of pairs specifications provide for the proper association of multiple sets of (x,c) or (x,y) pairs with individual histogram variables. For example, in the following specification

```
histogram_bin_uncertain = 2
  num_pairs =        3           4
  abscissas =  5  8 10 .1 .2 .3 .4
  counts    = 17 21  0 12 24 12  0
```

`num_pairs` associates the first 3 (x,c) pairs from `abscissas` and `counts` ((5,17),(8,21),(10,0)) with one bin-based histogram variable, where one bin is defined between 5 and 8 with a count of 17 and another bin is defined between 8 and 10 with a count of 21. The following set of 4 (x,c) pairs ((.1,12),(.2,24),(.3,12),(.4,0)) defines a second bin-based histogram variable containing three equal-width bins with counts 12, 24, and 12 (middle bin is twice as probable as the other two).

Description	Keyword	Associated Data	Status	Default
histogram bin uncertain variables	`histogram_-bin_-uncertain`	integer	Optional group	no histogram bin uncertain variables
key to apportionment among bin-based histogram variables	`num_pairs`	list of integers	Optional	equal distribution
sets of abscissas for bin-based histogram variables	`abscissas`	list of reals	Required	N/A
sets of ordinates for bin-based histogram variables	`ordinates`	list of reals	Required (1 of 2 selections)	N/A
sets of counts for bin-based histogram variables	`counts`	list of reals	Required (1 of 2 selections)	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'hubv_i'` where `i = 1,2,3,...`

Table 40.12: Specification detail for histogram bin uncertain variables

40.13 Poisson Distribution

The Poisson distribution is used to predict the number of discrete events that happen in a given time interval. The expected number of occurences in the time interval is λ, which must be a positive real number. For example, if events occur on average 4 times per year and we are interested in the distribution of events over six months, λ would be 2 in this case. However, if we were interested in the distribution of events occuring over 5 years, λ would be 20.

The density function for the poisson distribution is given by:

$$f(x) = \frac{\lambda e^{-\lambda}}{x!}$$

where λ is the frequency of events happening, and `x` is the number of events that occur. The poisson distribution returns samples representing number of occurrences in the time period of interest.

Description	Keyword	Associated Data	Status	Default
poisson uncertain variables	`poisson_-uncertain`	integer	Optional group	no poisson uncertain variables
poisson uncertain lambdas	`lambdas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'puv_i'` where `i = 1,2,3,...`

Table 40.13: Specification detail for poisson uncertain variables

40.14 Binomial Distribution

The binomial distribution is typically used to predict the number of failures (or defective items or some type of event) in a total of `n` independent tests or trials, where each trial has the probability `p` of failing or being defective. Each particular test can be considered as a Bernoulli trial.

The density function for the binomial distribution is given by:

$$f(x) = \left(\begin{array}{c} n \\ x \end{array} \right) p^x (1-p)^{(n-x)}$$

where `p` is the probability of failure per trial and `n` is the number of trials.

Description	Keyword	Associated Data	Status	Default
binomial uncertain variables	`binomial_-uncertain`	integer	Optional group	no binomial uncertain variables
binomial uncertain prob_per_trial	`prob_per_-trial`	list of reals	Required	N/A
binomial uncertain num_trials	`num_trials`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'biuv_i'` where `i = 1,2,3,...`

Table 40.14: Specification detail for binomial uncertain variables

40.15 Negative Binomial Distribution

The negative binomial distribution is typically used to predict the number of times to perform a test to have a total of `n` successes, where each test has a probability `p` of success.

The density function for the negative binomial distribution is given by:

$$f(x) = \left(\begin{array}{c} n+x-1 \\ x \end{array} \right) p^n (1-p)^x$$

where `p` is the probability of success per trial and `n` is the number of successful trials.

Description	Keyword	Associated Data	Status	Default
negative binomial uncertain variables	negative_binomial_uncertain	integer	Optional group	no negative binomial uncertain variables
negative binomial uncertain success prob_per_trial	prob_per_trial	list of reals	Required	N/A
negative binomial uncertain success num_trials	num_trials	list of integers	Required	N/A
Descriptors	descriptors	list of strings	Optional	vector of 'nbuv_i' where i = 1,2,3,...

Table 40.15: Specification detail for negative binomial uncertain variables

40.16 Geometric Distribution

The geometric distribution represents the number of successful trials that might occur before a failure is observed. The density function for the geometric distribution is given by:

$$f(x) = p(1-p)^x$$

where p is the probability of failure per trial.

Description	Keyword	Associated Data	Status	Default
geometric uncertain variables	geometric_uncertain	integer	Optional group	no geometric uncertain variables
geometric uncertain prob_per_trial	prob_per_trial	list of reals	Required	N/A
Descriptors	descriptors	list of strings	Optional	vector of 'geuv_i' where i = 1,2,3,...

Table 40.16: Specification detail for geometric uncertain variables

40.17 Hypergeometric Distribution

The hypergeometric distribution is used to define the number of failures (or the number of successes; the number of some type of event) in a set of tests that has a known proportion of failures. The hypergeometric is often described using an urn model. For example, say we have a total population containing N balls, and we know that m of the balls are white and the remaining balls are green. If we draw n balls from the urn without replacement, the hypergeometric distribution describes the distribution of the number of white balls drawn from the urn.

The density function for the hypergeometric distribution is given by:

$$f(x) = \frac{\begin{pmatrix} m \\ x \end{pmatrix} \begin{pmatrix} N-m \\ n-x \end{pmatrix}}{\begin{pmatrix} N \\ n \end{pmatrix}}$$

where N is the total population, m is the number of items in the selected population (e.g. the number of white balls in the full urn of N items), and n is the number of balls drawn.

Description	Keyword	Associated Data	Status	Default
hypergeometric uncertain variables	`hypergeometric_-uncertain`	integer	Optional group	no hypergeometric uncertain variables
hypergeometric uncertain total_population	`total_-population`	list of integers	Required	N/A
hypergeometric uncertain selected_-population	`selected_-population`	list of integers	Required	N/A
hypergeometric uncertain num_drawn	`num_drawn`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'hguv_i'` where `i = 1,2,3,...`

Table 40.17: Specification detail for hypergeometric uncertain variables

40.18 Histogram Point Distribution

As mentioned above, histogram uncertain variables are typically used to model a set of empirical data. A point histogram is a discrete aleatory distribution that allows the user to specify a set of real-valued points and associated frequency values.

Point histograms are similar to Discrete Design Real Set Variables and Discrete State Real Set Variables, but differ in the inclusion of information on the relative probabilities of observing the different values within the set.

Within the histogram point uncertain optional group specification, the number of histogram point uncertain variables is a required specification, the number of pairs is an optional key for apportionment of abscissas and counts, the sets of abscissas and counts are required, and the variable descriptors are optional. When using a histogram point variable, one must define at least one set of abscissa/count pairs. As for Histogram Bin Distribution, the `abscissas` specifications define abscissa values ("x" coordinates) for the PDF of each histogram variable. When paired with `counts`, the specifications provide sets of (x,c) pairs for each histogram variable where `c` defines a count (i.e., a frequency or relative probability) associated with a point.

To fully specify a point-based histogram with `n` points, `n` (x,c) pairs (note that (x,c) and (x,y) are equivalent in this case) must be specified with the following features:

- x is the point value and c is the corresponding count for that value.
- the x values must be strictly increasing.
- all c values must be positive.
- a minimum of one pair must be specified for each point-based histogram.

The num_pairs specification provides for the proper association of multiple sets of (x,c) or (x,y) pairs with individual histogram variables. For example, in the following specification,

```
histogram_point_uncertain = 2
  num_pairs =   2           3
  abscissas = 3 4 100 200 300
  counts    = 1 1   1   2   1
```

num_pairs associates the (x,c) pairs ((3,1),(4,1)) with one point-based histogram variable (where the values 3 and 4 are equally probable) and associates the (x,c) pairs ((100,1),(200,2),(300,1)) with a second point-based histogram variable (where the value 200 is twice as probable as either 100 or 300).

Description	Keyword	Associated Data	Status	Default
histogram point uncertain variables	histogram_-point_-uncertain	integer	Optional group	no histogram point uncertain variables
key to apportionment among point-based histogram variables	num_pairs	list of integers	Optional	equal distribution
sets of abscissas for point-based histogram variables	abscissas	list of reals	Required	N/A
sets of counts for point-based histogram variables	counts	list of reals	Required	N/A
Descriptors	descriptors	list of strings	Optional	vector of 'hupv_i' where i = 1,2,3,...

Table 40.18: Specification detail for histogram point uncertain variables

40.19 Correlations

Aleatory uncertain variables may have correlations specified through use of an uncertain_correlation_-matrix specification. This specification is generalized in the sense that its specific meaning depends on the non-deterministic method in use. When the method is a nondeterministic sampling method (i.e., sampling), then the

correlation matrix specifies *rank correlations* [Iman and Conover, 1982]. When the method is instead a reliability (i.e., `local_reliability` or `global_reliability`) or stochastic expansion (i.e., `polynomial_chaos` or `stoch_collocation`) method, then the correlation matrix specifies *correlation coefficients* (normalized covariance) [Haldar and Mahadevan, 2000]. In either of these cases, specifying the identity matrix results in uncorrelated uncertain variables (the default). The matrix input should be symmetric and have all n^2 entries where *n* is the total number of aleatory uncertain variables (all normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, histogram bin, poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point specifications, in that order). Table 7.24 summarizes the specification details:

Description	Keyword	Associated Data	Status	Default
correlations in aleatory uncertain variables	`uncertain_correlation_matrix`	list of reals	Optional	identity matrix (uncorrelated)

Table 40.19: Specification detail for aleatory uncertain correlations

Chapter 41

Epistemic Uncertain Variables

In addition to continuous and discrete aleatory probability distributions, DAKOTA provides support for epistemic uncertainties through its interval variable specification. This is not a probability distribution; rather, it specifies a set of belief structures based on intervals that may be contiguous, overlapping, or disjoint. It is used in specifying the inputs necessary for an epistemic uncertainty analysis using Dempster-Shafer theory of evidence.

41.1 Interval Uncertain Variable

The interval uncertain variable is NOT a probability distribution. Although it may seem similar to a histogram, the interpretation of this uncertain variable is different. It is used in epistemic uncertainty analysis, where one is trying to model uncertainty due to lack of knowledge. In DAKOTA, epistemic uncertainty analysis is performed using Dempster-Shafer theory of evidence. In this approach, one does not assign a probability distribution to each uncertain input variable. Rather, one divides each uncertain input variable into one or more intervals. The input parameters are only known to occur within intervals: nothing more is assumed. Each interval is defined by its upper and lower bounds, and a Basic Probability Assignment (BPA) associated with that interval. The BPA represents a probability of that uncertain variable being located within that interval. The intervals and BPAs are used to construct uncertainty measures on the outputs called "belief" and "plausibility." Belief represents the smallest possible probability that is consistent with the evidence, while plausibility represents the largest possible probability that is consistent with the evidence. For more information about the Dempster-Shafer approach, see the nondeterministic evidence method, `evidence`, in the Methods section of this Reference manual. As an example, in the following specification:

```
interval_uncertain = 2
  num_intervals   =                        3               2
  interval_probs  =     0.2     0.5     0.3     0.4     0.6
  interval_bounds = 2.0 2.5 4.0 5.0 4.5 6.0 1.0 5.0 3.0 5.0
```

there are 2 interval uncertain variables. The first one is defined by three intervals, and the second by two intervals. The three intervals for the first variable have basic probability assignments of 0.2, 0.5, and 0.3, respectively, while the basic probability assignments for the two intervals for the second variable are 0.4 and 0.6. The basic probability assignments for each interval variable must sum to one. The interval bounds for the first variable are [2, 2.5], [4, 5], and [4.5, 6], and the interval bounds for the second variable are [1.0, 5.0] and [3.0, 5.0].

Note that the intervals can be overlapping or disjoint. Table 7.25 summarizes the specification details for the `interval_uncertain` variable.

Description	Keyword	Associated Data	Status	Default
interval uncertain variables	`interval_-uncertain`	integer	Optional group	no interval uncertain variables
number of intervals defined for each interval variable	`num_-intervals`	list of integers	Required group	None
basic probability assignments per interval	`interval_-probs`	list of reals	Required group. Note that the probabilities per variable must sum to one.	None
bounds per interval	`interval_-bounds`	list of reals	Required group. Specify bounds as (lower, upper) per interval, per variable	None
Descriptors	`descriptors`	list of strings	Optional	vector of `'iuv_i'` where `i = 1,2,3,...`

Table 41.1: Specification detail for interval uncertain variables

Chapter 42

State Variables

State variables provide a convenient mechanism for managing additional model parameterizations such as mesh density, simulation convergence tolerances, and time step controls. Types include continuous real, discrete range of integer values (contiguous integers), discrete set of integer values, and discrete set of real values. Within each optional state variables specification group, the number of variables is always required. The following Tables 7.26 through 7.29 summarize the required and optional specifications for each state variable subtype. The `initial_state` specifications provide the initial values for the state variables which will be passed through to the simulator (e.g., in order to define parameterized modeling controls). The remaining specifications are analagous to those for Design Variables.

42.1 Continuous State Variables

Description	Keyword	Associated Data	Status	Default
Continuous state variables	`continuous_state`	integer	Optional group	No continuous state variables
Initial states	`initial_state`	list of reals	Optional	vector values = `0.` (repaired to bounds, if required)
Lower bounds	`lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Upper bounds	`upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'csv_i'` where `i = 1,2,3,...`

Table 42.1: Specification detail for continuous state variables

Description	Keyword	Associated Data	Status	Default
Discrete state range variables	discrete_-state_range	integer	Optional group	No discrete state variables
Initial states	initial_-state	list of integers	Optional	vector values = 0 (repaired to bounds, if required)
Lower bounds	lower_bounds	list of integers	Optional	vector values = INT_MIN
Upper bounds	upper_bounds	list of integers	Optional	vector values = INT_MAX
Descriptors	descriptors	list of strings	Optional	vector of 'dsriv_i' where i = 1,2,3,...

Table 42.2: Specification detail for discrete state range variables

42.2 Discrete State Range Variables

42.3 Discrete State Integer Set Variables

Description	Keyword	Associated Data	Status	Default
Discrete state set of integer variables	discrete_-state_set_-integer	integer	Optional group	no discrete state set of integer variables
Initial state	initial_-state	list of integers	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	num_set_-values	list of integers	Optional	equal distribution
Set values	set_values	list of integers	Required	N/A
Descriptors	descriptors	list of strings	Optional	vector of 'dssiv_i' where i = 1,2,3,...

Table 42.3: Specification detail for discrete state set of integer variables

42.4 Discrete State Real Set Variables

Description	Keyword	Associated Data	Status	Default
Discrete state set of real variables	`discrete_state_set_real`	integer	Optional group	no discrete state set of real variables
Initial state	`initial_state`	list of reals	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	`num_set_values`	list of integers	Optional	equal distribution
Set values	`set_values`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'dssrv_i'` where `i = 1,2,3,...`

Table 42.4: Specification detail for discrete state set of real variables

Chapter 43

Interface Commands

Chapter 44

Interface Description

The interface section in a DAKOTA input file specifies how function evaluations will be performed in order to map a set of parameters into a set of responses. Function evaluations are performed using either algebraic mappings, interfaces to simulation codes, or a combination of the two.

When employing algebraic mappings, the AMPL solver library [Gay, 1997] is used to evaluate a directed acyclic graph (DAG) specification from a separate `stub.nl` file. Separate `stub.col` and `stub.row` files are also required to declare the string identifiers of the subset of inputs and outputs, respectively, that will be used in the algebraic mappings.

When employing mappings with simulation codes, the simulations may be available internally or externally to DAKOTA. The interface invokes the simulation using either system calls, forks, direct function invocations, or computational grid invocations. In the system call and fork cases, the simulation is external to DAKOTA and communication between DAKOTA and the simulation occurs through parameter and response files. In the direct function case, the simulation is internal to DAKOTA and communication occurs through the function parameter list. The direct case can involve linked simulation codes or test functions which are compiled into the DAKOTA executable. The test functions allow for rapid testing of algorithms without process creation overhead or engineering simulation expense. The grid case is experimental and under development, but is intended to support simulations which are external to DAKOTA and geographically distributed.

Several examples follow. The first example shows a fork interface specification which specifies the names of the analysis executable and the parameters and results files, and that parameters and responses files will be tagged and saved. Refer to Fork interface for more information on the use of these options.

```
interface,
fork
  analysis_drivers = 'rosenbrock'
  parameters_file  = 'params.in'
  results_file     = 'results.out'
  file_tag
  file_save
```

The next example shows a similar specification, except that an external `rosenbrock` executable has been replaced by use of the internal `rosenbrock` test function from the **DirectApplicInterface** class. Refer to Direct function interface for more information on this specification.

```
interface,
direct
  analysis_drivers = 'rosenbrock'
```

The final example demonstrates an interface employing both algebraic and simulation-based mappings. The results from the individual mappings are overlaid based on the variable and response descriptors used by the individual mappings.

```
interface,
algebraic_mappings = 'ampl/fma.nl'
fork
  analysis_driver = 'text_book'
  parameters_file = 'tb.in'
  results_file    = 'tb.out'
  file_tag
asynchronous
```

Chapter 45

Interface Specification

The interface specification has the following top-level structure:

```
interface,
<interface independent controls>
<algebraic mappings specification>
<simulation interface selection>
  <simulation interface dependent controls>
```

The `<interface independent controls>` are those controls which are valid for all interfaces. Referring to dakota.input.summary, these controls are defined externally from the algebraic mappings and simulation interface selection blocks (before and after). Both the algebraic mappings specification and the simulation interface selection are optional specifications, allowing the use of algebraic mappings alone, simulation-based mappings alone, or a combination. The simulation interface selection blocks are all required group specifications separated by logical OR's, where the interface selection must be `system`, `fork`, `direct`, or `grid`. The `<interface dependent controls>` are those controls which are only meaningful for a specific simulation interface selection. These controls are defined within each interface selection block. Defaults for interface independent and simulation interface dependent controls are defined in **DataInterface**. The following sections provide additional detail on the interface independent controls followed by the algebraic mappings specification, the simulation interface selections, and their corresponding simulation interface dependent controls.

Chapter 46

Interface Independent Controls

The optional set identifier specification uses the keyword `id_interface` to input a string for use in identifying a particular interface specification. A model can then identify the use of this interface by specifying the same string in its `interface_pointer` specification (see Model Commands). For example, a model whose specification contains `interface_pointer = 'I1'` will use an interface specification with `id_interface = 'I1'`. If the `id_interface` specification is omitted, a particular interface specification will be used by a model only if that model omits specifying a `interface_pointer` and if the interface set was the last set parsed (or is the only set parsed). In common practice, if only one interface set exists, then `id_interface` can be safely omitted from the interface specification and `interface_pointer` can be omitted from the model specification(s), since there is no potential for ambiguity in this case.

Table 8.1 summarizes the set identifier interface independent control.

Description	Keyword	Associated Data	Status	Default
Interface set identifier	`id_interface`	string	Optional	use of last interface parsed

Table 46.1: Specification detail for interface independent controls: set identifier

Table 8.2 summarizes the interface independent controls associated with parallel computing.

The optional `asynchronous` flag specifies use of asynchronous protocols (i.e., background system calls, non-blocking forks, POSIX threads) when evaluations or analyses are invoked. The `evaluation_concurrency` and `analysis_concurrency` specifications serve a dual purpose:

- when running DAKOTA on a single processor in `asynchronous` mode, the default concurrency of evaluations and analyses is all concurrency that is available. The `evaluation_concurrency` and `analysis_concurrency` specifications can be used to limit this concurrency in order to avoid machine overload or usage policy violation.

- when running DAKOTA on multiple processors in message passing mode, the default concurrency of evaluations and analyses on each of the servers is one (i.e., the parallelism is exclusively that of the message passing). With the `evaluation_concurrency` and `analysis_concurrency` specifications, a hybrid parallelism can be selected through combination of message passing parallelism with asynchronous parallelism on each server.

Description	Keyword	Associated Data	Status	Default
Asynchronous interface usage	asynchronous	none	Optional group	synchronous interface usage
Asynchronous evaluation concurrency	evaluation_-concurrency	integer	Optional	local: unlimited concurrency, hybrid: no concurrency
Self-schedule local evals	local_-evaluation_-self_-scheduling	none	Optional (1 of 2)	self-scheduled local evals
Static-schedule local evals	local_-evaluation_-static_-scheduling	none	Optional (1 of 2)	self-scheduled local evals
Asynchronous analysis concurrency	analysis_-concurrency	integer	Optional	local: unlimited concurrency, hybrid: no concurrency
Number of evaluation servers	evaluation_-servers	integer	Optional	no override of auto configure
Self scheduling of evaluations	evaluation_-self_-scheduling	none	Optional	no override of auto configure
Static scheduling of evaluations	evaluation_-static_-scheduling	none	Optional	no override of auto configure
Number of analysis servers	analysis_-servers	integer	Optional	no override of auto configure
Self scheduling of analyses	analysis_-self_-scheduling	none	Optional	no override of auto configure
Static scheduling of analyses	analysis_-static_-scheduling	none	Optional	no override of auto configure

Table 46.2: Specification detail for interface independent controls: parallelism

The optional `evaluation_servers` and `analysis_servers` specifications support user overrides of the automatic parallel configuration for the number of evaluation servers and the number of analysis servers. Similarly, the optional `evaluation_self_scheduling`, `evaluation_static_scheduling`, `analysis_self_scheduling`, and `analysis_static_scheduling` specifications can be used to override the automatic parallel configuration of scheduling approach at the evaluation and analysis parallelism levels. That is, if the automatic configuration is undesirable for some reason, the user can enforce a desired number of partitions and a desired scheduling policy at these parallelism levels. Refer to **ParallelLibrary** and the Parallel Computing chapter of the Users Manual [Adams et al., 2010] for additional information.

When performing asynchronous local evaluations, the local evaluation scheduling keywords control how new evaluation jobs are dispatched when one completes. If `local_evaluation_self_scheduling` (default) is specified, each completed evaluation will be replaced by the next in the local evaluation queue. If `local_evaluation_static_scheduling` is specified, each completed evaluation will be replaced by an evaluation number congruent modulo the `evaluation_concurrency`. This is helpful for relative node scheduling as described in `Dakota/examples/parallelism`. For example, assuming only local concurrency (no MPI), if the local concurrency is 7 and job 2 completes, it will be replaced with job 9. This can result in idle processors if runtimes are non-uniform.

Chapter 47

Algebraic mappings

If desired, one can define algebraic input-output mappings using the AMPL code [Fourer et al., 2003] and save these mappings in 3 files: `stub.nl`, `stub.col`, and `stub.row`, where `stub` is a particular root name describing a particular problem. These files names can be communicated to DAKOTA using the `algebraic_mappings` input. This string may either specify the `stub.nl` filename, or alternatively, just the `stub` itself.

DAKOTA then uses `stub.col` and `stub.row` to extract the input and output identifier strings and employs the AMPL solver library [Gay, 1997] to process the DAG specification in `stub.nl`. The variable and objective function names declared within AMPL should be a subset of the variable descriptors and response descriptors used by DAKOTA (see Variables Commands and Response Labels). Ordering is not important, as DAKOTA will reorder data as needed.

Table 8.3 summarizes the algebraic mappings specification.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Algebraic mappings file	`algebraic_mappings`	string	Optional	no algebraic mappings

Table 47.1: Specification detail for algebraic mappings

Chapter 48

Simulation interfaces

Each simulation interface uses one or more simulator programs, and optionally filter programs, to perform the parameter to response mapping. The simulator and filter programs are invoked with system calls, forks, direct function calls, or computational grid invocations. In the system call and fork cases, a separate process is created for the simulator program and files are used for transfer of parameter and response data between DAKOTA and the simulator program. This approach is simple and reliable and does not require any modification to simulator programs. In the direct function case, subroutine parameter lists are used to pass the parameter and response data. This approach requires modification to simulator programs so that they can be linked into DAKOTA; however it can be more efficient through the elimination of process creation overhead and deactivation of unnecessary simulator functions (e.g., output), can be less prone to loss of precision in that data can be passed directly rather than written to and read from a file, and can enable completely internal management of multiple levels of parallelism through the use of MPI communicator partitioning. In the grid case, computational grid services are utilized in order to enable distribution of simulations across different computer resources. This capability targets Condor and/or Globus services but is currently experimental and incomplete.

Table 8.4 summarizes the interface independent controls associated with the simulator programs.

The required `analysis_drivers` specification provides the names of executable analysis programs or scripts which comprise a function evaluation. The specification can also give values to environment variables that the programs will see; for details, see the subsection on Syntax for Filter and Driver Strings in the Interfaces chapter of the Users Manual [Adams et al., 2010]. The common case of a single analysis driver is simply accommodated by specifying a list of one driver (this also provides backward compatibility with previous DAKOTA versions). The optional `analysis_components` specification allows the user to provide additional identifiers (e.g., mesh file names) for use by the analysis drivers. This is particularly useful when the same analysis driver is to be reused multiple times for slightly different analyses. The specific content within the strings is open-ended and can involve whatever syntax is convenient for a particular analysis driver. The number of analysis components n_c should be an integer multiple of the number of drivers n_d, and the first n_c/n_d component strings will be passed to the first driver, etc. The optional `input_filter` and `output_filter` specifications provide the names of separate pre- and post-processing programs or scripts which assist in mapping DAKOTA parameters files into analysis input files and mapping analysis output files into DAKOTA results files, respectively. If there is only a single analysis driver, then it is usually most convenient to combine pre- and post-processing requirements into a single analysis driver script and omit the separate input and output filters. However, in the case of multiple analysis drivers, the input and output filters provide a convenient location for non-repeated pre- and post-processing requirements. That is, input and output filters are only executed once per function evaluation, regardless of the number of analysis drivers, which makes them convenient locations for data processing operations that are shared among the analysis

Description	Keyword	Associated Data	Status	Default
Analysis drivers	analysis_-drivers	list of strings	Required	N/A
Additional identifiers for use by the analysis_-drivers	analysis_-components	list of strings	Optional	no additional identifiers
Input filter	input_filter	string	Optional	no input filter
Output filter	output_-filter	string	Optional	no output filter
Failure capturing	failure_-capture	abort | retry (with integer data) | recover (with list of reals data) | continuation	Optional group	abort
Feature deactivation	deactivate	active_set_-vector, evaluation_-cache, and/or restart_file	Optional group	Active set vector control, function evaluation cache, and restart file features are active

Table 48.1: Specification detail for simulation interface controls: drivers, filters, failure capturing, and feature management

drivers.

Failure capturing in interfaces is governed by the optional failure_capture specification. Supported directives for mitigating captured failures are abort (the default), retry, recover, and continuation. The retry selection supports an integer input for specifying a limit on retries, and the recover selection supports a list of reals for specifying the dummy function values (only zeroth order information is supported) to use for the failed function evaluation. Refer to the Simulation Code Failure Capturing chapter of the Users Manual [Adams et al., 2010] for additional information.

The optional deactivate specification block includes three features which a user may deactivate in order to simplify interface development, increase execution speed, and/or reduce memory and disk requirements:

- Active set vector (ASV) control: deactivation of this feature using a deactivate active_set_-vector specification allows the user to turn off any variability in ASV values so that active set logic can be omitted in the user's simulation interface. This option trades some efficiency for simplicity in interface development. The default behavior is to request the minimum amount of data required by an algorithm at any given time, which implies that the ASV values may vary from one function evaluation to the next. Since the user's interface must return the data set requested by the ASV values, this interface must contain additional logic to account for any variations in ASV content. Deactivating this ASV control causes DAKOTA to always request a "full" data set (the full function, gradient, and Hessian data that is available from the interface as specified in the responses specification) on each function evaluation. For example, if ASV control has been deactivated and the responses section specifies four response functions, analytic gradients, and no Hessians, then the ASV on every function evaluation will be { 3 3 3 3 }, regardless of what subset of this data is currently needed. While wasteful of computations in many instances, this simplifies the interface and allows the user to return the same data set on every evaluation. Conversely, if ASV control

is active (the default behavior), then the ASV requests in this example might vary from { 1 1 1 1 } to { 2 0 0 2 }, etc., according to the specific data needed on a particular function evaluation. This will require the user's interface to read the ASV requests and perform the appropriate logic in conditionally returning only the data requested. In general, the default ASV behavior is recommended for the sake of computational efficiency, unless interface development time is a critical concern. Note that in both cases, the data returned to DAKOTA from the user's interface must match the ASV passed in, or else a response recovery error will result. However, when the ASV control is deactivated, the ASV values are invariant and need not be checked on every evaluation. *Note:* Deactivating the ASV control can have a positive effect on load balancing for parallel DAKOTA executions. Thus, there is significant overlap in this ASV control option with speculative gradients (see Method Independent Controls). There is also overlap with the mode override approach used with certain optimizers (see **SNLLOptimizer** and **SNLLLeastSq**) to combine individual value, gradient, and Hessian requests.

- Function evaluation cache: deactivation of this feature using a `deactivate evaluation_cache` specification allows the user to avoid retention of the complete function evaluation history in memory. This can be important for reducing memory requirements in large-scale applications (i.e., applications with a large number of variables or response functions) and for eliminating the overhead of searching for duplicates within the function evaluation cache prior to each new function evaluation (e.g., for improving speed in problems with 1000's of inexpensive function evaluations or for eliminating overhead when performing timing studies). However, the downside is that unnecessary computations may be performed since duplication in function evaluation requests may not be detected. For this reason, this option is not recommended when function evaluations are costly. *Note:* duplication detection within DAKOTA can be deactivated, but duplication detection features within specific optimizers may still be active.

- Restart file: deactivation of this feature using a `deactivate restart_file` specification allows the user to eliminate the output of each new function evaluation to the binary restart file. This can increase speed and reduce disk storage requirements, but at the expense of a loss in the ability to recover and continue a run that terminates prematurely (e.g., due to a system crash or network problem). This option is not recommended when function evaluations are costly or prone to failure. Please note that using the `deactivate restart_file` specification will result in a zero length restart file with the default name `dakota.rst`.

In addition to these simulation interface specifications, the type of interface involves a selection between `system`, `fork`, `direct`, or `grid` required group specifications. The following sections describe these group specifications in detail.

48.1 System call interface

Users are strongly encouraged to use the fork interface if possible, though the system interface is still supported for portability and backward compatibility. For system call interfaces, the `system` keyword anchors the group specification and the `parameters_file`, `results_file`, `verbatim`, `aprepro`, `file_tag`, and `file_save` are additional settings within the group specification. The parameters and results file names are supplied as strings using the `parameters_file` and `results_file` specifications. Both specifications are optional with the default data transfer files being Unix temporary files with system-generated names (e.g., `/usr/tmp/aaaa08861`). The parameters and results file names are passed on the command line to the analysis driver(s) and any specified input/output filters, unless the `verbatim` option is invoked, in which case the driver/filter invocation syntax is used verbatim without command line argument augmentation. For additional information on invocation syntax, see the Interfaces chapter of the Users Manual [Adams et al., 2010]. The

format of data in the parameters files can be modified for direct usage with the APREPRO pre-processing tool [Sjaardema, 1992] using the `aprepro` specification (NOTE: the DPrePro pre-processing utility does not require this special formatting). File tagging (appending parameters and results files with the function evaluation number) and file saving (leaving parameters and results files in existence after their use is complete) are controlled with the `file_tag` and `file_save` flags. If these specifications are omitted, the default is no file tagging (no appended function evaluation number) and no file saving (files will be removed after a function evaluation). File tagging is most useful when multiple function evaluations are running simultaneously using files in a shared disk space, and file saving is most useful when debugging the data communication between DAKOTA and the simulation.

By default DAKOTA will remove existing results files before invoking the `analysis_driver` to avoid problems created by stale files in the current directory. To override this behavior and not delete existing files, specify `allow_existing_files`.

When performing concurrent evaluations and/or analyses, it is often necessary to cloister input and output files in separate directories to avoid conflicts. When the `work_directory` feature is enabled, DAKOTA will create a directory for each evaluation/analysis (with optional tagging and saving as with files), and execute the analysis driver from that working directory. If an optional `template_directory` and/or list of `template_files` is specified, DAKOTA will link (or copy) those files into each working directory. The specifications for system call interfaces are summarized in Table 8.5.

48.2 Fork interface

For fork interfaces, the `fork` keyword anchors a group specification. The `parameters_file`, `results_file`, `verbatim`, `aprepro`, `file_tag`, and `file_save` are additional settings within this group specification and have identical meanings to those for the system call interface as summarized previously in Table 8.5.

48.3 Direct function interface

For direct function interfaces, `processors_per_analysis` is an additional optional setting within the required group which can be used to specify multiprocessor analysis partitions. As with the `evaluation_servers`, `analysis_servers`, `evaluation_self_scheduling`, `evaluation_static_scheduling`, `analysis_self_scheduling`, and `analysis_static_scheduling` specifications described above in Interface Independent Controls, `processors_per_analysis` provides a means for the user to override the automatic parallel configuration (refer to **ParallelLibrary** and the Parallel Computing chapter of the Users Manual [Adams et al., 2010]) for the number of processors used for each analysis partition. Note that if both `analysis_servers` and `processors_per_analysis` are specified and they are not in agreement, then `analysis_servers` takes precedence. The direct interface specifications are summarized in Table 8.6.

DAKOTA supports direct interfaces to a few select simulation codes. One example is ModelCenter, a commercial simulation management framework from Phoenix Integration. To utilize this interface, a user must first define the simulation specifics within a ModelCenter session and then save these definitions to a ModelCenter configuration file. The `analysis_components` specification provides the means to communicate this configuration file to DAKOTA's ModelCenter interface. A similar direct interface to The Mathworks' (`http://www.mathworks.com/`) Matlab (specified by `analysis_driver = 'matlab'`) enables a user to employ the `analysis_components` specification to point to a Matlab m-file containing a function that performs the simulation. This capability is disabled by default in DAKOTA binaries, but information on enabling

Description	Keyword	Associated Data	Status	Default
System call interface	system	none	Required group (1 of 4 selections)	N/A
Parameters file name	parameters_-file	string	Optional	Unix temp files
Results file name	results_file	string	Optional	Unix temp files
Allow existing results files	allow_-existing_-results	none	Optional	results files removed before each evaluation
Verbatim driver/filter invocation syntax	verbatim	none	Optional	driver/filter invocation syntax augmented with file names
Aprepro parameters file format	aprepro	none	Optional	standard parameters file format
Parameters and results file tagging	file_tag	none	Optional	no tagging
Parameters and results file saving	file_save	none	Optional	file cleanup
Create work directory	work_-directory	none	Optional	no work directory
Name of work directory	named	string	Optional	workdir
Tag work directory	directory_-tag	none	Optional	no work directory tagging
Save work directory	directory_-save	none	Optional	remove work directory
Template directory	template_-directory	string path	Optional	no template directory
Template files	template_-files	list of strings	Optional	no template files
Copy template files	copy	none	Optional	link template files
Replace existing files	replace	none	Optional	do not overwrite files

Table 48.2: Additional specifications for system call interfaces

Description	Keyword	Associated Data	Status	Default
Direct function interface	direct	none	Required group (1 of 4 selections)	N/A
Number of processors per analysis	processors_-per_analysis	integer	Optional	no override of auto configure

Table 48.3: Additional specifications for direct function interfaces

and using it is available in the Users Manual [Adams et al., 2010]. Contact the DAKOTA users mailing list for assistance building and using DAKOTA with Matlab simulation support. An experimental interface to Python simulation codes is also available and can be enabled when compiling DAKOTA from source.

Other direct interfaces to simulation codes include Sandia's SALINAS structural dynamics code, Sandia's SIERRA multiphysics framework, and Sandia's SAGE computational fluid dynamics code, which are available within Sandia and supported to varying degrees. In addition to interfaces with simulation codes, a common usage of the direct interface is for invoking internal test functions which are available for performing parameter to response mappings as inexpensively as possible. These problems are compiled directly into the DAKOTA executable as part of the direct function interface class and are used for algorithm testing. Refer to **DirectApplicInterface** for currently available testers.

48.4 Grid interface

For grid interfaces, no additional specifications are used at this time.

This capability has been used for interfaces with IDEA and JAVASpaces in the past and is currently a placeholder for future work with Condor and/or Globus services. It is not currently operational. The grid interface specification is summarized in Table 8.7.

Description	Keyword	Associated Data	Status	Default
Grid interface	`grid`	none	Required group (1 of 4 selections)	N/A

Table 48.4: Additional specifications for grid interfaces

Chapter 49

Responses Commands

Chapter 50

Responses Description

Responses specify the data set produced by an interface after the completion of a "function evaluation." Here, the term function evaluation is used loosely to denote a data request from an iterator that is mapped through an interface in a single pass. Strictly speaking, this data request may actually involve multiple response functions and their derivatives, but the term function evaluation is widely used for this purpose. The data set is potentially comprised of a set of functions, their first derivative vectors (gradients), and their second derivative matrices (Hessians). This abstraction provides a generic data container (the **Response** class) whose contents are interpreted differently depending upon the type of iteration being performed. In the case of optimization, the set of functions consists of one or more objective functions, nonlinear inequality constraints, and nonlinear equality constraints. (Linear constraints are not part of a response set since their coefficients can be communicated to an optimizer at start up and then computed internally for all function evaluations; see Method Independent Controls). In the case of least squares iterators, the functions consist of individual residual terms or model responses together with an observed data file for comparison (as opposed to a sum of the squares objective function) as well as nonlinear inequality and equality constraints. In the case of nondeterministic iterators, the function set is made up of generic response functions for which the effect of parameter uncertainty is to be quantified. Parameter study and design of experiments iterators may be used with any of the response data set types. Thus the interpretation of the response data varies from iterator to iterator.

Gradient specification types include none, numerical, analytic, and mixed. The `no_gradients` selection indicates that gradient information is not needed in the study. The `numerical_gradients` selection means that gradient information is needed and will be computed with finite differences by DAKOTA or the optimization algorithm in use. The `analytic_gradients` selection means that gradient information is available directly from the simulation (finite differencing is not required). And the `mixed_gradients` selection means that some gradient information is available directly from the simulation whereas the rest will have to be estimated with finite differences.

Hessian availability is characterized as none, analytic, numerical, quasi, or mixed. Similar to gradients, the `no_hessians` selection indicates that Hessian information is not needed/available in the study, and the `analytic_hessians` selection indicates that Hessian information is available directly from the simulation. The `numerical_hessians` selection indicates that Hessian information will be estimated with finite differences. The `quasi_hessians` specification means that Hessian information will be accumulated over time using secant updates based on the existing gradient evaluations. Finally, the `mixed_hessians` selection allows for a mixture of analytic, numerical, and quasi Hessian response data.

Responses specify the *total* data set that is available for use by the method over the course of iteration. This is distinguished from the data *subset* described by an active set vector (see DAKOTA File Data Formats in the Users

Manual [Adams et al., 2010]) indicating the particular subset of the response data needed for a particular function evaluation. Thus, the responses specification is a broad description of the data to be used during a study whereas the active set vector indicates the subset currently needed.

Several examples follow. The first example shows an optimization data set containing an objective function and two nonlinear inequality constraints. These three functions have analytic gradient availability and no Hessian availability.

```
responses,
objective_functions = 1
nonlinear_inequality_constraints = 2
analytic_gradients
no_hessians
```

The next example shows a typical specification for a calibration data set. The six residual functions will have numerical gradients computed using the dakota finite differencing routine with central differences of 0.1% (plus/minus delta value = .001∗value).

```
responses,
calibration_terms = 6
numerical_gradients
  method_source dakota
  interval_type central
  fd_gradient_step_size = .001
no_hessians
```

The last example shows a specification that could be used with a nondeterministic sampling iterator. The three response functions have no gradient or Hessian availability; therefore, only function values will be used by the iterator.

```
responses,
response_functions = 3
no_gradients
no_hessians
```

Parameter study and design of experiments iterators are not restricted in terms of the response data sets which may be catalogued; they may be used with any of the function specification examples shown above.

Chapter 51

Responses Specification

The responses specification has the following structure (see dakota.input.summary):

```
responses,
<set identifier>
<response descriptors>
<function specification>
<gradient specification>
<Hessian specification>
```

The set identifier and response descriptors are optional. However, the function, gradient, and Hessian specifications are all required, their type selected from the options discussed above. For example, the function specification must be one of three types:

- objective and constraint functions
- calibration (least squares) terms and constraint functions
- generic response functions

The following sections describe each of these specification components and their options in additional detail.

Chapter 52

Responses Set Identifier

The optional set identifier specification uses the keyword `id_responses` to input a string for use in identifying a particular responses specification. A model can then identify the use of this response set by specifying the same string in its `responses_pointer` specification (see Model Independent Controls). For example, a model whose specification contains `responses_pointer = 'R1'` will use a responses set with `id_responses = 'R1'`.

If the `id_responses` specification is omitted, a particular responses specification will be used by a model only if that model omits specifying a `responses_pointer` and if the responses set was the last set parsed (or is the only set parsed). In common practice, if only one responses set exists, then `id_responses` can be safely omitted from the responses specification and `responses_pointer` can be omitted from the model specification(s), since there is no potential for ambiguity in this case. Table 9.1 summarizes the set identifier input.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Responses set identifier	`id_responses`	string	Optional	use of last responses parsed

Table 52.1: Specification detail for set identifier

Chapter 53

Response Labels

The optional response labels specification `response_descriptors` is a list of strings which will be printed in DAKOTA output to identify the values for particular response functions. The default descriptor strings use a root string plus a numeric identifier. This root string is `"obj_fn"` for objective functions, `"least_sq_term"` for least squares terms, `"response_fn"` for generic response functions, `"nln_ineq_con"` for nonlinear inequality constraints, and `"nln_eq_con"` for nonlinear equality constraints. Table 9.2 summarizes the response descriptors input.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Response labels	`descriptors`	list of strings	Optional	root strings plus numeric identifiers

Table 53.1: Specification detail for response labels

Chapter 54

Function Specification

The function specification must be one of three types: 1) a group containing objective and constraint functions, 2) a group containing calibration (least squares) terms and constraint functions, or 3) a generic response functions specification. These function sets correspond to optimization, least squares, and uncertainty quantification iterators, respectively. Parameter study and design of experiments iterators may be used with any of the three function specifications.

54.1 Objective and constraint functions (optimization data set)

An optimization data set is specified using `objective_functions` and optionally `objective_function_scale_types`, `objective_function_scales`, `multi_objective_weights`, `nonlinear_inequality_constraints`, `nonlinear_inequality_lower_bounds`, `nonlinear_inequality_upper_bounds`, `nonlinear_inequality_scale_types`, `nonlinear_inequality_scales`, `nonlinear_equality_constraints`, `nonlinear_equality_targets`, `nonlinear_equality_scale_types`, and `nonlinear_equality_scales`. The `objective_functions`, `nonlinear_inequality_constraints`, and `nonlinear_equality_constraints` inputs specify the number of objective functions, nonlinear inequality constraints, and nonlinear equality constraints, respectively. The number of objective functions must be 1 or greater, and the number of inequality and equality constraints must be 0 or greater. The `objective_function_scale_types` specification includes strings specifying the scaling type for each objective function value in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `objective_function_scale_types` may be selected from `'none'`, `'value'`, or `'log'`, to select no, characteristic value, or logarithmic scaling, respectively. Automatic scaling is not available for objective functions. If a single string is specified it will apply to each objective function. Each entry in `objective_function_scales` may be a user-specified nonzero characteristic value to be used in scaling each objective function. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'log'`. If a single real value is specified it will apply to each function. If the number of objective functions is greater than 1, then a `multi_objective_weights` specification provides a simple weighted-sum approach to combining multiple objectives:

$$f = \sum_{i=1}^{n} w_i f_i$$

If this is not specified, then each objective function is given equal weighting:

$$f = \sum_{i=1}^{n} \frac{f_i}{n}$$

If scaling is specified, it is applied before multi-objective weighted sums are formed.

The `nonlinear_inequality_lower_bounds` and `nonlinear_inequality_upper_bounds` specifications provide the lower and upper bounds for 2-sided nonlinear inequalities of the form

$$g_l \leq g(x) \leq g_u$$

The defaults for the inequality constraint bounds are selected so that one-sided inequalities of the form

$$g(x) \leq 0.0$$

result when there are no user constraint bounds specifications (this provides backwards compatibility with previous DAKOTA versions). In a user bounds specification, any upper bound values greater than `+bigRealBoundSize` (1.e+30, as defined in **Minimizer**) are treated as +infinity and any lower bound values less than `-bigRealBoundSize` are treated as -infinity. This feature is commonly used to drop one of the bounds in order to specify a 1-sided constraint (just as the default lower bounds drop out since `-DBL_MAX` < `-bigRealBoundSize`). The same approach is used for nonexistent linear inequality bounds as described in Method Independent Controls and for nonexistent design variable bounds as described in Design Variables.

The `nonlinear_equality_targets` specification provides the targets for nonlinear equalities of the form

$$g(x) = g_t$$

and the defaults for the equality targets enforce a value of `0.` for each constraint

$$g(x) = 0.0$$

The `nonlinear_inequality_scale_types` and `nonlinear_equality_scale_types` specifications include strings specifying the scaling type for each nonlinear inequality or equality constraint, respectively, in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `objective_function_scale_types` may be selected from `'none'`, `'value'`, `'auto'`, or `'log'`, to select no, characteristic value, automatic, or logarithmic scaling, respectively. If a single string is specified it will apply to all components of the relevant nonlinear constraint vector. Each entry in `nonlinear_inequality_scales` and `nonlinear_equality_scales` may be a user-specified nonzero characteristic value to be used in scaling each constraint component. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'` and `'log'`. If a single real value is specified it will apply to each constraint.

Any linear constraints present in an application need only be input to an optimizer at start up and do not need to be part of the data returned on every function evaluation (see the linear constraints description in Method Independent Controls). Table 9.3 summarizes the optimization data set specification.

54.2 Calibration terms and constraint functions (least squares data set)

A calibration data set is specified using `calibration_terms` and optionally the specifications summarized in Table 9.4 and Table 9.5, including weighting/scaling, data, and constraints. Each of the calibration terms is a

Description	Keyword	Associated Data	Status	Default
Number of objective functions	objective_-functions	integer	Required group	N/A
Objective function scaling types	objective_-function_-scale_types	list of strings	Optional	vector values = 'none'
Objective function scales	objective_-function_-scales	list of reals	Optional	vector values = 1. (no scaling)
Multiobjective weightings	multi_-objective_-weights	list of reals	Optional	equal weightings
Number of nonlinear inequality constraints	nonlinear_-inequality_-constraints	integer	Optional	0
Nonlinear inequality constraint lower bounds	nonlinear_-inequality_-lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Nonlinear inequality constraint upper bounds	nonlinear_-inequality_-upper_bounds	list of reals	Optional	vector values = 0.
Nonlinear inequality constraint scaling types	nonlinear_-inequality_-scale_types	list of strings	Optional	vector values = 'none'
Nonlinear inequality constraint scales	nonlinear_-inequality_-scales	list of reals	Optional	vector values = 1. (no scaling)
Number of nonlinear equality constraints	nonlinear_-equality_-constraints	integer	Optional	0
Nonlinear equality constraint targets	nonlinear_-equality_-targets	list of reals	Optional	vector values = 0.
Nonlinear equality constraint scaling types	nonlinear_-equality_-scale_types	list of strings	Optional	vector values = 'none'
Nonlinear equality constraint scales	nonlinear_-equality_-scales	list of reals	Optional	vector values = 1. (no scaling)

Table 54.1: Specification detail for optimization data sets

residual function to be driven toward zero, and the nonlinear inequality and equality constraint specifications have identical meanings to those described in Objective and constraint functions (optimization data set). These types of problems are commonly encountered in parameter estimation, system identification, and model calibration. Least squares calibration problems are most efficiently solved using special-purpose least squares solvers such as Gauss-Newton or Levenberg-Marquardt; however, they may also be solved using general-purpose optimization algorithms.

While DAKOTA can solve these problems with either least squares or optimization algorithms, the response data sets to be returned from the simulator are different. Least squares calibration involves a set of residual functions whereas optimization involves a single objective function (sum of the squares of the residuals), i.e.,

$$f = \sum_{i=1}^{n} R_i^2$$

where f is the objective function and the set of R_i are the residual functions. Therefore, function values and derivative data in the least squares case involve the values and derivatives of the residual functions, whereas the optimization case involves values and derivatives of the sum of squares objective function. This means that in the least squares calibration case, the user must return each of `n` residuals separately as a separate calibration term. Switching between the two approaches sometimes requires different simulation interfaces capable of returning the different granularity of response data required, although DAKOTA supports automatic recasting of residuals into a sum of squares for presentation to an optimization method. Typically, the user must compute the difference between the model results and the observations when computing the residuals. However, the user has the option of specifying the observational data (e.g. from physical experiments or other sources) in a file. The specification `calibration_data_file` may be used to specify a text file containing `calibration_terms` observed data values (in a supported DAKOTA tabular format; default formats change in DAKOTA .5.2 -- see User's Manual) to be used in computing the residuals

$$R_i = y_i^M - y_i^O$$

where *M* denotes model and *O*, observation from the file. In this case the simulator should return the actual model response, as DAKOTA will compute the residual internally using the supplied data.

The `calibration_term_scale_types` specification includes strings specifying the scaling type for each residual term in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `calibration_term_scale_types` may be selected from `'none'`, `'value'`, or `'log'`, to select no, characteristic value, or logarithmic scaling, respectively. Automatic scaling is not available for calibration terms. If a single string is specified it will apply to each least squares terms. Each entry in `calibration_term_scales` may be a user-specified nonzero characteristic value to be used in scaling each term. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'log'`. If a single real value is specified it will apply to each term. The `calibration_weights` specification provides a means to specify a relative emphasis among the vector of squared residuals through multiplication of these squared residuals by a vector of weights:

$$f = \sum_{i=1}^{n} w_i R_i^2 = \sum_{i=1}^{n} w_i (y_i^M - y_i^O)^2$$

If characteristic value scaling is additionally specified, then it is applied to each residual prior to squaring:

$$f = \sum_{i=1}^{n} w_i (\frac{y_i^M - y_i^O}{s_i})^2$$

And in the case where experimental data uncertainties are supplied, then the weights are automatically defined to be the inverse of the experimental variance:

$$f = \sum_{i=1}^{n} \frac{1}{\sigma_i^2} (\frac{y_i^M - y_i^O}{s_i})^2$$

Description	Keyword	Associated Data	Status	Default
Number of calibration terms	`calibration_terms`	integer	Required	N/A
Calibration data file name	`calibration_data_file`	string	Optional	none
Experiments (rows) in file	`num_experiments`	integer	Optional	1
Data file in annotated format	`annotated`	boolean	Optional	annotated
Data file in freeform format	`freeform`	boolean	Optional	annotated
Configuration variable columns in file	`num_config_variables`	integer	Optional	0
Standard deviation columns in file	`num_std_deviations`	integer	Optional	0
Calibration scaling types	`calibration_term_scale_types`	list of strings	Optional	vector values = `'none'`
Calibration scales	`calibration_term_scales`	list of reals	Optional	no scaling (vector values = `1`.)
Calibration term weights	`calibration_weights`	list of reals	Optional	equal weighting

Table 54.2: Specification detail for nonlinear least squares data sets (calibration terms)

54.3 Response functions (generic data set)

A generic response data set is specified using `response_functions`. Each of these functions is simply a response quantity of interest with no special interpretation taken by the method in use. This type of data set is used by uncertainty quantification methods, in which the effect of parameter uncertainty on response functions is quantified, and can also be used in parameter study and design of experiments methods (although these methods are not restricted to this data set), in which the effect of parameter variations on response functions is evaluated. Whereas objective, constraint, and residual functions have special meanings for optimization and least squares algorithms, the generic response function data set need not have a specific interpretation and the user is free to define whatever functional form is convenient. Table 9.6 summarizes the generic response function data set specification.

Description	Keyword	Associated Data	Status	Default
Number of nonlinear inequality constraints	nonlinear_-inequality_-constraints	integer	Optional	0
Nonlinear inequality lower bounds	nonlinear_-inequality_-lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Nonlinear inequality upper bounds	nonlinear_-inequality_-upper_bounds	list of reals	Optional	vector values = 0.
Nonlinear inequality scaling types	nonlinear_-inequality_-scale_types	list of strings	Optional	vector values = 'none'
Nonlinear inequality scales	nonlinear_-inequality_-scales	list of reals	Optional	no scaling (vector values = 1.)
Number of nonlinear equality constraints	nonlinear_-equality_-constraints	integer	Optional	0
Nonlinear equality targets	nonlinear_-equality_-targets	list of reals	Optional	vector values = 0.
Nonlinear equality scaling types	nonlinear_-equality_-scale_types	list of strings	Optional	vector values = 'none'
Nonlinear equality scales	nonlinear_-equality_-scales	list of reals	Optional	no scaling (vector values = 1.)

Table 54.3: Specification detail for nonlinear least squares data sets (constraints)

Description	Keyword	Associated Data	Status	Default
Number of response functions	response_-functions	integer	Required	N/A

Table 54.4: Specification detail for generic response function data sets

Chapter 55

Gradient Specification

The gradient specification must be one of four types: 1) no gradients, 2) numerical gradients, 3) analytic gradients, or 4) mixed gradients.

55.1 No gradients

The `no_gradients` specification means that gradient information is not needed in the study. Therefore, it will neither be retrieved from the simulation nor computed with finite differences. The `no_gradients` keyword is a complete specification for this case.

55.2 Numerical gradients

The `numerical_gradients` specification means that gradient information is needed and will be computed with finite differences using either the native or one of the vendor finite differencing routines.

The `method_source` setting specifies the source of the finite differencing routine that will be used to compute the numerical gradients: `dakota` denotes DAKOTA's internal finite differencing algorithm and `vendor` denotes the finite differencing algorithm supplied by the iterator package in use (DOT, CONMIN, NPSOL, NL2SOL, NLSSOL, and OPT++ each have their own internal finite differencing routines). The `dakota` routine is the default since it can execute in parallel and exploit the concurrency in finite difference evaluations (see Exploiting Parallelism in the Users Manual [Adams et al., 2010]). However, the `vendor` setting can be desirable in some cases since certain libraries will modify their algorithm when the finite differencing is performed internally. Since the selection of the `dakota` routine hides the use of finite differencing from the optimizers (the optimizers are configured to accept user-supplied gradients, which some algorithms assume to be of analytic accuracy), the potential exists for the `vendor` setting to trigger the use of an algorithm more optimized for the higher expense and/or lower accuracy of finite-differencing. For example, NPSOL uses gradients in its line search when in user-supplied gradient mode (since it assumes they are inexpensive), but uses a value-based line search procedure when internally finite differencing. The use of a value-based line search will often reduce total expense in serial operations. However, in parallel operations, the use of gradients in the NPSOL line search (user-supplied gradient mode) provides excellent load balancing without need to resort to speculative optimization approaches. In summary, then, the `dakota` routine is preferred for parallel optimization, and the `vendor` routine may be preferred for serial optimization in special cases.

The `interval_type` setting is used to select between `forward` and `central` differences in the numerical gradient calculations. The `dakota`, DOT `vendor`, and OPT++ `vendor` routines have both forward and central differences available, the CONMIN and NL2SOL `vendor` routines support forward differences only, and the NPSOL and NLSSOL `vendor` routines start with forward differences and automatically switch to central differences as the iteration progresses (the user has no control over this). The following forward difference expression

$$\nabla f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

and the following central difference expression

$$\nabla f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}$$

are used to estimate the ith component of the gradient vector.

Lastly, `fd_gradient_step_size` specifies the relative finite difference step size to be used in the computations. Either a single value may be entered for use with all parameters, or a list of step sizes may be entered, one for each parameter. The latter option of a list of step sizes is only valid for use with the DAKOTA finite differencing routine. For DAKOTA, DOT, CONMIN, and OPT++, the differencing intervals are computed by multiplying the `fd_gradient_step_size` with the current parameter value. In this case, a minimum absolute differencing interval is needed when the current parameter value is close to zero. This prevents finite difference intervals for the parameter which are too small to distinguish differences in the response quantities being computed. DAKOTA, DOT, CONMIN, and OPT++ all use `.01*fd_gradient_step_size` as their minimum absolute differencing interval. With a `fd_gradient_step_size = .001`, for example, DAKOTA, DOT, CONMIN, and OPT++ will use intervals of .001∗current value with a minimum interval of 1.e-5. NPSOL and NLSSOL use a different formula for their finite difference intervals: `fd_gradient_step_size*(1+|current parameter value|)`. This definition has the advantage of eliminating the need for a minimum absolute differencing interval since the interval no longer goes to zero as the current parameter value goes to zero.

When DAKOTA computes gradients or Hessians by finite differences and the variables in question have bounds, it by default chooses finite-differencing steps that keep the variables within their specified bounds. Older versions of DAKOTA generally ignored bounds when computing finite differences. To restore the older behavior, one can add keyword `ignore_bounds` to the `response` specification when `method_source dakota` (or just `dakota`) is also specified. In forward difference or backward difference computations, honoring bounds is straightforward. To honor bounds when approximating $\partial f/\partial x_i$, i.e., component i of the gradient of f, by central differences, DAKOTA chooses two steps h_1 and h_2 with $h_1 \neq h_2$, such that $x + h_1 e_i$ and $x + h_2 e_i$ both satisfy the bounds, and then computes

$$\frac{\partial f}{\partial x_i} \cong \frac{h_2^2(f_1 - f_0) - h_1^2(f_2 - f_0)}{h_1 h_2 (h_2 - h_1)},$$

with $f_0 = f(x)$, $f_1 = f(x + h_1 e_i)$, and $f_2 = f(x + h_2 e_i)$.

Table 9.7 summarizes the numerical gradient specification.

55.3 Analytic gradients

The `analytic_gradients` specification means that gradient information is available directly from the simulation (finite differencing is not required). The simulation must return the gradient data in the DAKOTA format (enclosed in single brackets; see DAKOTA File Data Formats in the Users Manual [Adams et al., 2010]) for the case of file transfer of data. The `analytic_gradients` keyword is a complete specification for this case.

Description	Keyword	Associated Data	Status	Default
Numerical gradients	numerical_-gradients	none	Required group	N/A
Method source	method_-source	dakota | vendor	Optional group	dakota
Interval type	interval_-type	forward | central	Optional group	forward
Finite difference step size	fd_-gradient_-step_size	list of reals	Optional	0.001
Ignore variable bounds	ignore_bounds	none	Optional	bounds respected

Table 55.1: Specification detail for numerical gradients

55.4 Mixed gradients

The mixed_gradients specification means that some gradient information is available directly from the simulation (analytic) whereas the rest will have to be finite differenced (numerical). This specification allows the user to make use of as much analytic gradient information as is available and then finite difference for the rest. For example, the objective function may be a simple analytic function of the design variables (e.g., weight) whereas the constraints are nonlinear implicit functions of complex analyses (e.g., maximum stress). The id_-analytic_gradients list specifies by number the functions which have analytic gradients, and the id_-numerical_gradients list specifies by number the functions which must use numerical gradients. Each function identifier, from 1 through the total number of functions, must appear once and only once within the union of the id_analytic_gradients and id_numerical_gradients lists. The method_source, interval_type, and fd_gradient_step_size specifications are as described previously in Numerical gradients and pertain to those functions listed by the id_numerical_gradients list. Table 9.8 summarizes the mixed gradient specification.

Description	Keyword	Associated Data	Status	Default
Mixed gradients	mixed_-gradients	none	Required group	N/A
Analytic derivatives function list	id_-analytic_-gradients	list of integers	Required	N/A
Numerical derivatives function list	id_-numerical_-gradients	list of integers	Required	N/A
Method source	method_-source	dakota | vendor	Optional group	dakota
Interval type	interval_-type	forward | central	Optional group	forward
Finite difference step size	fd_step_size	list of reals	Optional	0.001
Ignore variable bounds	ignore_-bounds	none	Optional	bounds respected

Table 55.2: Specification detail for mixed gradients

Chapter 56

Hessian Specification

Hessian availability must be specified with either `no_hessians`, `numerical_hessians`, `quasi_-hessians`, `analytic_hessians`, or `mixed_hessians`.

56.1 No Hessians

The `no_hessians` specification means that the method does not require DAKOTA to manage the computation of any Hessian information. Therefore, it will neither be retrieved from the simulation nor computed by DAKOTA. The `no_hessians` keyword is a complete specification for this case. Note that, in some cases, Hessian information may still be being approximated internal to an algorithm (e.g., within a quasi-Newton optimizer such as `optpp_q_newton`); however, DAKOTA has no direct involvement in this process and the responses specification need not include it.

56.2 Numerical Hessians

The `numerical_hessians` specification means that Hessian information is needed and will be computed with finite differences using either first-order gradient differencing (for the cases of `analytic_gradients` or for the functions identified by `id_analytic_gradients` in the case of `mixed_gradients`) or first- or second-order function value differencing (all other gradient specifications). In the former case, the following expression

$$\nabla^2 f(\mathbf{x})_i \cong \frac{\nabla f(\mathbf{x} + h\mathbf{e}_i) - \nabla f(\mathbf{x})}{h}$$

estimates the ith Hessian column, and in the latter case, the following expressions

$$\nabla^2 f(\mathbf{x})_{i,j} \cong \frac{f(\mathbf{x} + h_i\mathbf{e}_i + h_j\mathbf{e}_j) - f(\mathbf{x} + h_i\mathbf{e}_i) - f(\mathbf{x} - h_j\mathbf{e}_j) + f(\mathbf{x})}{h_i h_j}$$

and

$$\nabla^2 f(\mathbf{x})_{i,j} \cong \frac{f(\mathbf{x} + h\mathbf{e}_i + h\mathbf{e}_j) - f(\mathbf{x} + h\mathbf{e}_i - h\mathbf{e}_j) - f(\mathbf{x} - h\mathbf{e}_i + h\mathbf{e}_j) + f(\mathbf{x} - h\mathbf{e}_i - h\mathbf{e}_j)}{4h^2}$$

provide first- and second-order estimates of the ij^{th} Hessian term. Prior to DAKOTA 5.0, DAKOTA always used second-order estimates. In DAKOTA 5.0 and newer, the default is to use first-order estimates (which honor bounds on the variables and require only about a quarter as many function evaluations as do the second-order estimates), but specifying `central` after `numerical_hessians` causes DAKOTA to use the old second-order estimates, which do not honor bounds. In optimization algorithms that use Hessians, there is little reason to use second-order differences in computing Hessian approximations.

The `fd_hessian_step_size` specifies the relative finite difference step size to be used in these differences. Either a single value may be entered for use with all parameters, or a list of step sizes may be entered, one for each parameter. The differencing intervals are computed by multiplying the `fd_hessian_step_size` with the current parameter value. A minimum absolute differencing interval of `.01*fd_hessian_step_size` is used when the current parameter value is close to zero. Table 9.9 summarizes the numerical Hessian specification.

Description	Keyword	Associated Data	Status	Default
Numerical Hessians	`numerical_-hessians`	none	Required group	N/A
Finite difference step size	`fd_step_size`	list of reals	Optional	`0.001` (1st-order), `0.002` (2nd-order)
Difference order	`forward` | `central`	none	Optional	forward

Table 56.1: Specification detail for numerical Hessians

56.3 Quasi Hessians

The `quasi_hessians` specification means that Hessian information is needed and will be approximated using secant updates (sometimes called "quasi-Newton updates, though any algorithm that approximates Newton's method is a quasi-Newton method). Compared to finite difference numerical Hessians, secant approximations do not expend additional function evaluations in estimating all of the second-order information for every point of interest. Rather, they accumulate approximate curvature information over time using the existing gradient evaluations. The supported secant approximations include the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update (specified with the keyword `bfgs`)

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

and the Symmetric Rank 1 (SR1) update (specified with the keyword `sr1`)

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}$$

where B_k is the kth approximation to the Hessian, $s_k = x_{k+1} - x_k$ is the step and $y_k = \nabla f_{k+1} - \nabla f_k$ is the corresponding yield in the gradients. In both cases, an initial scaling of $\frac{y_k^T y_k}{y_k^T s_k} I$ is used for B_0 prior to the first update. In addition, both cases employ basic numerical safeguarding to protect against numerically small denominators within the updates. This safeguarding skips the update if $|y_k^T s_k| < 10^{-6} s_k^T B_k s_k$ in the BFGS case or if $|(y_k - B_k s_k)^T s_k| < 10^{-6} ||s_k||_2 ||y_k - B_k s_k||_2$ in the SR1 case. In the BFGS case, additional safeguarding

can be added using the `damped` option, which utilizes an alternative damped BFGS update when the curvature condition $y_k^T s_k > 0$ is nearly violated. Table 9.10 summarizes the quasi Hessian specification.

Description	Keyword	Associated Data	Status	Default
Quasi Hessians	`quasi_-hessians`	`bfgs` | `sr1`	Required group	N/A
Numerical safeguarding of BFGS update	`damped`	none	Optional	undamped BFGS

Table 56.2: Specification detail for quasi Hessians

56.4 Analytic Hessians

The `analytic_hessians` specification means that Hessian information is available directly from the simulation. The simulation must return the Hessian data in the DAKOTA format (enclosed in double brackets; see DAKOTA File Data Formats in Users Manual [Adams et al., 2010]) for the case of file transfer of data. The `analytic_hessians` keyword is a complete specification for this case.

56.5 Mixed Hessians

The `mixed_hessians` specification means that some Hessian information is available directly from the simulation (analytic) whereas the rest will have to be estimated by finite differences (numerical) or approximated by secant updating. As for mixed gradients, this specification allows the user to make use of as much analytic information as is available and then estimate/approximate the rest. The `id_analytic_hessians` list specifies by number the functions which have analytic Hessians, and the `id_numerical_hessians` and `id_quasi_-hessians` lists specify by number the functions which must use numerical Hessians and secant Hessian updates, respectively. Each function identifier, from 1 through the total number of functions, must appear once and only once within the union of the `id_analytic_hessians`, `id_numerical_hessians`, and `id_quasi_-hessians` lists. The `fd_hessian_step_size` and `bfgs`, `damped bfgs`, or `sr1` secant update selections are as described previously in Numerical Hessians and Quasi Hessians and pertain to those functions listed by the `id_numerical_hessians` and `id_quasi_hessians` lists. Table 9.11 summarizes the mixed Hessian specification.

Description	Keyword	Associated Data	Status	Default
Mixed Hessians	`mixed_-hessians`	none	Required group	N/A
Analytic Hessians function list	`id_-analytic_-hessians`	list of integers	Required	N/A
Numerical Hessians function list	`id_-numerical_-hessians`	list of integers	Required	N/A
Finite difference step size	`fd_step_size`	list of reals	Optional	`0.001` (1st-order), `0.002` (2nd-order)
Quasi Hessians function list	`id_quasi_-hessians`	list of integers	Required	N/A
Quasi-Hessian update	`bfgs` | `sr1`	none	Required	N/A
Numerical safeguarding of BFGS update	`damped`	none	Optional	undamped BFGS

Table 56.3: Specification detail for mixed Hessians

Chapter 57

Bibliography

- Anderson, G., and Anderson, P., 1986 *The UNIX C Shell Field Guide*, Prentice-Hall, Englewood Cliffs, NJ.

- Argaez, M., Tapia, R. A., and Velazquez, L., 2002. "Numerical Comparisons of Path-Following Strategies for a Primal-Dual Interior-Point Method for Nonlinear Programming", *Journal of Optimization Theory and Applications*, Vol. 114 (2).

- Breitung, K., 1984. "Asymptotic approximation for multinormal integrals," *J. Eng. Mech., ASCE*, Vol. 110, No. 3, pp. 357-366.

- Byrd, R. H., Schnabel, R. B., and Schultz, G. A., 1988. "Parallel quasi-Newton Methods for Unconstrained Optimization," *Mathematical Programming*, 42(1988), pp. 273-306.

- Conn, A. R. and Gould, N. I. M. and Toint, P. L., 2000. *Trust-Region Methods*, SIAM-MPS, Philadelphia.

- Du,Q., V. Faber, and M. Gunzburger, 1999. "Centroidal Voronoi Tessellations: Applications and Algorithms," *SIAM Review*, Volume 41, 1999, pages 637-676.

- Eddy, J. E. and Lewis, K., 2001. "Effective Generation of Pareto Sets using Genetic Programming," Proceedings of ASME Design Engineering Technical Conference.

- El-Bakry, A. S., Tapia, R. A., Tsuchiya, T., and Zhang, Y., 1996. "On the Formulation and Theory of the Newton Interior-Point Method for Nonlinear Programming," *Journal of Optimization Theory and Applications*, (89) pp. 507-541.

- Bichon, B.J., Eldred, M.S., Swiler, L.P., Mahadevan, S., and McFarland, J.M., 2007. "Multimodal Reliability Assessment for Complex Engineering Applications using Efficient Global Optimization," paper AIAA-2007-1946 in *Proceedings of the 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (9th AIAA Non-Deterministic Approaches Conference)*, Honolulu, HI, April 23-26.

- Bichon, B.J., Eldred, M.S., Swiler, L.P., Mahadevan, S., and McFarland, J.M., 2008. "Efficient Global Reliability Analysis for Nonlinear Implicit Performance Functions," *AIAA Journal*, Vol. 46, No. 10, pp. 2459-2468.

- Eldred, M.S. and Swiler, L.P., 2009. Efficient Algorithms for Mixed Aleatory-Epistemic Uncertainty Quantification with Application to Radiation-Hardened Electronics. Part 1: Algorithms and Benchmark Results, Sandia Technical report SAND2009-5805.

- Eldred, M.S. and Swiler, L.P. and Tang, G., 2011. "Mixed Aleatory-Epistemic Uncertainty Quantification with Stochastic Expansions and Optimization-Based Interval Estimation." *Reliability Engineering and System Safety*, Vol. 96, No. 9, pp. 1092-1113.

- Eldred, M.S. and Dunlavy, D.M., 2006. "Formulations for Surrogate-Based Optimization with Data Fit, Multifidelity, and Reduced-Order Models," paper AIAA-2006-7117 in the *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA, Sept. 6-8.

- Eldred, M.S. and Bichon, B.J., 2006. "Second-Order Reliability Formulations in DAKOTA/UQ," paper AIAA-2006-1828 in *Proceedings of the 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (8th AIAA Non-Deterministic Approaches Conference)*, Newport, Rhode Island, May 1 - 4.

- Adams, B. M., Bohnhoff, W. J., Dalbey, K. R., Eddy, J. P., Eldred, M. S., Hough, P. D., Lefantzi, S., Swiler, L.∼P., and Vigil, D.∼M., 2011. DAKOTA: A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 5.2 Users Manual, Sandia Technical Report SAND2010-2183, updated Nov. 2011.

- Eldred, M. S., Giunta, A. A., and Collis, S. S., 2004. "Second-Order Corrections for Surrogate-Based Optimization with Model Hierarchies," *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, Aug. 30 - Sept. 1.

- Eldred, M.S., Agarwal, H., Perez, V.M., Wojtkiewicz, S.F., Jr., and Renaud, J.E., 2004. "Investigation of Reliability Method Formulations in DAKOTA/UQ," *Proceedings of the 9th ASCE Joint Specialty Conference on Probabilistic Mechanics and Structural Reliability*, Albuquerque, NM, July 26-28.

- Gablonsky, J., 2001. "DIRECT version 2.0 Userguide Technical Report CRSC-TR01-08", Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC.

- Fourer, R., Gay, D. M., and Kernighan, B. W., 2003. "AMPL: A Modeling Language for Mathematical Programming," Duxbury Press/Brooks/Cole Publishing Co. 2nd ed.

- Gay, D. M., 1997. "Hooking Your Solver to AMPL," Technical Report 97-4-06, Bell Laboratories, Murray Hill, NJ. Available as `http://www.ampl.com/REFS/hooking2.ps.gz` .

- Gay, David M., 2008. "Specifying and Reading Program Input with NIDR", Sandia Technical Report SAND2008-2261P. Available as `http://www.sandia.gov/~dmgay/nidr08.pdf` .

- Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H., 1986. "User's Guide for NPSOL (Version 4.0): A Fortran Package for Nonlinear Programming," System Optimization Laboratory Technical Report SOL-86-2, Stanford University, Stanford, CA.

- Giunta, A. A., Swiler, L.P., Brown, S.L, Eldred, M.S., Richards, M.D., and Cyr, E.C., 2006. The Surfpack Software Library for Surrogate Modeling of Sparse, Irregularly Spaced Multidimensional Data, paper AIAA-2006-7049 in *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA.

- Gunburger, M. and J. Burkardt, 2004. "Uniformity Measures for Point Samples in Hypercubes." Available on John Burkardt's web site: http://www.csit.fsu.edu/~burkardt/

- Haftka, R. T., 1991. "Combining Global and Local Approximations," *AIAA Journal*, Vol. 29, No. 9, pp. 1523-1525.

- Haldar, A., and Mahadevan, S., 2000. *Probability, Reliability, and Statistical Methods in Engineering Design*, John Wiley and Sons, New York.

- Halton, J. H. "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numerische Mathematik*,Volume 2 pages 84-90.

- Halton, J. H. and G. B. Smith, 1964. Algorithm 247: Radical-Inverse Quasi-Random Point Sequence, *Communications of the ACM*, Volume 7, pages 701-702.

- Hart, W. E., Giunta, A. A., Salinger, A. G., and van Bloemen Waanders, B. G., 2001. "An Overview of the Adaptive Pattern Search Algorithm and its Application to Engineering Optimization Problems," abstract in *Proceedings of the McMaster Optimization Conference: Theory and Applications*, McMaster University, Hamilton, Ontario, Canada.

- Hart, W. E., and Hunter, K. O., 1999. A Performance Analysis of Evolutionary Pattern Search with Generalized Mutation Steps, *Proc Conf Evolutionary Computation*, pp. 672-679.

- Helton, J.C. and W.L. Oberkampf, 2004. Special Issue of Reliability Engineering and System Safety: Issue on Alternative Representations of Epistemic Uncertainty. Vol 85, Nos. 1-3, July-Sept. 2004.

- Hohenbichler, M. and Rackwitz, R. "Improvement of second-order reliability estimates by importance sampling," *ASCE Journal of Engineering Mechanics*, Vol. 114, No. 12, 1988, pp. 2195-2199.

- Hong, H.P. "Simple Approximations for Improving Second-Order Reliability Estimates," *ASCE Journal of Engineering Mechanics*, Vol. 125, No. 5, 1999, pp. 592-595.

- Gray, G. A. and Kolda, T. G., 2006. Algorithm 856: APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization, *ACM Transactions on Mathematical Software*, Vol. 32, No. 3, pp. 485-507.

- Iman, R. L., and Conover, W. J., 1982. A Distribution-Free Approach to Inducing Rank Correlation Among Input Variables, *Communications in Statistics: Simulation and Computation*, Vol. B11, no. 3, pp. 311-334.

- Jones, D., Schonlau, M., and W. Welch, 1998. "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, Vol. 13, pp. 455-492.

- Kocis, L. and W. Whiten, 1997. "Computational Investigations of Low-Discrepancy Sequences," *ACM Transactions on Mathematical Software*, Volume 23, Number 2, 1997, pages 266-294.

- Lewis, R. M., and Nash, S. G., 2000. A Multigrid Approach to the Optimization of Systems Governed by Differential Equations, paper AIAA-2000-4890 in *Proceedings of the 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Long Beach, CA, Sept. 6-8.

- Meza, J. C., Oliva, R. A., Hough, P. D., and Williams, P. J., 2007. "OPT++: An Object-Oriented Toolkit for Nonlinear Optimization," *ACM Transactions on Mathematical Software* 33(2).

- More, J., and Thuente, D., 1994. "Line Search Algorithms with Guaranteed Sufficient Decrease," *ACM Transactions on Mathematical Software* 20(3):286-307.

- Morris, M. D., "Factorial Sampling Plans for Preliminary Computational Experiments," *Technometrics* 33(2): 161-174.

- Oberkampf, W.L. and J.C. Helton, 2003. "Evidence Theory for Engineering Applications." Sandia National Laboratories Technical Report SAND2003-3559P.

- Perez, V. M., J. E. Renaud, L. T. Watson, 2004. "An Interior-Point Sequetial Approximation Optimization Methodology", *Structural and Multidisciplinary Optimization*, 27(5):360-370.

- Plantenga, T. D., 2009. "HOPSPACK 2.0 User Manual", Sandia National Laboratories Technical Report Sandia National Laboratories Technical Report SAND2009-6265.

- Robinson, D.G. and C. Atcitty, 1999. "Comparison of Quasi- and Pseudo-Monte Carlo Sampling for Reliability and Uncertainty Analysis." *Proceedings of the AIAA Probabilistic Methods Conference*, St. Louis MO, AIAA99-1589.

- Saltelli, A., S. Tarantola, F. Campolongo, and M. Ratto, 2004. "Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models." John Wiley & Sons.

- Schittkowski, K., 2004. NLPQLP: A Fortran Implementation of a Sequential Quadratic Programming Algorithm with Distributed and Non-Monotone Line Search -- User's Guide, Technical Report, Department of Mathematics, University of Bayreuth, Bayreuth, Germany.

- Sjaardema, G. D., 1992. "APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses," Sandia National Laboratories Technical Report SAND92-2291, Albuquerque, NM.

- Stroud, A. H., 1971. "Approximate Calculation of Multiple Integrals," Prentice Hall.

- Tapia, R. A., and Argaez, M., Global Convergence of a Primal-Dual Interior-Point Newton Method for Nonlinear Programming Using a Modified Augmented Lagrangian Function. (In Preparation).

- Tong, C. H., 2005. "The PSUADE Software Library." Lawrence Livermore National Laboratories UCRL UCRL-WEB-211742. Web site: http://www.llnl.gov/CASC/uncertainty_-quantification/#psuade.

- Vanderbei, R. J., and Shanno, D. F., 1999. "An interior-point algorithm for nonconvex nonlinear programming", *Computational Optimization and Applications*, 13:231-259.

- Vanderplaats, G. N., 1973. "CONMIN - A FORTRAN Program for Constrained Function Minimization," NASA TM X-62282. (see also: Addendum to Technical Memorandum, 1978).

- Vanderplaats Research and Development, Inc., 1995. "DOT Users Manual, Version 4.20," Colorado Springs.

- Weirs, V. G. and Kamm, J. R. and Swiler, L. P. and Ratto, M. and Tarantola, S. and Adams, B. M. and Rider, W. J. and Eldred, M. S, 2010. "Sensitivity Analysis Techniques Applied to a System of Hyperbolic Conservation Laws." Submitted to Reliability Engineering and System Safety, Oct. 2010.

- Wright, S. J., 1997. "Primal-Dual Interior-Point Methods", SIAM.

- Wyss, G. D., and Jorgensen, K. H., 1998. "A User s Guide to LHS: Sandia's Latin Hypercube Sampling Software," Sandia National Laboratories Technical Report SAND98-0210, Albuquerque, NM.

- Xiu, D., 2008. "Numerical integration formulas of degree two," Applied Numerical Mathematics, Vol. 58, pp. 1515-1520.

- Xu, S. and Grandhi, R. V., 1998. "Effective Two-Point Function Approximation for Design Optimization," AIAA Journal, Vol. 36, No. 12, December 1998.

Chapter 58

File Index

Chapter 59

File Documentation

59.1 dakota.input.summary File Reference

File containing the input specification for DAKOTA.

59.1.1 Detailed Description

File containing the input specification for DAKOTA. This file is derived automatically from dakota.input.nspec, which is used in the generation of parser system files that are compiled into the DAKOTA executable. Therefore, these files are the definitive source for input syntax, capability options, and associated data inputs. Refer to **Instructions for Modifying DAKOTA's Input Specification** for information on how to modify the input specification and propagate the changes through the parsing system.

Key features of the input specification and the associated user input files include:

- In the input specification, required individual specifications simply appear, optional individual and group specifications are enclosed in [], required group specifications are enclosed in (), and either-or relationships are denoted by the | symbol. These symbols only appear in dakota.input.nspec; they must not appear in actual user input files.

- Keyword specifications (i.e., `strategy`, `method`, `model`, `variables`, `interface`, and `responses`) begin with the keyword possibly preceded by white space (blanks, tabs, and newlines) both in the input specifications and in user input files. For readability, keyword specifications may be spread across several lines. Earlier versions of DAKOTA (prior to 4.1) required a backslash character (\) at the ends of intermediate lines of a keyword. While such backslashes are still accepted, they are no longer required.

- Some of the keyword components within the input specification indicate that the user must supply `INTEGER`, `REAL`, `STRING`, `INTEGERLIST`, `REALLIST`, or `STRINGLIST` data as part of the specification. In a user input file, the `"="` is optional, data in a `LIST` can be separated by commas or whitespace, and the `STRING` data are enclosed in single or double quotes (e.g., `'text_book'` or "text_book").

- In user input files, input is largely order-independent (except for entries in lists of data), case insensitive, and white-space insensitive. Although the order of input shown in the Sample dakota.in Files generally follows the order of options in the input specification, this is not required.

- In user input files, specifications may be abbreviated so long as the abbreviation is unique. For example, the `npsol_sqp` specification within the method keyword could be abbreviated as `npsol`, but `dot_sqp` should not be abbreviated as `dot` since this would be ambiguous with other DOT method specifications.

- In both the input specification and user input files, comments are preceded by `#`.

The dakota.input.summary file for DAKOTA version 5.1 is:

```
KEYWORD01 strategy
        [ graphics ]
        [ tabular_graphics_data
          [ tabular_graphics_file STRING ]
          ]
        [ output_precision INTEGER >= 0 ]
        [ iterator_servers INTEGER ]
        [ iterator_self_scheduling ]
        [ iterator_static_scheduling ]
        ( hybrid
          ( sequential ALIAS uncoupled
            method_list STRINGLIST
            )
          |
          ( embedded ALIAS coupled
            global_method_pointer STRING
            local_method_pointer STRING
            [ local_search_probability REAL ]
            )
          |
          ( collaborative
            method_list STRINGLIST
            )
          )
        |
        ( multi_start
          method_pointer STRING
          [ random_starts INTEGER
            [ seed INTEGER ]
            ]
          [ starting_points REALLIST ]
          )
        |
        ( pareto_set
          method_pointer ALIAS opt_method_pointer STRING
          [ random_weight_sets INTEGER
            [ seed INTEGER ]
            ]
          [ weight_sets ALIAS multi_objective_weight_sets REALLIST ]
          )
        |
        ( single_method
          [ method_pointer STRING ]
          )

KEYWORD12 method
```

```
[ id_method STRING ]
[ model_pointer STRING ]
[ output
  debug
  | verbose
  | normal
  | quiet
  | silent
  ]
[ max_iterations INTEGER ]
[ max_function_evaluations INTEGER ]
[ speculative ]
[ convergence_tolerance REAL ]
[ constraint_tolerance REAL ]
[ scaling ]
[ final_solutions INTEGER >= 0 ]
( surrogate_based_local
  approx_method_name STRING
  | approx_method_pointer STRING
  [ soft_convergence_limit INTEGER ]
  [ truth_surrogate_bypass ]
  [ trust_region
    [ initial_size REAL ]
    [ minimum_size REAL ]
    [ contract_threshold REAL ]
    [ expand_threshold REAL ]
    [ contraction_factor REAL ]
    [ expansion_factor REAL ]
    ]
  [ approx_subproblem
    original_primary
    | single_objective
    | augmented_lagrangian_objective
    | lagrangian_objective
    original_constraints
    | linearized_constraints
    | no_constraints
    ]
  [ merit_function
    penalty_merit
    | adaptive_penalty_merit
    | lagrangian_merit
    | augmented_lagrangian_merit
    ]
  [ acceptance_logic
    tr_ratio
    | filter
    ]
  [ constraint_relax
    homotopy
    ]
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
|
```

```
( dot_frcg
  [ optimization_type
    minimize
    | maximize
    ]
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
| dot_mmfd
| dot_bfgs
| dot_slp
| dot_sqp
|
( dot
  frcg
  | mmfd
  | bfgs
  | slp
  | sqp
  [ optimization_type
    minimize
    | maximize
    ]
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
|
( conmin_frcg
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
|
( conmin_mfd
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
```

```
        [ linear_equality_scales REALLIST ]
        )
      |
      ( dl_solver STRING
        [ linear_inequality_constraint_matrix REALLIST ]
        [ linear_inequality_lower_bounds REALLIST ]
        [ linear_inequality_upper_bounds REALLIST ]
        [ linear_inequality_scale_types STRINGLIST ]
        [ linear_inequality_scales REALLIST ]
        [ linear_equality_constraint_matrix REALLIST ]
        [ linear_equality_targets REALLIST ]
        [ linear_equality_scale_types STRINGLIST ]
        [ linear_equality_scales REALLIST ]
        )
      |
      ( npsol_sqp
        [ verify_level INTEGER ]
        [ function_precision REAL ]
        [ linesearch_tolerance REAL ]
        [ linear_inequality_constraint_matrix REALLIST ]
        [ linear_inequality_lower_bounds REALLIST ]
        [ linear_inequality_upper_bounds REALLIST ]
        [ linear_inequality_scale_types STRINGLIST ]
        [ linear_inequality_scales REALLIST ]
        [ linear_equality_constraint_matrix REALLIST ]
        [ linear_equality_targets REALLIST ]
        [ linear_equality_scale_types STRINGLIST ]
        [ linear_equality_scales REALLIST ]
        )
      | nlssol_sqp
      |
      ( stanford
        npsol
        | nlssol
        [ verify_level INTEGER ]
        [ function_precision REAL ]
        [ linesearch_tolerance REAL ]
        [ linear_inequality_constraint_matrix REALLIST ]
        [ linear_inequality_lower_bounds REALLIST ]
        [ linear_inequality_upper_bounds REALLIST ]
        [ linear_inequality_scale_types STRINGLIST ]
        [ linear_inequality_scales REALLIST ]
        [ linear_equality_constraint_matrix REALLIST ]
        [ linear_equality_targets REALLIST ]
        [ linear_equality_scale_types STRINGLIST ]
        [ linear_equality_scales REALLIST ]
        )
      |
      ( nlpql_sqp
        [ linear_inequality_constraint_matrix REALLIST ]
        [ linear_inequality_lower_bounds REALLIST ]
        [ linear_inequality_upper_bounds REALLIST ]
        [ linear_inequality_scale_types STRINGLIST ]
        [ linear_inequality_scales REALLIST ]
        [ linear_equality_constraint_matrix REALLIST ]
        [ linear_equality_targets REALLIST ]
        [ linear_equality_scale_types STRINGLIST ]
        [ linear_equality_scales REALLIST ]
        )
      |
      ( optpp_cg
        [ max_step REAL ]
```

```
    [ gradient_tolerance REAL ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( optpp_q_newton
    | optpp_fd_newton
    | optpp_g_newton
    | optpp_newton
    [ search_method
      value_based_line_search
      | gradient_based_line_search
      | trust_region
      | tr_pds
      ]
    [ merit_function STRING ]
    [ central_path STRING ]
    [ steplength_to_boundary REAL ]
    [ centering_parameter REAL ]
    [ max_step REAL ]
    [ gradient_tolerance REAL ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( optpp_pds
    [ search_scheme_size INTEGER ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( asynch_pattern_search ALIAS coliny_apps
    [ initial_delta REAL ]
    [ contraction_factor REAL ]
    [ threshold_delta REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ synchronization
      blocking
      | nonblocking
      ]
    [ merit_function
```

```
      merit_max
      | merit_max_smooth
      | merit1
      | merit1_smooth
      | merit2
      | merit2_smooth
      | merit2_squared
      ]
    [ constraint_penalty REAL ]
    [ smoothing_factor REAL ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( moga
    [ fitness_type
      layer_rank
      | domination_count
      ]
    [ replacement_type
      elitist
      | roulette_wheel
      | unique_roulette_wheel
      |
      ( below_limit REAL
        [ shrinkage_fraction ALIAS shrinkage_percentage REAL ]
        )
      ]
    [ niching_type
      radial REALLIST
      | distance REALLIST
      ]
    [ convergence_type
      metric_tracker
      [ percent_change REAL ]
      [ num_generations INTEGER >= 0 ]
      ]
    [ postprocessor_type
      orthogonal_distance REALLIST
      ]
    [ population_size INTEGER ]
    [ log_file STRING ]
    [ print_each_pop ]
    [ initialization_type
      simple_random
      | unique_random
      | flat_file STRING
      ]
    [ crossover_type
      multi_point_binary INTEGER
      | multi_point_parameterized_binary INTEGER
      | multi_point_real INTEGER
      |
      ( shuffle_random
        [ num_parents INTEGER > 0 ]
```

```
        [ num_offspring INTEGER > 0 ]
        )
      [ crossover_rate REAL ]
      ]
    [ mutation_type
      bit_random
      | replace_uniform
      |
      ( offset_normal
        | offset_cauchy
        | offset_uniform
        [ mutation_scale REAL ]
        )
      [ mutation_rate REAL ]
      ]
    [ seed INTEGER ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( soga
    [ fitness_type
      merit_function
      [ constraint_penalty REAL ]
      ]
    [ replacement_type
      elitist
      | favor_feasible
      | roulette_wheel
      | unique_roulette_wheel
      ]
    [ convergence_type
      ( best_fitness_tracker
        [ percent_change REAL ]
        [ num_generations INTEGER >= 0 ]
        )
      |
      ( average_fitness_tracker
        [ percent_change REAL ]
        [ num_generations INTEGER >= 0 ]
        )
      ]
    [ population_size INTEGER ]
    [ log_file STRING ]
    [ print_each_pop ]
    [ initialization_type
      simple_random
      | unique_random
      | flat_file STRING
      ]
    [ crossover_type
      multi_point_binary INTEGER
      | multi_point_parameterized_binary INTEGER
      | multi_point_real INTEGER
      |
```

```
      ( shuffle_random
        [ num_parents INTEGER > 0 ]
        [ num_offspring INTEGER > 0 ]
        )
      [ crossover_rate REAL ]
      ]
    [ mutation_type
      bit_random
      | replace_uniform
      |
      ( offset_normal
        | offset_cauchy
        | offset_uniform
        [ mutation_scale REAL ]
        )
      [ mutation_rate REAL ]
      ]
    [ seed INTEGER ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( coliny_pattern_search
    [ constant_penalty ]
    [ no_expansion ]
    [ expand_after_success INTEGER ]
    [ pattern_basis
      coordinate
      | simplex
      ]
    [ stochastic ]
    [ total_pattern_size INTEGER ]
    [ exploratory_moves
      multi_step
      | adaptive_pattern
      | basic_pattern
      ]
    [ synchronization
      blocking
      | nonblocking
      ]
    [ contraction_factor REAL ]
    [ constraint_penalty REAL ]
    initial_delta REAL
    threshold_delta REAL
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_solis_wets
    [ contract_after_failure INTEGER ]
    [ no_expansion ]
    [ expand_after_success INTEGER ]
```

```
    [ constant_penalty ]
    [ contraction_factor REAL ]
    [ constraint_penalty REAL ]
    initial_delta REAL
    threshold_delta REAL
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_cobyla
    initial_delta REAL
    threshold_delta REAL
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_direct
    [ division
      major_dimension
      | all_dimensions
      ]
    [ global_balance_parameter REAL ]
    [ local_balance_parameter REAL ]
    [ max_boxsize_limit REAL ]
    [ min_boxsize_limit REAL ]
    [ constraint_penalty REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_ea
    [ population_size INTEGER ]
    [ initialization_type
      simple_random
      | unique_random
      | flat_file STRING
      ]
    [ fitness_type
      linear_rank
      | merit_function
      ]
    [ replacement_type
      random INTEGER
      | chc INTEGER
      | elitist INTEGER
      [ new_solutions_generated INTEGER ]
      ]
    [ crossover_rate REAL ]
    [ crossover_type
      two_point
      | blend
      | uniform
      ]
    [ mutation_rate REAL ]
    [ mutation_type
      replace_uniform
```

```
      |
      ( offset_normal
        | offset_cauchy
        | offset_uniform
        [ mutation_scale REAL ]
        [ mutation_range INTEGER ]
        )
      [ non_adaptive ]
      ]
    [ constraint_penalty REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( nl2sol
    [ function_precision REAL ]
    [ absolute_conv_tol REAL ]
    [ x_conv_tol REAL ]
    [ singular_conv_tol REAL ]
    [ singular_radius REAL ]
    [ false_conv_tol REAL ]
    [ initial_trust_radius REAL ]
    [ covariance INTEGER ]
    [ regression_diagnostics ]
    )
  |
  ( nonlinear_cg
    [ misc_options STRINGLIST ]
    )
  |
  ( surrogate_based_global
    approx_method_name STRING
    | approx_method_pointer STRING
    [ replace_points ]
    )
  |
  ( efficient_global
    [ gaussian_process ALIAS kriging
      surfpack
      | dakota
      ]
    [ use_derivatives ]
    [ seed INTEGER ]
    )
  |
  ( polynomial_chaos ALIAS nond_polynomial_chaos
    [ p_refinement
      uniform
      |
      ( dimension_adaptive
        sobol
        | decay
        | generalized
        )
      ]
    [ askey
    | wiener ]
    ( quadrature_order INTEGERLIST
      [ dimension_preference REALLIST ]
      [ nested
```

```
    | non_nested ]
    )
  |
  ( sparse_grid_level INTEGERLIST
    [ restricted
    | unrestricted ]
    [ dimension_preference REALLIST ]
    [ nested
    | non_nested ]
    )
  | cubature_integrand INTEGER
  |
  ( collocation_points INTEGER
    | collocation_ratio REAL
    [ ratio_order REAL ]
    [ reuse_points ALIAS reuse_samples ]
    [ use_derivatives ]
    [ tensor_grid ]
    expansion_order INTEGERLIST
    | expansion_terms INTEGER
    )
  |
  ( expansion_samples INTEGER
    [ incremental_lhs ]
    expansion_order INTEGERLIST
    | expansion_terms INTEGER
    )
  |
  ( expansion_import_file STRING
    expansion_order INTEGERLIST
    | expansion_terms INTEGER
    )
  [ variance_based_decomp
    [ univariate_effects ]
    [ drop_tolerance REAL ]
    ]
  [ sample_refinement
    import
    | adapt_import
    | mm_adapt_import
    ]
  [ sample_type
    lhs
    | random
    ]
  [ all_variables ]
  [ fixed_seed ]
  [ reliability_levels REALLIST
    [ num_reliability_levels INTEGERLIST ]
    ]
  [ response_levels REALLIST
    [ num_response_levels INTEGERLIST ]
    [ compute
      probabilities
      | reliabilities
      | gen_reliabilities
      ]
    ]
  [ distribution
    cumulative
    | complementary
    ]
```

```
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( stoch_collocation ALIAS nond_stoch_collocation
    [ p_refinement
      | h_refinement
      uniform
      |
      ( dimension_adaptive
        sobol
        | generalized
        )
      ]
    [ ( piecewise
        [ nodal
        | hierarchical ]
        )
    | askey
    | wiener ]
    quadrature_order INTEGERLIST
    |
    ( sparse_grid_level INTEGERLIST
      [ restricted
      | unrestricted ]
      )
    [ dimension_preference REALLIST ]
    [ use_derivatives ]
    [ nested
    | non_nested ]
    [ variance_based_decomp
      [ univariate_effects ]
      [ drop_tolerance REAL ]
      ]
    [ sample_refinement
      import
      | adapt_import
      | mm_adapt_import
      ]
    [ sample_type
      lhs
      | random
      ]
    [ all_variables ]
    [ fixed_seed ]
    [ reliability_levels REALLIST
      [ num_reliability_levels INTEGERLIST ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
```

```
        | reliabilities
        | gen_reliabilities
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( sampling ALIAS nond_sampling
    [ sample_type
      random
      | lhs
      |
      ( incremental_lhs
        | incremental_random
        previous_samples INTEGER
        )
      ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ all_variables ]
    [ fixed_seed ]
    [ reliability_levels REALLIST
      [ num_reliability_levels INTEGERLIST ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | reliabilities
        | gen_reliabilities
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
```

```
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( importance_sampling ALIAS nond_importance_sampling
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( global_evidence ALIAS nond_global_evidence
    [ ( ego
        [ gaussian_process ALIAS kriging
          surfpack
          | dakota
          ]
        [ use_derivatives ]
        )
    | lhs ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER ]
```

```
    )
  |
  ( global_interval_est ALIAS nond_global_interval_est
    [ ( ego
        [ gaussian_process ALIAS kriging
          surfpack
          | dakota
          ]
        [ use_derivatives ]
        )
    | lhs ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( bayes_calibration ALIAS nond_bayes_calibration
    ( queso
      [ emulator
        ( gaussian_process ALIAS gp
          [ emulator_samples INTEGER ]
          )
        |
        ( pce
          [ sparse_grid_level INTEGERLIST ]
          )
        |
        ( sc
          [ sparse_grid_level INTEGERLIST ]
          )
        ]
      )
    |
    ( gpmsa
      emulator_samples INTEGER
      )
    [ rejection
      standard
      | delayed
      ]
    [ metropolis
      hastings
      | adaptive
      ]
    [ use_derivatives ]
    [ rng
      mt19937
      | rnum2
      ]
    [ proposal_covariance_scale REAL ]
    [ likelihood_scale REAL ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( dace
    grid
    | random
    | oas
```

```
    | lhs
    | oa_lhs
    | box_behnken
    | central_composite
    [ main_effects ]
    [ quality_metrics ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ fixed_seed ]
    [ symbols INTEGER ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( fsu_cvt
    [ latinize ]
    [ quality_metrics ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ fixed_seed ]
    [ trial_type
      grid
      | halton
      | random
      ]
    [ num_trials INTEGER ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( psuade_moat
    [ partitions INTEGERLIST ]
    [ samples INTEGER ]
    [ seed INTEGER ]
    )
  |
  ( ncsu_direct
    [ solution_target ALIAS solution_accuracy REAL ]
    [ min_boxsize_limit REAL ]
    [ volume_boxsize_limit REAL ]
    )
  |
  ( local_evidence ALIAS nond_local_evidence
    [ sqp
    | nip ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        ]
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ distribution
      cumulative
```

```
      | complementary
      ]
    )
  |
  ( local_interval_est ALIAS nond_local_interval_est
    [ sqp
    | nip ]
    )
  |
  ( local_reliability ALIAS nond_local_reliability
    [ mpp_search
      x_taylor_mean
      | u_taylor_mean
      | x_taylor_mpp
      | u_taylor_mpp
      | x_two_point
      | u_two_point
      | no_approx
      [ sqp
      | nip ]
      ]
    [ integration
      first_order
      | second_order
      [ sample_refinement
        import
        | adapt_import
        | mm_adapt_import
        [ samples INTEGER ]
        [ seed INTEGER ]
        ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | reliabilities
        | gen_reliabilities
        ]
      ]
    [ reliability_levels REALLIST
      [ num_reliability_levels INTEGERLIST ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    )
  |
  ( global_reliability ALIAS nond_global_reliability
    ( x_gaussian_process ALIAS x_kriging
      | u_gaussian_process ALIAS u_kriging
      [ surfpack
      | dakota ]
      )
    [ all_variables ]
```

```
    [ use_derivatives ]
    [ seed INTEGER ]
    [ rng
      mt19937
      | rnum2
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    )
  |
  ( fsu_quasi_mc
    halton
    | hammersley
    [ latinize ]
    [ quality_metrics ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ samples INTEGER ]
    [ fixed_sequence ]
    [ sequence_start INTEGERLIST ]
    [ sequence_leap INTEGERLIST ]
    [ prime_base INTEGERLIST ]
    )
  |
  ( vector_parameter_study
    final_point REALLIST
    | step_vector REALLIST
    num_steps INTEGER
    )
  |
  ( list_parameter_study
    list_of_points REALLIST
    )
  |
  ( centered_parameter_study
    step_vector REALLIST
    steps_per_variable ALIAS deltas_per_variable INTEGERLIST
    )
  |
  ( multidim_parameter_study
    partitions INTEGERLIST
    )
  |
  ( richardson_extrap
    estimate_order
    | converge_order
```

```
          | converge_qoi
          [ refinement_rate REAL ]
          )

KEYWORD model
        [ id_model STRING ]
        [ variables_pointer STRING ]
        [ responses_pointer STRING ]
        ( single
          [ interface_pointer STRING ]
          )
        |
        ( surrogate
          [ id_surrogates INTEGERLIST ]
          ( global
            ( gaussian_process ALIAS kriging
              ( dakota
                [ point_selection ]
                [ trend
                  constant
                  | linear
                  | reduced_quadratic
                  ]
                )
              |
              ( surfpack
                [ trend
                  constant
                  | linear
                  | reduced_quadratic
                  | quadratic
                  ]
                [ optimization_method STRING ]
                [ max_trials INTEGER > 0 ]
                [ correlation_lengths REALLIST ]
                )
              )
            |
            ( mars
              [ max_bases INTEGER ]
              [ interpolation
                linear
                | cubic
                ]
              )
            |
            ( moving_least_squares
              [ poly_order INTEGER ]
              [ weight_function INTEGER ]
              )
            |
            ( neural_network
              [ nodes INTEGER ]
              [ range REAL ]
              [ random_weight INTEGER ]
              )
            |
            ( radial_basis
              [ bases INTEGER ]
              [ max_pts INTEGER ]
              [ min_partition INTEGER ]
              [ max_subsets INTEGER ]
```

```
          )
        |
        ( polynomial
          linear
          | quadratic
          | cubic
          )
        [ total_points INTEGER
        | minimum_points
        | recommended_points ]
        [ dace_method_pointer STRING ]
        [ reuse_points ALIAS reuse_samples
          all
          | region
          | none
          ]
        [ points_file ALIAS samples_file STRING
          [ annotated
          | freeform ]
          ]
        [ use_derivatives ]
        [ correction
          zeroth_order
          | first_order
          | second_order
          additive
          | multiplicative
          | combined
          ]
        [ diagnostics STRINGLIST ]
        )
      |
      ( multipoint
        tana
        actual_model_pointer STRING
        )
      |
      ( local
        taylor_series
        actual_model_pointer STRING
        )
      |
      ( hierarchical
        low_fidelity_model_pointer STRING
        high_fidelity_model_pointer STRING
        ( correction
          zeroth_order
          | first_order
          | second_order
          additive
          | multiplicative
          | combined
          )
        )
      )
    |
    ( nested
      [ optional_interface_pointer STRING
        [ optional_interface_responses_pointer STRING ]
        ]
      ( sub_method_pointer STRING
        [ primary_variable_mapping STRINGLIST ]
```

```
            [ secondary_variable_mapping STRINGLIST ]
            [ primary_response_mapping REALLIST ]
            [ secondary_response_mapping REALLIST ]
            )
          )

KEYWORD12 variables
        [ id_variables STRING ]
        [ continuous_design INTEGER > 0
          [ initial_point ALIAS cdv_initial_point REALLIST ]
          [ lower_bounds ALIAS cdv_lower_bounds REALLIST ]
          [ upper_bounds ALIAS cdv_upper_bounds REALLIST ]
          [ scale_types ALIAS cdv_scale_types STRINGLIST ]
          [ scales ALIAS cdv_scales REALLIST ]
          [ descriptors ALIAS cdv_descriptors STRINGLIST ]
          ]
        [ discrete_design_range INTEGER > 0
          [ initial_point ALIAS ddv_initial_point INTEGERLIST ]
          [ lower_bounds ALIAS ddv_lower_bounds INTEGERLIST ]
          [ upper_bounds ALIAS ddv_upper_bounds INTEGERLIST ]
          [ descriptors ALIAS ddv_descriptors STRINGLIST ]
          ]
        [ discrete_design_set_integer INTEGER > 0
          [ initial_point INTEGERLIST ]
          [ num_set_values INTEGERLIST ]
          set_values INTEGERLIST
          [ descriptors STRINGLIST ]
          ]
        [ discrete_design_set_real INTEGER > 0
          [ initial_point REALLIST ]
          [ num_set_values INTEGERLIST ]
          set_values REALLIST
          [ descriptors STRINGLIST ]
          ]
        [ normal_uncertain INTEGER > 0
          means ALIAS nuv_means REALLIST
          std_deviations ALIAS nuv_std_deviations REALLIST
          [ lower_bounds ALIAS nuv_lower_bounds REALLIST ]
          [ upper_bounds ALIAS nuv_upper_bounds REALLIST ]
          [ descriptors ALIAS nuv_descriptors STRINGLIST ]
          ]
        [ lognormal_uncertain INTEGER > 0
          ( lambdas ALIAS lnuv_lambdas REALLIST
            zetas ALIAS lnuv_zetas REALLIST
            )
          |
          ( means ALIAS lnuv_means REALLIST
            std_deviations ALIAS lnuv_std_deviations REALLIST
            | error_factors ALIAS lnuv_error_factors REALLIST
            )
          [ lower_bounds ALIAS lnuv_lower_bounds REALLIST ]
          [ upper_bounds ALIAS lnuv_upper_bounds REALLIST ]
          [ descriptors ALIAS lnuv_descriptors STRINGLIST ]
          ]
        [ uniform_uncertain INTEGER > 0
          lower_bounds ALIAS uuv_lower_bounds REALLIST
          upper_bounds ALIAS uuv_upper_bounds REALLIST
          [ descriptors ALIAS uuv_descriptors STRINGLIST ]
          ]
        [ loguniform_uncertain INTEGER > 0
          lower_bounds ALIAS luuv_lower_bounds REALLIST
          upper_bounds ALIAS luuv_upper_bounds REALLIST
```

```
    [ descriptors ALIAS luuv_descriptors STRINGLIST ]
    ]
  [ triangular_uncertain INTEGER > 0
    modes ALIAS tuv_modes REALLIST
    lower_bounds ALIAS tuv_lower_bounds REALLIST
    upper_bounds ALIAS tuv_upper_bounds REALLIST
    [ descriptors ALIAS tuv_descriptors STRINGLIST ]
    ]
  [ exponential_uncertain INTEGER > 0
    betas ALIAS euv_betas REALLIST
    [ descriptors ALIAS euv_descriptors STRINGLIST ]
    ]
  [ beta_uncertain INTEGER > 0
    alphas ALIAS buv_alphas REALLIST
    betas ALIAS buv_betas REALLIST
    lower_bounds ALIAS buv_lower_bounds REALLIST
    upper_bounds ALIAS buv_upper_bounds REALLIST
    [ descriptors ALIAS buv_descriptors STRINGLIST ]
    ]
  [ gamma_uncertain INTEGER > 0
    alphas ALIAS gauv_alphas REALLIST
    betas ALIAS gauv_betas REALLIST
    [ descriptors ALIAS gauv_descriptors STRINGLIST ]
    ]
  [ gumbel_uncertain INTEGER > 0
    alphas ALIAS guuv_alphas REALLIST
    betas ALIAS guuv_betas REALLIST
    [ descriptors ALIAS guuv_descriptors STRINGLIST ]
    ]
  [ frechet_uncertain INTEGER > 0
    alphas ALIAS fuv_alphas REALLIST
    betas ALIAS fuv_betas REALLIST
    [ descriptors ALIAS fuv_descriptors STRINGLIST ]
    ]
  [ weibull_uncertain INTEGER > 0
    alphas ALIAS wuv_alphas REALLIST
    betas ALIAS wuv_betas REALLIST
    [ descriptors ALIAS wuv_descriptors STRINGLIST ]
    ]
  [ histogram_bin_uncertain INTEGER > 0
    [ num_pairs ALIAS huv_num_bin_pairs INTEGERLIST ]
    abscissas ALIAS huv_bin_abscissas REALLIST
    ordinates ALIAS huv_bin_ordinates REALLIST
    | counts ALIAS huv_bin_counts REALLIST
    [ descriptors ALIAS huv_bin_descriptors STRINGLIST ]
    ]
  [ poisson_uncertain INTEGER > 0
    lambdas REALLIST
    [ descriptors STRINGLIST ]
    ]
  [ binomial_uncertain INTEGER > 0
    prob_per_trial REALLIST
    num_trials INTEGERLIST
    [ descriptors STRINGLIST ]
    ]
  [ negative_binomial_uncertain INTEGER > 0
    prob_per_trial REALLIST
    num_trials INTEGERLIST
    [ descriptors STRINGLIST ]
    ]
  [ geometric_uncertain INTEGER > 0
    prob_per_trial REALLIST
```

```
          [ descriptors STRINGLIST ]
          ]
        [ hypergeometric_uncertain INTEGER > 0
          total_population INTEGERLIST
          selected_population INTEGERLIST
          num_drawn INTEGERLIST
          [ descriptors STRINGLIST ]
          ]
        [ histogram_point_uncertain INTEGER > 0
          [ num_pairs ALIAS huv_num_point_pairs INTEGERLIST ]
          abscissas ALIAS huv_point_abscissas REALLIST
          counts ALIAS huv_point_counts REALLIST
          [ descriptors ALIAS huv_point_descriptors STRINGLIST ]
          ]
        [ uncertain_correlation_matrix REALLIST ]
        [ interval_uncertain INTEGER > 0
          [ num_intervals ALIAS iuv_num_intervals INTEGERLIST ]
          interval_probs ALIAS iuv_interval_probs REALLIST
          interval_bounds ALIAS iuv_interval_bounds REALLIST
          [ descriptors ALIAS iuv_descriptors STRINGLIST ]
          ]
        [ continuous_state INTEGER > 0
          [ initial_state ALIAS csv_initial_state REALLIST ]
          [ lower_bounds ALIAS csv_lower_bounds REALLIST ]
          [ upper_bounds ALIAS csv_upper_bounds REALLIST ]
          [ descriptors ALIAS csv_descriptors STRINGLIST ]
          ]
        [ discrete_state_range INTEGER > 0
          [ initial_state ALIAS dsv_initial_state INTEGERLIST ]
          [ lower_bounds ALIAS dsv_lower_bounds INTEGERLIST ]
          [ upper_bounds ALIAS dsv_upper_bounds INTEGERLIST ]
          [ descriptors ALIAS dsv_descriptors STRINGLIST ]
          ]
        [ discrete_state_set_integer INTEGER > 0
          [ initial_state INTEGERLIST ]
          [ num_set_values INTEGERLIST ]
          set_values INTEGERLIST
          [ descriptors STRINGLIST ]
          ]
        [ discrete_state_set_real INTEGER > 0
          [ initial_state REALLIST ]
          [ num_set_values INTEGERLIST ]
          set_values REALLIST
          [ descriptors STRINGLIST ]
          ]

KEYWORD12 interface
        [ id_interface STRING ]
        [ algebraic_mappings STRING ]
        [ analysis_drivers STRINGLIST
          [ analysis_components STRINGLIST ]
          [ input_filter STRING ]
          [ output_filter STRING ]
          ( system
            | fork
            [ parameters_file STRING ]
            [ results_file STRING ]
            [ allow_existing_results ]
            [ verbatim ]
            [ aprepro ]
            [ file_tag ]
            [ file_save ]
```

```
            [ work_directory
              [ named STRING ]
              [ directory_tag ALIAS dir_tag ]
              [ directory_save ALIAS dir_save ]
              [ template_directory STRING
                | template_files STRINGLIST
                [ copy ]
                [ replace ]
                ]
              ]
            )
          |
          ( direct
            [ processors_per_analysis INTEGER ]
            )
          | grid
          [ failure_capture
            abort
            | retry INTEGER
            | recover REALLIST
            | continuation
            ]
          [ deactivate
            [ active_set_vector ]
            [ evaluation_cache ]
            [ restart_file ]
            ]
          ]
        [ asynchronous
          [ evaluation_concurrency INTEGER ]
          [ local_evaluation_self_scheduling
          | local_evaluation_static_scheduling ]
          [ analysis_concurrency INTEGER ]
          ]
        [ evaluation_servers INTEGER ]
        [ evaluation_self_scheduling
        | evaluation_static_scheduling ]
        [ analysis_servers INTEGER ]
        [ analysis_self_scheduling
        | analysis_static_scheduling ]

KEYWORD12 responses
        [ id_responses STRING ]
        [ descriptors ALIAS response_descriptors STRINGLIST ]
        ( objective_functions ALIAS num_objective_functions INTEGER >= 0
          [ objective_function_scale_types STRINGLIST ]
          [ objective_function_scales REALLIST ]
          [ multi_objective_weights REALLIST ]
          [ nonlinear_inequality_constraints ALIAS num_nonlinear_inequality_const
      raints INTEGER >= 0
            [ nonlinear_inequality_lower_bounds REALLIST ]
            [ nonlinear_inequality_upper_bounds REALLIST ]
            [ nonlinear_inequality_scale_types STRINGLIST ]
            [ nonlinear_inequality_scales REALLIST ]
            ]
          [ nonlinear_equality_constraints ALIAS num_nonlinear_equality_constrain
      ts INTEGER >= 0
            [ nonlinear_equality_targets REALLIST ]
            [ nonlinear_equality_scale_types STRINGLIST ]
            [ nonlinear_equality_scales REALLIST ]
            ]
          )
```

```
  |
  ( calibration_terms ALIAS num_least_squares_terms INTEGER >= 0
    [ calibration_term_scale_types ALIAS least_squares_term_scale_types STR
INGLIST ]
    [ calibration_term_scales ALIAS least_squares_term_scales REALLIST ]
    [ calibration_weights ALIAS least_squares_weights REALLIST ]
    [ calibration_data_file ALIAS least_squares_data_file STRING
      [ num_experiments INTEGER >= 0 ]
      [ annotated
      | freeform ]
      [ num_config_variables INTEGER >= 0 ]
      [ num_std_deviations INTEGER >= 0 ]
      ]
    [ nonlinear_inequality_constraints ALIAS num_nonlinear_inequality_const
raints INTEGER >= 0
      [ nonlinear_inequality_lower_bounds REALLIST ]
      [ nonlinear_inequality_upper_bounds REALLIST ]
      [ nonlinear_inequality_scale_types STRINGLIST ]
      [ nonlinear_inequality_scales REALLIST ]
      ]
    [ nonlinear_equality_constraints ALIAS num_nonlinear_equality_constrain
ts INTEGER >= 0
      [ nonlinear_equality_targets REALLIST ]
      [ nonlinear_equality_scale_types STRINGLIST ]
      [ nonlinear_equality_scales REALLIST ]
      ]
    )
  | response_functions ALIAS num_response_functions INTEGER >= 0
  no_gradients
  | analytic_gradients
  |
  ( mixed_gradients
    id_numerical_gradients INTEGERLIST
    id_analytic_gradients INTEGERLIST
    [ method_source ]
    [ ( dakota
        [ ignore_bounds ]
        )
    | vendor ]
    [ interval_type ]
    [ forward
    | central ]
    [ fd_step_size ALIAS fd_gradient_step_size REALLIST := 0.001 ]
    )
  |
  ( numerical_gradients
    [ method_source ]
    [ ( dakota
        [ ignore_bounds ]
        )
    | vendor ]
    [ interval_type ]
    [ forward
    | central ]
    [ fd_step_size ALIAS fd_gradient_step_size REALLIST := 0.001 ]
    )
  no_hessians
  |
  ( numerical_hessians
    [ fd_step_size ALIAS fd_hessian_step_size REALLIST ]
    [ forward
    | central ]
```

```
    )
  |
  ( quasi_hessians
    ( bfgs
      [ damped ]
      )
    | sr1
    )
  | analytic_hessians
  |
  ( mixed_hessians
    [ id_numerical_hessians INTEGERLIST
      [ fd_step_size ALIAS fd_hessian_step_size REALLIST ]
      ]
    [ forward
    | central ]
    [ id_quasi_hessians INTEGERLIST
      ( bfgs
        [ damped ]
        )
      | sr1
      ]
    [ id_analytic_hessians INTEGERLIST ]
    )
```