SANDIA REPORT

SAND2004-6574
Unlimited Release
Printed January 4, 2005

Sensitivity Technologies for Large Scale Simulation

B. G. van Bloemen Waanders, R. A. Bartlett, S. S. Collis,
E. R. Keiter, C. C. Ober, T. M. Smith (Sandia National Laboratories)
V. Akcelik, O. Ghattas, J. C. Hill (Carnegie Mellon University)
M. Berggren (University of Uppsala),
M. Heinkenschloss (Rice Unversity),
L. C. Wilcox (Brown University)

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

SAND2004-6574
Unlimited Release
Printed January 4, 2005

Sensitivity Technologies for Large Scale Simulation

B. G. van Bloemen Waanders, R. B. Bartlett, S. S. Collis,
E. R. Keiter, C.C. Ober, T. M. Smith
(Sandia National Laboratories [1])
V. Akcelik, O. Ghattas, J. C. Hill (Carnegie Mellon University),
M. Berggren (University of UppSala),
M. Heinkenschloss (Rice University),
L. C. Wilcox (Brown University)

[1]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Acknowledgement

The authors would like to thank the LDRD office for providing funding for the majority of this work, but also acknowledge CSRF for providing funding for time domain decomposition and error estimation research. The authors would like to thank the following individuals for valuable discussion and contributions: Larry Biegler, George Biros, Paul Boggs, Matt Gammel, Mike Heroux, Jan Hesthaven, Russel Hooper, Kevin Long, Andy Salinger, John Shadid, and Tim Warburton.

The format of this report is based on information found in [149].

Contents

Executive Summary xvi

1 Introduction **1**

1.1 Mechanics of sensitivity calculations 2

1.2 Direct and Adjoint based inversion of contamination events 4

1.3 Automatic differentiation and linear algebra interfaces 6

1.4 Transient simulations 7

1.5 Adjoints for error estimation, uncertainty quantification 8

1.6 Outline of the report 9

2 Mathematical overview of sensitivity computations **10**

2.1 Steady-state sensitivities 11

2.1.1 Implicit state solution and constraint sensitivities 12

2.1.2 Reduced gradients 13

2.2 Transient sensitivities 14

2.2.1 Implicit DAE solutions and reduced auxiliary response functions 15

2.2.2 Reduced auxiliary response function derivatives 15

2.2.3 Transient direct sensitivities 16

2.2.4 Transient adjoint sensitivities 16

3 Offline/Online QP Solver Stratgies for Real Time Inversion **18**

3.1 Introduction 18

3.1.1 Statement of the Problem: General QP Formulation 19

3.2 Reduced QP problem 20
3.3 Decomposition of Reduced QP into Offline and Online Subproblems 21
3.3.1 Offline Subproblem 22
3.3.1.1 Computing Compressed Sensitivities using the Forward Approach 22
3.3.1.2 Computing Compressed Sensitivities using the Adjoint Approach 23
3.3.2 Online Subproblem 23
3.4 Example Application: Source Inversion for Steady-State Convection-Diffusion in a 3D Airport Terminal 24
3.5 Summary and Conclusions 27
3.6 Recommendations and Future Work 28

4 Large Scale Inversion of a Contaminant in a Regional Model using Convection-Diffusion 29

4.1 Introduction 29
4.2 Convection-Diffusion and First Order Optimality Conditions 30
4.2.1 Contaminant Transport by Convection-Diffusion 31
4.2.2 First Order Optimality Conditions and Inversion Equations 32
4.3 Implementation and Numerical Results 34
4.3.1 Transport of a Source Contaminant 34
4.3.2 Source Inversion Results 34
4.3.2.1 Number of sensors 35
4.3.2.2 Regularization Parameter 37
4.4 Conclusions 37

5 Automatic Differentiation 39

5.1 Introduction 39
5.2 Background 40
5.3 Hybrid symbolic/AD approaches for element-based assembly models 45
5.3.1 Global derivative computations 46
5.3.2 Sparse Jacobian matrix and Jacobian-vector assembly 47

5.3.3 Precomputed-storage and storage-free approaches 48
5.3.4 Element derivative computations 48
5.3.5 Levels of hybrid symbolic/AD differentiation 51
5.4 Finite-volume discretization example 52
5.4.1 The basic residual function and level-0 hybrid symbolic/AD differentiation 52
5.4.2 Two-loop edge-based residual assembly 53
5.4.3 State Jacobian-vector product assembly 54
5.4.4 Results for various Jacobian-vector product assembly methods 59
5.5 Conclusions 62

6 Comparison of Operators for Newton-Krylov Method for Solving Compressible Flows on Unstructured Meshes 64

6.1 Introduction 64
6.2 Governing Equations 65
6.3 Algorithmic Formulation 66
6.4 Steady-State Solution Strategies 69
6.4.1 Newton-based Strategies 69
6.5 Transient 70
6.6 Pseudo-Transient 70
6.7 Homotopy 71
6.8 Newton-Krylov Based Solver 72
6.9 Matrix-Free Operator (MF) 72
6.10 Finite-Difference Coloring Operator (FDC) 73
6.11 Approximate Analytic Operator (AD) 73
6.12 Preconditioning 74

7 Towards Shape Optimization 86

7.1 Introduction 86
7.2 Residual and state computations 89

7.2.1 The Euler equations 89
7.2.2 A node-centered finite-volume discretization of MUSCL type 90
7.2.3 State computations 93
7.2.3.1 Gradient and residual 93
7.2.3.2 Forward state Jacobian-vector product 94
7.2.3.3 Adjoint state Jacobian-vector product 97
7.2.4 Adjoint geometric Jacobian-vector product 99
7.2.5 Finite-volume dual-mesh algorithm 99
7.2.5.1 Adjoint dual-mesh Jacobian-vector product 100
7.3 Summary 101

8 The Development of Abstract Numerical Algorithms and Interfacing to Linear Algebra Libraries and Applications. 102

8.1 Introduction 102
8.2 Classification of linear algebra interfaces 103
8.3 Vectors in Numerical Software and Challenges in Developing Abstract Interfaces 105
8.3.1 Variety of vector operations needed 105
8.3.2 Current approaches to developing interfaces for vectors and vector operations 106
8.4 Vector Reduction/Transformation Operators 107
8.4.1 Introduction to vector reduction/transformation operations 107
8.4.2 An object-oriented design for reduction and transformation operators 108
8.4.3 Computational results for reduction/transformation operators 111
8.5 TSFCore 112
8.5.1 Basic requirements for TSFCore 112
8.5.2 Overview of TSFCore 113
8.5.3 Some TSFCore details and examples 115
8.5.3.1 A motivating example sub-ANA : Compact limited-memory BFGS 116
8.5.3.2 `MultiVector` 117
8.6 Conclusions 121

9 Instrusive Optimization of a PDE Device Model using Partial Nonlinear Elimination Newton Based Solver 124

9.1 Introduction 124

9.2 Partial Nonlinear Elmination in a Newton-based Nonlinear Equation Solver 125

9.2.1 Two-level Newton methods 126

9.2.2 Full Newton step computation with two-level Newton globalization 127

9.3 Electrical Simulation Example 129

9.4 Problem Definition 129

9.4.1 Device Simulation 129

9.4.1.1 Poisson equation 130

9.4.1.2 Species continuity equations 130

9.4.1.3 Initial Condition 131

9.4.1.4 Boundary Conditions 132

9.4.2 Coupling Equations 133

9.5 Solution Methods 133

9.5.1 Similarities 134

9.5.2 Differences 134

9.5.3 Coupling Algorithms 135

9.5.3.1 Tight Coupling: Full Newton 135

9.5.3.2 Loose Coupling: Two-Level Newton 135

9.5.3.3 Loose Coupling: Modified Two-Level Newton 136

9.5.3.4 Loose Coupling: Two-Level Newton with Continuation 137

9.5.3.5 Voltage Limiting with Two-Level Newton 138

9.6 Conclusions 139

10 Time-Domain Decomposition Iterative Methods for the Solution of PDE Constrained Optimization Problems 140

10.1 Introduction 140

10.2 Temporal decomposition of the problem 142

10.3 The Lagrange-Newton-SQP Method 143
10.4 Iterative solution of the Newton system 150
10.4.1 The Newton system 150
10.4.2 Gauss–Seidel Iterations 152
10.5 Application to the Optimal Control of Burgers Equation 153
10.6 Application to the Dirichlet Control of the Two-Dimensional Heat Equation 156
10.7 Conclusions 157

11 Adjoint-based *a posteriori* error analysis **159**
11.1 Introduction 159
11.2 Formulation: An Example from Linear Algebra 160
11.3 Formulation: An Example PDE 161
11.4 Continuous Adjoint Formulation 161
11.5 Local Discontinuous Galerkin Discretization 162
11.6 Computation of Error Estimates 163
11.6.1 hp-adaption 164
11.6.2 Refinement Strategies 165
11.6.3 Smoothness Indicator 165
11.7 Implementation 165
11.8 Numerical Results 166
11.8.1 Average Functional 166
11.8.2 Gaussian Weighted Average Functional 166
11.8.3 Boundary Flux Functional 166
11.8.4 L-Shaped Domain 170
11.9 Conclusion 171

12 Continuous versus Discrete Adjoints **174**
12.1 Formulation 174
12.2 Results 175

12.3 Time Regularization 176
12.4 Summary 178

References **193**

Appendix

Tensor-valued matrices, Jacobians and eigendecompositioins 193
Roe-type dissipation 198
Derivation of partial derivatives of the reconstruction 200
Implementation of "assignment to scalar" `RTOp` transformation operator. 202
Automatic generation of RTOp subclasses in C 204
Example transformation operator 204
Example reduction operator 206

List of Figures

3.1 Two-D cross-section of the 3D airport terminal model. Also shown is an example flow pattern and contaminate release simulation. 25

3.2 Plots for a 2D cross-section of the top bottom floor in the 3D airport terminal model (shown in Figure 3.1) of input source and inverted source for $n_u = 100$ discretized inversion parameters. 26

4.1 Transport of chemical over an 8 Gaussian synthetic landscape at initial time, 30, 60, and 90 timesteps. 35

4.2 Inversion results: target concentration (top left), 6x6x6 sensor array inversion (top right), 11x11x11 sensor array inversion (bottom left), and 21x21x21 sensor array array (bottom right). 36

5.1 Example of a simple function $f(x) \in \mathbf{R}^4 \rightarrow \mathbf{R}$ with nonlinear operations up front followed by all linear operations. 45

6.1 Convergence history vs. cpu time for the first-order simulations. 81

6.2 Convergence history vs. cpu time for the second-order simulations. 81

6.3 Convergence history vs. # nonlinear iterations for inviscid transonic flow over a NACA 0012 airfoil. 82

6.4 Convergence history vs. # nonlinear iterations for the supersonic inviscid flow over a NACA 0012 airfoil with Ma=2. 82

6.5 Convergence history vs. # nonlinear iterations for laminar flow over NACA 0012 airfoil with $\alpha = 0$ deg. 83

6.6 Convergence history vs. # nonlinear iterations for laminar flow over NACA 0012 airfoil with $\alpha = 10$ deg. 83

6.7 Convergence history of the force coefficients vs. # nonlinear iterations. 84

6.8 Convergence history vs. # nonlinear iterations for turbultent flow over NACA 0012 airfoil. . . 84

6.9 Convergence history vs. # nonlinear iterations of the lift coefficient. 85

6.10 Comparison of surface pressure coefficient to experimental data (symbols) found in AGARD, 1979. 85

7.1 A control volume having a nonzero intersection with a solid wall Γ_w. 90

7.2 Control volume norms in 2D 91

7.3 Dual mesh algorithm in 3D 101

8.1 UML [42] class diagram : Interfaces between abstract numerical algorithm (ANA), linear algebra library (LAL), and application (APP) software. 104

8.2 UML class diagram : vector reduction/transformation operators 109

8.3 UML interaction diagram : Applying a RTOp operator for an out-of-core vector 110

8.4 UML collaboration diagram : Applying a RTOp operator for a distributed parallel vector . . . 110

8.5 Ratio of total process CPU times for using six primitive operations versus the all-at-once operator for the operation in (8.6) (number of processes = 128, number elements per process = 50, 500, 5000, 50000 and `num_axpys = 1 ... 400`) 112

8.6 Ratio of total process CPU times for using five primitive reductions versus the all-at-once operator for the operation in (8.2) and (8.5). The times for the primitive operation approaches with cached temporary vectors and dynamically allocated temporary vectors are both given. 113

8.7 UML class diagram : Major components of the TSF interface to linear algebra 115

8.8 A compact limited-memory representation of the inverse of a BFGS matrix. 116

8.9 Carton of disturbed and locally replicated multi-vectors which are used in the example compact LBFGS sub-ANA when run in SPMD mode on four processors. The process boundaries are shown as dotted lines. The numbers for rows and columns of each multi-vector are also shown. 119

8.10 Carton of distributed-memory matrix-matrix product (i.e. block dot product) $T_1 = S^T V$ run in SPMD mode on four processors. This operation first performs local matrix-matrix multiplication with the entries of S^T and V on each processor using level-3 BLAS and then a global reduction summation operation is performed (using MPI) to produce T_1 which is returned to all of the processors. 120

8.11 Carton of local matrix-matrix product $T_3 = Q_{ss} T_1$ involving locally replicated multi-vectors run in SPMD mode on four processors. In SPMD mode this operation involves no processor-to-processor communication at all. 121

8.12 Carton of distributed-local matrix-matrix product $U = ST_3$ involving both distributed and locally multi-vectors run in SPMD mode on four processors. To perform this operation, the local elements in the distributed multi-vector S are multiplied with the locally replicated multi-vector T_3 and the result of the matrix-matrix product are set to the local elements of the distributed multi-vector U. In SPMD mode this operation requires no processor-to-processor communication at all. 122

10.1 The variables $\mathbf{x}$ and the operator $\mathbf{G}$ 145

10.2 The operator vector product $\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x}$. (Here, $y_i = y_i(\cdot; \overline{y}_i, \overline{u}_i)$ and $\lambda_i = \lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ are the solution of the time subinterval state equation (10.4) and the time subinterval adjoint equation (10.8), respectively, and $z_i = z_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i)$ and $\eta_i = \eta_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1})$ are the solution of the time subinterval linearized state equation (10.11) and the time subinterval linearized adjoint equation (10.12), respectively). 147

10.3 Standard integration versus Multiple Shooting approach 149

10.4 Convergence history of GMRES (Example 2, $N_t = 4, 8, 16$) 157

11.1 Element meshes for the average functional test and the Gaussian weighted average functional test. 166

11.2 Contour plots of the primal and dual solutions for the average functional test case on the initial mesh. (a) Primal solution for $p = 5$ and $u = \sin(\pi x)\sin(\pi y)$. (b) Dual solution $p = 7$ and $J(w) = \int_\Omega w\, dx$ 167

11.3 Contour plots of the primal and dual solutions for the Gaussian weighted average functional test case on the initial mesh. (a) Primal solution for $p = 5$ and $u = \sin(\pi x)\sin(\pi y)$. (b) Dual solution for $p = 7$ and $J(w) = \int_\Omega \exp\left(-(x+\frac{1}{2})^2 - (y+\frac{1}{2})^2\right) w\, dx$. 167

11.4 Meshes for the boundary flux functional test where all of the elements are refined between each level. (a) starting mesh, (b) refinement level 1, (c) refinement level 2, (d) refinement level 3. 170

11.5 Contour plots of the primal and dual solutions for the boundary flux functional test case on the initial mesh. (a) Primal solution for $p = 4$ and $u = \sin(\pi x)\sin(\pi y)$. (b) Dual solution $p = 6$ and $J(w) = \int_{\partial\Omega} \kappa \frac{\partial w}{\partial n}\, ds$ where $\kappa(x, y) = 1$ on the interior hole boundary. 171

11.6 (a) Contours of the primal solution on the staring mesh defined in polar coordinates as $u(r, \theta) = r^{2/3}\sin(\frac{2\pi}{3}\theta)$. (b) Contours of the dual solution for $J(w) = \int_\Omega uw\ dx$. (c) The final mesh where different colors represent the polynomial-order of the element $p + 1$. We have used the global TOL refinement strategy with TOL $= 10^{-7}$ and $c_f = 0.1$. (d) Here is a contour plot of the error estimate η_K. Most of the error is concentrated at the corner. . . . 172

11.7 The error in the functional, $J(w) = \int_\Omega uw\, dx$ as a function of the square root of the degrees of freedom. We see that for just h-adaption the best choice is $p = 1$ and higher order interpolation does not help — it actually hurts the performance. We also see that for both refinement strategies we get exponential convergence in the functional of interest. If we put a straight-line fit through ◇ it has a slope of 2.8. 173

12.1 Convergence of nonlinear CG for Robin boundary control of the heat equation with and without time-regularization. (Left) without time-regularization: —— DA,- - - - CA(BC), —·— CA($p-1$). (Right) with time-regularization $\alpha_t = 0.1$: —— DA, - - - - CA(BC), —·— CA(TR). 177

12.2 Comparison of states and controls for Robin boundary control of the heat equation. (Left) State: ····· Target, —— DA, - - - - CA(BC), —·— CA($p-1$). (Right) Control: —— DA, - - - - CA(BC), —·— CA($p-1$) 177

12.3 Effect of time-regularization ($\alpha_t = 0.1$) on Robin boundary control for the heat equation (results after 6 iterations). (Right) State: ····· Target, —— DA, - - - - CA(BC), —·— CA(TR). (Left) Control: —— DA, - - - - CA(BC), —·— CA(TR). 178

Executive Summary

Sensitivity analysis is critically important to numerous analysis algorithms, including large scale optimization, uncertainty quantification,reduced order modeling, and error estimation. Our research focused on developing tools, algorithms and standard interfaces to facilitate the implementation of sensitivity type analysis into existing code and equally important, the work was focused on ways to increase the visibility of sensitivity analysis. We attempt to accomplish the first objective through the development of hybrid automatic differentiation tools, standard linear algebra interfaces for numerical algorithms, time domain decomposition algorithms and two level Newton methods. We atttempt to accomplish the second goal by presenting the results of several case studies in which direct sensitivities and adjoint methods have been effectively applied, in addition to an investigation of h-p adaptivity using adjoint based a posteriori error estimation.

A mathematical overview is provided of direct sensitivities and adjoint methods for both steady state and transient simulations. Two case studies are presented to demonstrate the utility of these methods. A direct sensitivity method is implemented to solve a source inversion problem for steady state internal flows subject to convection diffusion. Real time performance is achieved using novel decomposition into offline and online calculations. Adjoint methods are used to reconstruct initial conditions of a contamination event in an external flow. We demonstrate an adjoint based transient solution. In addition, we investigated time domain decomposition algorithms in an attempt to improve the efficiency of transient simulations.

Because derivative calculations are at the root of sensitivity calculations, we have developed hybrid automatic differentiation methods and implemented this approach for shape optimization for gas dynamics using the Euler equations. The hybrid automatic differentiation method was applied to a first order approximation of the Euler equations and used as a preconditioner. In comparison to other methods, the AD preconditioner showed better convergence behavior. Our ultimate target is to perform shape optimization and hp adaptivity using adjoint formulations in the Premo compressible fluid flow simulator.

A mathematical formulation for mixed-level simulation algorithms has been developed where different physics interact at potentially different spatial resolutions in a single domain. To minimize the implementation effort, explicit solution methods can be considered, however, implicit methods are preferred if computational efficiency is of high priority. We present the use of a partial elimination nonlinear solver technique to solve these mixed level problems and show how these formulation are closely coupled to intrusive optimization approaches and sensitivity analyses.

Production codes are typically not designed for sensitivity analysis or large scale optimization. The implementation of our optimization libraries into multiple production simulation codes in which each code has their own linear algebra interface becomes an intractable problem. In an attempt to streamline this task, we have developed a standard interface between the numerical algorithm (such as optimization) and the underlying linear algebra. These interfaces (TSFCore and TSFCoreNonlin) have been adopted by the Trilinos framework and the goal is to promote the use of these interfaces especially with new developments.

Finally, an adjoint based a posteriori error estimator has been developed for discontinuous Galerkin discretications of Poisson's equation. The goal is to investigate other ways to leverage the adjoint calculations and we show how the convergence of the forward problem can be improved by adapting the grid using adjoint-based error estimates. Error estimation is usually conducted with continuous adjoints but if discrete adjoints are available it may be possible to reuse the discrete version for error estimation. We investigate the advantages and disadvantages of continuous and discrete adjoints through a simple example.

Chapter 1

Introduction

Sensitivity analysis of large-scale systems is important in many engineering and scientific applications, such as chemical, mechanical and electrical engineering and economics. Sensitivity analysis generates essential information for parameter estimation, optimization, control, model simplification and experimental design. Consequently, algorithms that perform such an analysis in an efficient and rapid manner are invaluable to researchers in many fields.

Considerable amount of work has been conducted in the technical community that involves sensitivity type calculations. Some examples include shape optimization in computational fluid dynamics, parameter identification in the climate forecasting community, performance identification for electrical circuits, image reconstruction, and uncertainty quantification in groundwater hydrology. The aerodynamics community have actively made use of sensitivity analysis. In particular, adjoint formulations have been developed to solve optimization problems in three dimensions from smaller components to entire aircraft design [128, 129, 181, 209, 180]. The primary purpose of these adjoint calculations has been to improve the shape of aircraft components and in some case entire aircraft structures by minimizing the drag or maximizing the lift with respect to mesh movements. The use of direct sensitivities and adjoint methods for atmospheric and oceanic systems was first described by Marchuk et al. [156] and Konterev [137]. Kaminski et al. [132] present the implementation of adjoint methods through the use of an reverse automatic differentiation compiler for a three dimensional atmospheric transport of a passive scalar. Daescu et al. [67] discuss the implementation of adjoint sensitivities into an atmospheric chemical transport code using automatic differentiation methods (source transformation). They attempt inversion for global models through a qualitative review of sensitivity contours of the source with respect to various chemicals. Daescu et al. [68] demonstrate a four dimensional data assimilation to obtain an optimal estimate of the initial state of a dynamical system solving a large unconstrained optimization problem. Finally, Sandu et al. [194] discuss adjoint sensitivities for the gradient calculations using the Kinetic preprocessor (KPP) software tool that facilitates direct and adjoint sensitivities for the transport of chemical species. Sensitivity analysis has been used extensively in electrical simulation. In fact, one of first adjoint method developments was initiated in this community [74]. Numerous work has been done over the years in particular to assess the performance and robustness of electrical circuits [203, 82, 57, 173]. The primary use for sensitivity analysis is to understand and improve the performance of circuits. In medical imaging, inverse problems are solved using adjoint methods in tomography [78] for image reconstruction. Likewise in geophysics, adjoint implementations are used to invert for material properties [205, 18, 171, 172, 171, 8].

Although the mathematics of sensitivity calculations are well understood, the implementation of

sensitivities in support of optimization and other analysis tools in large production codes has not been fully addressed. Part of the reason why most existing production codes face a difficult implementation task is because the original design concentrated exclusively on the forward simulation and did not consider any analysis capabilities, including fundamental building blocks such as sensitivity calculations. The lack of an appropriate software structure and the implementation of algorithms that are not suitable towards sensitivities cause inefficient future implementation issues, especially if large numbers of sensitivity parameters are of interest. Some of the specific implementation deficiencies, include poorly designed software interfaces, explicit solutions, inexactness of the state Jacobians, matrix-free methods, explicit solvers, non-differentiable methods, non-smooth behaving state solutions and complicating issues with transient simulations. Although we do not address all these issues, most of them can be circumvented by making sensitivity analysis a high priority as part of the initial design. Retrofitting existing codes depends on the underlying physics algorithms, linear algebra structure and how much implementation effort can be afforded. One of the motivations of this work was to develop techniques, methods and software tools to help promote the consideration of sensitivities as a priority equivalent to the development of the underlying physics so that costly future refactoring can be avoided by concentrating on 1) mechanics of calculating direct sensitivities and adjoints, 2) software interfaces including automatic differentiation methods, 3) improvements of sensitivity related calculations for transient methods, 4) continuous adjoints, and 5) an investigation of adjoint-based error estimation to demonstrate the potential dual use of sensitivity information.

1.1 Mechanics of sensitivity calculations

The use of sensitivity analysis appeared as early as the 1940's through perturbation and variational approaches [93, 217]. Initial forward and adjoint formalisms can be traced back to Levin et at [144], Roussopolos [186], and Stacey [200]. This work concentrates on the formalism of sensitivity calculations through variational calculus where the set of nonlinear PDE equations is abstracted away [48] and obviously issues related to implementation to the underlying PDE-based simulation are not addressed. Additionally, in all the application studies that have been conducted over the last couple of decades, the implementation issues are not addressed and are specific to the simulation code. It is exactly this interface that is a primary focus of our research. We focus on development of tools and methods that can be reused and we focus on various studies to help motivate the use of direct sensitivities and adjoint methods.

At the center of all these problems is a set of linear or nonlinear state equations that define the *forward problem*. These state equations may be steady-state or transient in nature. In many of the problems considered here, these equations arise from a spatial discretization of a PDE over some geometrical domain. The addition of analysis methods applied to these simulation problems require some subset of input variables of interest $u \in \mathbf{R}^{n_u}$. Here the input variables of interest u will be referred to as *auxiliary* variables in the most general setting. Once the simulation problem is solved for some value of u, one or more auxiliary response functions are defined which inform the analysis method of an important observation or performance metric of the simulation problem. These auxiliary response functions are generally some type of reduction over the solved state variables. Examples of these response functions of interest might be a maximum stress, lift of a wing, drag over a wing, weight of a system, or flux across some surface. These response functions are typically what drive a design or evaluation study of a system.

Through a simulation solve, the mapping of auxiliary variables u to response functions can be represented

in the general form

$$u \rightarrow \hat{g}(u) \tag{1.1}$$

where

$u \in \mathbf{R}^{n_u}$ are the auxiliary variables or parameters, and
$\hat{g}(u) \in \mathbf{R}^{n_u} \rightarrow \mathbf{R}^{n_g}$ are the reduced auxiliary response functions.

An evaluation of $\hat{g}(u)$ implies a solution of the underlying simulation problem.

All of the analysis methods considered here not only need to compute the auxiliary functions $\hat{g}(u)$ themselves, but they also require the first derivatives of these functions in the form of the derivative object

$$\frac{\partial \hat{g}}{\partial u} \in \mathbf{R}^{n_g \times n_u} \tag{1.2}$$

which is often times referred to as the *reduced gradient*. Actually, to be precise, the reduced gradient of the response function $\hat{g}_k(u)$ (where $k = 1 \ldots n_g$) is $(\partial \hat{g}_k / \partial u)^T$. Therefore, each row of $\partial \hat{g} / \partial u$ is a reduced gradient.

Note that the first derivatives $\partial \hat{g} / \partial u$ in (1.2) are typically approximated using finite differences. Assuming forward finite-differences are used, this requires n_u full simulations to be solved, one solve for each perturbation in u. In addition, the resulting derivatives are often significantly inaccurate due to truncation and round-off errors which are well known problems for finite-difference methods. The primary advantage of these types of *black-box* finite-difference approaches is that they require almost no extra code development effort to implement. The *black-box* term refers to treating the target function as an unknown entity. This report focuses on approaches for computing (1.2) that are more efficient and/or more accurate than standard black-box finite-difference approaches.

All of the approaches that are described in this report for computing reduced gradients of the form $\partial \hat{g} / \partial u$ are computed using one of two general classes of approaches, *direct* or *adjoint*, where:

- *Direct sensitivity approaches* compute $\partial \hat{g} / \partial u$ in $O(n_u)$ flops[1], independent of n_g, and
- *Adjoint sensitivity approaches* compute $\partial \hat{g} / \partial u$ in $O(n_g)$ flops, independent of n_u.

Therefore, in general, when there are many more auxiliary variables n_u than response functions n_g (i.e. $n_u \gg n_g$) – which is often the case for applications and analysis methods – adjoint approaches are more efficient. Alternatively, if there are many more response functions n_g than auxiliary variables n_u, direct approaches are generally more efficient.

Sensitivity methods for PDE-constrained problems can also be further classified into *discrete* and *continuous* approaches. Discrete approaches start from a discretization of the forward problem and are based on strictly finite-dimensional algebraic manipulations. An alternative strategy is to derive the continuous adjoint, or dual problem. The dual problem can be derived based on the Euler-Lagrange identity. There are a number of reasons for considering a continuous approach: it is the standard implementation strategy for *a posteriori* error estimation, it may be an easier implementation option if transposed Jacobians are not available, and additional interpretation capabilities are available. This adjoint

[1] floating point operations

formulation can be implemented in the same way as the forward simulation and, in addition, can reuse the same solution procedures as the forward problem. The continuous adjoint can be used as an additional interpretation tool where the adjoint solution variables, also known as the Lagrange variables in an optimization method, can be interpreted as a sensitivity of the auxiliary response functional with respect to perturbations in the state equations. A detailed comparison of discrete and continuous adjoints was conducted by Nadarajah et al. [168]. It should be noted that there are some disadvantages to the continuous adjoint, including a refined mesh is required to achieve an accurate reduced-gradient (with respect to the discrete forward problem), boundary conditions need to handled with care, and as much as twice as much code is required in comparison to the discrete implementation.

Discrete direct sensitivity methods require solving multiple linear systems involving the state Jacobian (related to the underlying state simulation equations) in which each right-hand-side column is dependent on a different auxiliary variable u_i, for $i = 1 \dots n_u$. For an implicit solution procedure (such as Newton's method), the forward simulation depends on the solution of linear systems with this same state Jacobian; therefore, the infrastructure for solving the forward problem can be often be used to solve for direct sensitivities. The obvious limitation is that for larger numbers of auxiliary variables n_u, many state solves are necessary which could become intractable. Continuous direct sensitivity methods, which are applicable to PDE-constrained problems, need not require linear system solves but are still require $O(n_u)$ flops over a single simulation.

To address the issue of computational efficiency for large numbers of auxiliary variables, the adjoint method can be considered. By a reordering of the computations, a discrete adjoint method can compute the required reduced gradients using only $O(n_g)$ solves (independent of n_u) with the adjoint of the state Jacobian. If the state Jacobian matrix is explicitly formed, there is little more implementation effort required in comparison to the direct case. Computing adjoints in the transient case involves further complications.

Adjoint based *a posteriori* error analysis for finite element discretizations requires the solution of the adjoint on a different solution space, which is efficiently achieved through the implementation of a continuous adjoint. A discrete adjoint could also be used, however this means that the forward problem needs to be solved for two different mesh resolutions, in addition to solving the adjoint equation (albeit one solve). For finite-volume discretizations, where Galerkin orthogonality is not an issue, then the continuous adjoint may not offer any computational efficiencies. If optimization and adaptivity both need to be solved for a finite element discretization, then the discrete adjoint may have to be solved on two different meshes. Other methods have been investigated to make use of the discrete adjoint, such as reconstruction procedures [210], that avoid the computational expense of an additional solve. However, it is not clear how effective these reconstruction methods are and this is an area for future research.

1.2 Direct and Adjoint based inversion of contamination events

Significant work has been conducted in the technical community using sensitivity type calculations. In this report, we apply sensitivity calculations to a variety of physics ranging from simple heat equations and convection diffusion to the more complex physics such as the Euler equations to simulate gas dynamics. We differentiate this effort from previous work by exploiting sensitivity calculations for time domain decomposition, adaptivity using a posteriori error estimation, offline calculation for real time performance, and implicit coupling of multi-physics problems. The primary goal is to motivate the use of direct sensitivity and adjoint methods especially in new developments so that a variety of analysis tools, in

particular optimization, can be easily implemented. Several application studies were conducted as part of this work to demonstrate the utility of sensitivity type calculations.

We consider inversion problems to reconstruct source terms and initial conditions in the event of a contamination event in both internal and external flow. We demonstrate the use of direct and adjoint sensitivities on a source inversion problem subject to convection-diffusion. The application context is related to developing numerical algorithms to combat intentional contamination events in certain large internal spaces (i.e. airports, enclosed sports stadiums, etc) and external spaces (i.e. regional models, urban canyons, etc). Assuming sensors are able to measure concentration values, a nonlinear least-squares formulation can determine the original location, magnitude, and characteristics of the event. This then enables a forward simulation to accurately predict the transport of the chemical for an appropriate mitigation strategy. For internal spaces we have designed an offline/online solution technique separating critical calculations so that we can achieve the goal of real time inversion capabilities.

The approach consists of decomposing repeated solution of a class of Quadratic Programming (QP) problems into offline and online subproblems. The general formulation for the class of QPs we consider encompasses several different types of inversion and least-squares problems with both steady-state and transient state models. Our method exploits the properties that each solved QP has the same linear state constraint matrices, and has a diagonal objective function Hessian with the same nonzero structure along the diagonal. We exploit these properties by splitting the computation of the solution of the QP into offline and online components. This approach allows for the "real time" solution of a relatively broad class of these QPs using modest computing resources (at least for the online computation).

A similar inversion problem was investigated to demonstrate the utility of adjoint methods by considering a transient inversion problem subject to convection-diffusion for an external flow problem. We are interested in an inversion of initial conditions for a contamination event in an outdoor regional model and we assume that inversion parameters exist everywhere in the spatial domain. The constrained optimization problem is reformulated as an unconstrained problem by eliminating the transient states using a time integrator. The solution process consists of first taking variations of this functional and then sequentially solving for the states, adjoints and the inversion parameters. The goal of this development has been on computational efficiency by using structured grids and matrix-free implementations. We are able to solve the inversion problem for a reasonably large dataset relatively efficiently. However, to solve highly resolved regional models, we need an efficient preconditioning strategy for the solution of the reduced Hessian. We have developed the basic interface for a multigrid preconditioning approach and at the writing of this report, we are in the process of performing various numerical tests.

We have applied this inversion problem on a synthetic landscape. Our numerical experiment consists of collecting concentration measurements from a sparse set of locations from a forward simulation that is initially used to predict the transport of chemical with known initial conditions throughout a domain. These concentration measurements are then considered target values in a nonlinear least-squares formulation in which the goal is to match these concentration values with predicted values by manipulating the initial conditions. The set of unknown inversion parameters consists of values at every point of the spatial domain. We are able to invert for the initial conditions with a reasonably small set of sensors.

1.3 Automatic differentiation and linear algebra interfaces

Sensitivity information is dependent on the fundamental derivative calculation and, the accuracy is very important especially for analysis methods such as optimization. Analytic values are often time-consuming to derive and difficult to implement. Finite-difference (FD) methods are dependent on the perturbation step size which are problematic to set a priori. Standard automatic differentiation (AD) tools that use source transformation are sensitive to the simulation code implementation and result in a cumbersome development environment. We have developed both non-invasive and invasive hybrid AD strategies that pre-compute and store Jacobian sub-matrices and then assemble Jacobian-vector products at the mesh-object level. This approach may result in computations that are nearly five times faster than one-sided finite differences. In addition, these derivative computations are accurate to machine precision. We also found that low storage on-the-fly black-box uses of AD at the mesh-object level can result in derivative computations that are very competitive in speed with FD methods and require little knowledge of the actual mesh-object-level computations. However, we show how exploiting the structure of the computations at a lower level and combining symbolically derived and coded derivatives with AD resulted in significant improvements in CPU time and storage.

Furthermore hybrid symbolic and AD approaches are well suited to discretization methods for PDEs due to the hierarchal structure of such methods. Hybrid symbolic AD methods can result in very accurate and affordable derivative computations without having to rewrite an entire code but instead focus differentiation tools on lower-level computations based on mesh objects (such as elements, edges, faces, etc.). Additional interface issues need to be addressed if large scale optimization methods are to be interfaced. A common observation of large production codes is that they are not built with convenient underlying vector and matrix implementations. A portion of this report is devoted to the development of "standard" implementations for abstract numerical algorithms and linear algebra. Even though this is not central to sensitivity analysis, it is critical to our ability to efficiently implement sensitivities and optimization algorithms in multiple production codes.

One of our primary hurdles in implementing sensitivities throughout applications at Sandia is the different software implementations that have been used in simulation code. For optimization this interface is very specific and often not recognized in the simulation development community. Although, various framework are currently being developed to provide common services to different physics code developers, a standard interface for abstract numerical algorithms (ANAs), such as optimization, uncertainty quantification, stability analysis, nonlinear solvers, has not been adequately addressed. Ideally, all developers should adhere to a standard linear algebra implementation and perform numerical algorithm development within that standard. Gockenbach et al. has attempted to develop an approach to address this interfacing problem [95]. This is a powerful approach but it has some limitations, some of which are critical to our environment. The most obvious omission of the HCL framework is their serial implementation. This and other smaller reasons, prompted our development of TSFCore which was designed specifically as simple as possible to minimize the overhead of the PDE code developers. However, the interface consists of sufficient flexibility that it can be applied to a variety of numerical algorithms, such as continuation methods, nonlinear solvers, uncertainty quantification methods, and optimization.

TSFCore provides the intersection of all of the functionality required by a variety of abstract numerical algorithms ranging from iterative linear solvers all the way up to optimizers. The foundation of TSFCore described here only covers vector spaces, vectors, multi-vectors and linear operators. By adopting TSFCore as a standard interface layer, interoperability between applications, linear algebra libraries and abstract numerical algorithms in advanced scientific computing environments becomes automatic to a large

extent. An important subcomponent of TSFCore is the vector operator interface. Growing complexities associated with computational environments, application domains, and data mapping place difficult demands on the development of numerical algorithms. The vector operator interface proposed here addresses these issues and allows the development of many types of complex abstract numerical algorithms that are highly flexible and reusable.

Advanced object-oriented design patterns were used to develop the vector operator interface (RTOp) and somewhat predictable numerical experiments demonstrate high efficiency in comparison to using a combination of primitives. Also, simple scalability tests confirm minimal serial overhead for large number of processors. Though the numerical efficiencies are noteworthy, development efficiencies and functionality provide the main advantages of this approach including 1) vector implementation developers need only implement one operation and not a large collection of primitive vector operations, 2) ANA developers can implement vector operations without needing any support from vector implementation maintainers, 3) ANA developers can optimize time consuming vector operations on their own for the platforms they work with, 3) Reduction/transformation operators are more efficient than using primitive operations and temporary vectors, 4) ANA-appropriate vector interfaces that require built-in standard vector operations (i.e. axpy and norms) can use RTOp operators for the default implementations of these operations.

1.4 Transient simulations

Transient simulations with large numbers of design variables are a difficult challenge for optimization. The large design space requires the use of adjoints but this means that the adjoint equations need to be integrated backwards and this also means that the solution history of the forward simulation needs to be saved for each time step. In the case of a linear PDE only the final state needs to be saved, but in the case of a nonlinear PDE, storing the entire state history often becomes intractable.

Transient simulations pose problems when sensitivities of performance functions are required with respect to many design parameters. In this case, adjoints need to be calculated and in the transient case an integration backward in time is necessary for the adjoint solution. At each time point in a discrete adjoint approach, the transpose of the state Jacobian is used to evaluate the adjoint system. The adjoint solution therefore incurs the cost of either storing the state Jacobian at each time step or reconstructing the Jacobian at each time step. In the case of direct sensitivities, a linear solve at each time step is required per design variable. Although neither the storage nor the reconstruction cost are incurred for the direct method, only a small number of design parameters can be considered.

A considerable amount of work has been conducted for calculating sensitivities for transient simulations for ordinary differentiable equations (ODEs) and differentiable algebraic systems (DAEs) [155] [145] [146] [50] [51] [91]. [196]. Forward sensitivity analysis cannot handle large numbers of parameters because the computational expense becomes intractable. Whereas the adjoint, or reverse sensitivity analysis method, can handle large number of sensitivity parameter. Conversely, the adjoint method is not well suited to handle high dimensional functions of the solution. Powerful software has been developed by Maly et al. [155], Li et al. [145] and Serban et al. [116]. These methods assume that a set of PDEs are reduced to a set of ODEs or DAEs by discretizing the spatial components (method of lines). One primary reason for not leveraging these software tools is that our focus in time domain decomposition requires explicit control of all aspects of the calculation. Although access to the source code is possible, a complete code refactoring would have been necessary.

Our work on time-domain decomposition extends the ideas of Berggren and Heinkenschloss[33] [108] [109] [110] and attempts to achieve parallel speedup in the time domain. Time-domain decomposition techniques have been applied to the solution of parabolic PDE constrained optimal control problems. The time-domain decomposition approach is motivated by the need to cope with the large storage requirements arising out of the strong coupling in time of states, adjoints and controls.

The time decomposition of the original problem leads to a large scale discrete–time optimal control (DTOC) problem in Hilbert space. The solution of these DTOCs using a Lagrange-Newton-SQP method have been presented. Near a solution of the problem this is equivalent to applying Newton's method to the first-order optimality conditions. The tasks arising in the implementation of the Lagrange-Newton-SQP for the DTOC are related to the solution of the state, linearized state and adjoint equations restricted to time subintervals. The linear system that has to be solved in each iteration of the Lagrange-Newton-SQP method has a block structure, which motivates the application of block Gauss–Seidel (GS) methods for its solution. The forward backward GS method is particularly effective as a preconditioner.

Parallelization of the time-domain decomposition approach is currently under investigation. In particular, the time-domain decomposition approach will be used as an additional layer of parallelization, in addition to existing parallelization of PDE solvers or of the time-subdomain optimal control problems. Potential savings may also result from the need to handle only smaller time subinterval problem. Depending on problem size, this may allow one to reduce the use of slow memory and thus result in significant savings in computing time.

1.5 Adjoints for error estimation, uncertainty quantification

Sensitivities are useful quantities for other analyses such as error analysis, uncertainty quantification, and reduced order modeling. Although, the use of sensitivities for these types analyses are not quite as well developed as for optimization methods. In this section we briefly discuss the use of sensitivities in the context of these analysis tools. The primary goal is to provide additional motivation for the implementation of sensitivity information by recognizing that a variety of analyses can be accomplished. We have chosen to further investigate adjoint-based error estimation to better understand how one might leverage adjoint implementations for optimization.

Adjoints are of particular interest because in the context of optimization, large design-spaces can be considered at the cost of merely a single solve. However, adjoints can also be used for a posteriori error estimation which in turn can be used for adaptivity. Although error estimation is tangential to optimization it provides a general strategy to motivate developers beyond optimization to consider implementing adjoint sensitivities. The implementation of adjoints is clearly 90 percent of the effort towards large-scale optimization in a production code. Error estimation can be conducted with just residual calculations, but it has been shown that adjoint-based *a posteriori* error estimation methods are more accurate, especially since a particular performance functional has to be considered. Consequently adaptivity provides a more accurate and computationally efficient solution.

We have implemented an adjoint-based error estimator for the local discontinuous Galerkin discretization of Poisson's problem. Here the error is measured in some functional of the computed solution. This functional can be a weighted integral of the solution in the domain or a weighted integral of the flux of the solution across a boundary. A smoothness indicator has also been written that is based on the convergence of the coefficients of the projection of the solution on the orthonormal basis for a triangle. Using the error

estimates and the smoothness indicator we have obtained exponential decrease in the error of a weighted integral of a singular solution of Laplace's equation on an L-shaped domain.

For future work we would like to apply adjoint-based error estimates to time dependent problems, nonlinear problems, other discretizations and then to real applications such as the compressible fluid flow problems. This error estimate work is also considered preliminary prototyping for Sandia production codes, such as codes in the SIERRA framework.

Our primary interest in error estimation is to promote the development of adjoints sensitivities and investigate the dual use in both error estimation and optimization. Early work on error estimation was conducted [85]. Significant work has been done in adjoint-based error estimation for hp-adaptivity in PDEs using finite element discretizations
[28, 6, 6, 179, 201, 17, 80, 53, 141, 211, 29, 167, 4, 21, 89, 90, 106, 122]. Additional work has been done on finite-volume methods [221, 130]. For a detailed review of the history and recent developments see [29, 90]. Although residual-based error estimation is computationally more efficient than the adjoint based error estimators, the accuracy is lacking.

Other analysis tools can leverage adjoint systems, namely uncertainty quantification and reduced-order modeling. Uncertainty quantification is becoming an important analysis tool to account for variations in the model or in the data. Like error estimation, the adjoint sensitivity can be leveraged and, in this case, it can be used to predict the statistical relationship between certain parameters through covariance matrices. The use of adjoint methods for uncertainty quantification purposed was beyond the scope of this project, however, the interested reader can find additional details from Woodward et al. [219]. One of the connection of adjoint operators to reduced-order modeling is for computing observability Grammians. Again, this area of analysis is beyond the scope of our investigation, but we refer the interested reader to Willcox et al. for additional details. [218, 187].

1.6 Outline of the report

The remainder of this report is organized as follows. Direct and adjoint sensitivities are applied to large scale inversion problems that depend on convection-diffusion equations. We investigate inversion problems for both internal and external flows. Automatic differentiation is critical to the development of efficient derivative calculations in addition to providing accurate derivatives. Some performance results are presented for a preconditioner that was calculated with an automatic differentiation procedure. We discuss the mathematical formulation associated with shape optimization. The adjoint is derived and the linear objects associated with shape optimization are discussed. A development of standard and convenient linear algebra interfaces are presented. The goal is to avoid repeating the development of the same interfaces for different simulation codes. The mathematical formulation for a two-level Newton method is developed to address the implicit coupling of different physics. We use an electrical simulation code (Xyce) to show how this strategy could be used when PDE-based model devices are coupled into electrical circuits. Time domain decomposition methods for control problems are presented. We make use of multiple shooting techniques and show some potential parallel improvement in the time domain. Also discussed is the use of adjoints for a posteriori error estimation and adaptivity. Although error estimation appears tangential to our optimization focus, both are strongly coupled to the adjoint. An adjoint implementation can potentially solve an optimization problem and provide accurate grid adaptivity strategies to help improve the computational efficiencies. The last chapter looks at some basic comparisons between discrete and continuous adjoints when used in a model transient optimization problem.

Chapter 2

Mathematical overview of sensitivity computations

The goal of this chapter is to provide a concise presentation of the basic sensitivity formulation. We focus on two broad classes of applications: *steady-state* and *transient*. In either case, a set of auxiliary input variables $u \in \mathbf{R}^{n_u}$ is identified such that once specified, allow for a solution of the state equations. These auxiliary variables represent some adjustable data for the state model and may include such things as boundary conditions, reaction rates or other inputs of interest. The common element in steady-state and transient sensitivity problems is that, once they are solved for particular value of the input parameters u, they are used to define one or more reduced auxiliary response (or observation, or performance) functions of the form

$$u \rightarrow \hat{g}(u) \tag{2.1}$$

where

$u \in \mathbf{R}^{n_u}$ are the auxiliary variables or parameters, and
$\hat{g}(u) \in \mathbf{R}^{n_u} \rightarrow \mathbf{R}^{n_g}$ are the reduced auxiliary response functions.

The reduced auxiliary response functions $\hat{g}(u)$ have embedded in them the solution to the underlying simulation problem given the input auxiliary variables u. Auxiliary response functions can represent different types of quantities of interest. For example, in a CFD aerodynamic wing design problem, response functions of interest might include the lift or drag of the wing. In a structural design application, a functional of interest might be a maximum stress point.

We consider efficient and accurate approaches for computing the first derivatives of $\hat{g}(u)$ in the form of the derivative matrix object

$$\frac{\partial \hat{g}}{\partial u} \in \mathbf{R}^{n_g \times n_u}. \tag{2.2}$$

The matrix $\partial \hat{g}/\partial u$ is also know as the *reduced gradient*. Actually, each row $\partial \hat{g}_k/\partial u \in \mathbf{R}^{1 \times n_u}$ is the transpose reduced gradient for the auxiliary function $\hat{g}_k(u)$, for $k = 1 \ldots n_g$. The reduced gradients $\partial \hat{g}/\partial u$ can be computed using either direct or adjoint approaches. In summary:

- *Direct sensitivity approaches* compute $\partial \hat{g}/\partial u$ in $O(n_u)$ flops[1], independent of n_g, and
- *Adjoint sensitivity approaches* compute $\partial \hat{g}/\partial u$ in $O(n_g)$ flops, independent of n_u.

In this chapter, steady-state sensitivities are first presented followed by a discussion of transient sensitivities. This overview pertains to wide ranges of steady-state and transient sensitivity problems for applications including discretized PDEs and network-like problems. With respect to the PDEs, the approaches described here are considered to be discrete approaches with respect to some reference spatial discretization of the underlying PDE state equation. In later chapters, alternative continuous approaches to sensitivity computations are addressed.

2.1 Steady-state sensitivities

Central to all steady-state problems is a set of, possibly nonlinear, state constraints that can be represented as

$$c(y, u) = 0, \tag{2.3}$$

where

$y \in \mathbf{R}^{n_y}$ are the state variables,
$u \in \mathbf{R}^{n_u}$ are the auxiliary parameters, and
$c(y, u) \in \mathbf{R}^{n_y+n_u} \rightarrow \mathbf{R}^{n_y}$ are the state constraint functions.

Equation (2.3) represents a set of nonlinear simulation equations which are also known as the *constraints*. In this notation, $c(y, u)$ is a vector function where each component $c_j(y, u)$, $j = 1 \ldots n_y$, represents a nonlinear scalar function of the variables y and u. The variables y are referred to as the state (or simulation) variables and the variables u are a set of auxiliary variables that take on different meanings depending on the particular analysis method being considered. Note that the number of state y and auxiliary u variables are n_y and n_u respectively. A typical simulation code requires that the user specify the auxiliary variables u and then the square set of equations $c(y, u) = 0$ is solved for y. For a particular application, one or more auxiliary (or observable, or performance) response functions may be of interest and we denote these auxiliary response functions as

$$(y, u) \rightarrow g(y, u), \tag{2.4}$$

where

$g(y, u) \in \mathbf{R}^{n_y+n_u} \rightarrow \mathbf{R}^{n_g}$ are the auxiliary (observable) response functions.

The auxiliary response functions may represent objective and auxiliary constraints in an optimization problem, or just quantities of interest in an error estimation or uncertainty quantification problem. What is common to all of these analysis methods is that the functions $g(y, u)$ must be computable and, for the methods considered here, the reduced first derivatives of $g(y, u)$ are required.

[1] floating point operations

2.1.1 Implicit state solution and constraint sensitivities

The set of nonlinear equations $c(y,u) = 0$ can be solved for y using a variety of methods. Using the solution method it is possible to define an implicit function

$$y = y(u), \text{ s.t. } c(y,u) = 0. \tag{2.5}$$

The definition in (2.5) simply implies that for any reasonable selection of the design variables u, the solution method can compute the states y. Note that evaluating this implicit function requires a complete simulation or "analysis" to be performed by the solution method. The cost of performing the simulation solve may only be an $O(n_y)$ computation in a best-case-scenario, but for many applications the solution complexity is much worse.

We derive the sensitivities of the states y with respect to the auxiliaries u as related through the implicit function (2.5). We begin with a first-order Taylor expansion of $c(y,u)$ about (y_0,u_0) given by

$$c(y,u) = c(y_0,u_0) + \frac{\partial c}{\partial y}\delta y + \frac{\partial c}{\partial u}\delta u + O(||\delta y||^2) + O(||\delta u||^2) \tag{2.6}$$

where

$\frac{\partial c}{\partial y}$ is a square $\mathbf{R}^{n_y}$-by-$\mathbf{R}^{n_y}$ *state Jacobian* matrix evaluated at (y_0,u_0)
$\frac{\partial c}{\partial u}$ is a rectangular $\mathbf{R}^{n_y}$-by-$\mathbf{R}^{n_u}$ *auxiliary Jacobian* matrix evaluated at (y_0,u_0).

In this notation, the state Jacobian matrix $\frac{\partial c}{\partial y}$, for example, is defined element-wise as

$$\left(\frac{\partial c}{\partial y}\right)_{(j,l)} = \frac{\partial c_j}{\partial y_l}\text{ , for } j = 1\ldots n_y,\ l = 1\ldots n_y.$$

If the matrix $\frac{\partial c}{\partial y}$ exists and is nonsingular then the implicit function theorem [170, B.9] states that the function in (2.5) exists and is well defined in a neighborhood of a solution (y_0,u_0). In some applications, the matrix $\frac{\partial c}{\partial y}$ is always nonsingular in regions of interest. In other application areas where the selection of state and design variables is arbitrary, the variables are partitioned into states and designs based on the non-singularity of $\frac{\partial c}{\partial y}$. Note that the only requirement for the latter case is for the Jacobian of $c(y,u)$ to be full rank. In any case, we will assume for the remainder of this discussion that, for the given selection of states and designs, the matrix $\frac{\partial c}{\partial y}$ is nonsingular for every point (y,u) considered by an analysis algorithm. The non-singularity of $\frac{\partial c}{\partial y}$ allows us to compute a relationship between changes in y with changes in u. If we require that the residual not change (i.e. $c(y,u) = c(y_0,u_0)$) then for sufficiently small δy and δu the higher order terms can be ignored and (2.6) gives

$$\frac{\partial c}{\partial y}\delta y + \frac{\partial c}{\partial u}\delta u = 0. \tag{2.7}$$

If $\frac{\partial c}{\partial y}$ is nonsingular then we can solve (2.7) for

$$\delta y = -\frac{\partial c}{\partial y}^{-1}\frac{\partial c}{\partial u}\delta u. \tag{2.8}$$

The matrix in (2.8) represents the sensitivity of y with respect to u (for $c(y,u) = \textit{constant}$) which defines

$$\frac{\partial y}{\partial u} \equiv -\frac{\partial c}{\partial y}^{-1} \frac{\partial c}{\partial u}. \tag{2.9}$$

We refer to the matrix $\frac{\partial y}{\partial u}$ in (2.9) as the *direct sensitivity matrix*.

2.1.2 Reduced gradients

The major drawback of analysis approaches that rely on the finite-difference reduced gradient is that n_u state solves are required per analysis iteration and the accuracy of the computed optimal solution is degraded because of the truncation error involved with finite differences.

An alternative approach is to compute the reduced gradient in a more efficient and accurate manner. The exact reduced gradient of $\hat{g}(u) = g(y(u), u)$ is

$$\frac{\partial \hat{g}}{\partial u} = \frac{\partial g}{\partial y}\frac{\partial y}{\partial u} + \frac{\partial g}{\partial u} \in \mathbf{R}^{n_g \times n_u}, \tag{2.10}$$

where

$\frac{\partial g}{\partial y}$ is a $\mathbf{R}^{n_g \times n_y}$ auxiliary Jacobian of $g(y,y)$ w.r.t. y evaluated at (y_k, u_k),
$\frac{\partial g}{\partial u}$ is a $\mathbf{R}^{n_g \times n_u}$ auxiliary Jacobian of $g(y,y)$ w.r.t. u evaluated at (y_k, u_k),

and $\frac{\partial y}{\partial u}$ is the direct sensitivity matrix defined in (2.9). By substituting (2.9) into (2.10) we obtain

$$\frac{\partial \hat{g}}{\partial u} = -\frac{\partial g}{\partial y}\frac{\partial c}{\partial y}^{-1}\frac{\partial c}{\partial u} + \frac{\partial g}{\partial u}. \tag{2.11}$$

The first term in (2.11) can be computed in one of two ways. The first approach, called the *direct sensitivity approach*, is to compute the direct sensitivity matrix $\frac{\partial y}{\partial u} = -\frac{\partial c}{\partial y}^{-1}\frac{\partial c}{\partial u}$ first and then compute the product $\frac{\partial g}{\partial y}\frac{\partial y}{\partial u}$. The advantage of this approach is that many simulation codes are already setup to solve for linear systems with $\frac{\partial c}{\partial y}$ since they use a Newton-type method to solve the simulation problem. The disadvantage of the direct sensitivity approach is that to form $\frac{\partial y}{\partial u}$, n_u linear systems must be solved with the Jacobian $\frac{\partial c}{\partial y}$ for each column of $\frac{\partial c}{\partial u}$ as a right-hand side. This is generally a great improvement over the finite-difference reduced gradient in that the solution of a n_u linear systems with $\frac{\partial c}{\partial y}$ is cheaper than a full simulation to evaluate $y(u)$ and the resulting reduced gradient is much more accurate.

The second approach for evaluating (2.11), called the *adjoint sensitivity approach*, is to compute

$$P = \frac{\partial c}{\partial y}^{-T}\frac{\partial g}{\partial y}^{T} \in \mathbf{R}^{n_y \times n_g} \tag{2.12}$$

first, followed by the formation of the product $P^T\frac{\partial c}{\partial u}$. The columns of P are called the *adjoint variables* (or the Lagrange multipliers in an optimization method). The advantage of this approach is that only n_g solves with the matrix $\frac{\partial c}{\partial y}^T$ are required to compute the exact reduced gradients. The disadvantage of the adjoint sensitivities approach is that simulation codes which solve linear systems with the Newton Jacobian $\frac{\partial c}{\partial y}$

may not be able to solve a linear system efficiently with its transpose. It can be a major undertaking to revise a simulation code to solve with transposed systems.

In summary, adjoint approaches are generally more efficient than direct approaches when there there are more auxiliary variables then auxiliary response functions (i.e. $n_u > n_g$). For the majority of the analysis methods that we consider in this work $n_u \gg n_g$ and therefore adjoint methods are typically much more efficient than direct methods.

2.2 Transient sensitivities

Sensitivities for transient problems are more complicated to describe than for steady-state problems. However, if we initially consider a restricted class of problems, we can cleanly present the mathematics involved. Here we consider transient state problems of the form

$$c\left(\frac{\partial y(t)}{\partial t}, y(t), u, t\right) = 0,\ t \in [0, t_f] \tag{2.13}$$

$$y(0) = y_0 \tag{2.14}$$

where

$y(t) \in \mathbf{R}^{n_y}$ are the transient state variables,
$u \in \mathbf{R}^{n_u}$ are steady-state auxiliary variables, and
$c(\partial y(t)/\partial t, y(t), u, t) : \mathbf{R}^{(2n_y+n_u+1)} \rightarrow \mathbf{R}^{n_y}$ are the state DAEs.

Note that while the auxiliary variables u in (2.13)–(2.14) are independent of time, there are also transient problem formulations where the auxiliary variables are time dependent. In fact, some of the applications reported in the next section have time-dependent auxiliary variables. Our reason for focusing on time-independent auxiliary variables in this section is to simplify the mathematics and direct sensitivities for transient problems. However, the basic concepts for time-independent and time-dependent auxiliary variables are similar and therefore we focus on time-independent auxiliary variables.

For transient problems of this type, auxiliary response functions can be formulated by adding a set of auxiliary differential variables and DAEs

$$h\left(\frac{\partial w(t)}{\partial t}, w(t), y(t), u, t\right) = 0,\ t \in [0, t_f] \tag{2.15}$$

$$w(0) = w_0 \tag{2.16}$$

where

$w(t) \in \mathbf{R}^{n_w}$ are auxiliary differential variables,
$h(\partial w(t)/\partial t, w(t), y(t), u, t) : \mathbf{R}^{(2n_w+n_y+n_u+1)} \rightarrow \mathbf{R}^{n_y}$ are the auxiliary DAEs,

and a set of auxiliary response, or observable, functions

$$g(y(t_f), w(t_f), u) \tag{2.17}$$

where

$$g(y(t_f), w(t_f), u) : \mathbf{R}^{(n_y+n_w+n_u)} \rightarrow \mathbf{R}^{n_g}.$$

The purpose of the auxiliary DAEs in (2.15)–(2.16) are typical and the allow the accumulation of some set of integrated quantities needed to evaluate the auxiliary response functions in (2.17).

2.2.1 Implicit DAE solutions and reduced auxiliary response functions

The state and auxiliary DAEs in (2.13)–(2.14) and (2.15)–(2.16) can be integrated simultaneously from the initial conditions, using some integration method, to compute the final state $y(t_f)$ and auxiliary $w(t_f)$ DAE solutions. Under appropriate conditions, this integration method can define the implicit final state function

$$y(t_f) = y(t_f, u) \tag{2.18}$$

and the implicit final auxiliary function

$$w(t_f) = w(t_f, u). \tag{2.19}$$

These implicit DAE solution functions $y(t_f, u)$ and $w(t_f, u)$ allow the definition of the reduced auxiliary response functions

$$\hat{g}(u) = g(y(t_f, u), w(t_f, u), u) \tag{2.20}$$

where

$$\hat{g}(u) : \mathbf{R}^{n_u} \rightarrow \mathbf{R}^{n_g}.$$

It is these reduced auxiliary response functions $\hat{g}(u)$ that most analysis methods are concerned with and for which derivatives must be computed. The reduced derivative computations with $\hat{g}(u)$ are described in the next section.

2.2.2 Reduced auxiliary response function derivatives

The analysis methods require the first derivative $\partial\hat{g}/\partial u$ of the reduced auxiliary response function $\hat{g}(u)$. While finite-difference methods can always be used to approximate these derivatives, we focus on more efficient and accurate methods here. The chain rule of vector-function differentiation gives this reduced derivative as

$$\frac{\partial \hat{g}}{\partial u} = \frac{\partial g}{\partial y}\frac{\partial y(t_f)}{\partial u} + \frac{\partial g}{\partial w}\frac{\partial w(t_f)}{\partial u} + \frac{\partial g}{\partial u} \in \mathbf{R}^{n_g \times n_u}. \tag{2.21}$$

The first two terms in this reduced gradient, involving the implicit functions $y(t_f)$ and $w(t_f)$, dominate the computation. Just as in the steady-state case, these terms can be computed using either a direct approach or an adjoint approach. In the direct approach, the direct sensitivities $\partial y(t_f)/\partial u$ and $\partial w(t_f)/\partial u$ are computed first using a forward integration and then multiplied from the right by $\partial g/\partial y$ and $\partial g/\partial w$ respectively. In the adjoint approach, a new set of adjoint DAEs must be formulated and solved. These methods raise a number of challenging issues.

2.2.3 Transient direct sensitivities

The direct sensitivities of the state $y(t)$ and auxiliary $w(t)$ DAE solutions required in (2.21) can be obtained by simply differentiating (2.13)–(2.14) and (2.15)–(2.16) with respect to u. This differentiation yields the classic direct sensitivity equations

$$\frac{\partial c}{\partial \dot{y}}\left(\frac{\partial}{\partial t}\frac{\partial y(t)}{\partial u}\right)+\frac{\partial c}{\partial y}\left(\frac{\partial y(t)}{\partial u}\right) = -\frac{\partial c}{\partial u},\ t\in[0,t_f]\,, \tag{2.22}$$

$$\frac{\partial y(0)}{\partial u} = 0, \tag{2.23}$$

$$\frac{\partial h}{\partial \dot{w}}\left(\frac{\partial}{\partial t}\frac{\partial w(t)}{\partial u}\right)+\frac{\partial h}{\partial w}\left(\frac{\partial w(t)}{\partial u}\right) = -\frac{\partial h}{\partial y}\left(\frac{\partial y(t)}{\partial u}\right)-\frac{\partial h}{\partial u},\ t\in[0,t_f]\,, \tag{2.24}$$

$$\frac{\partial w(0)}{\partial u} = 0, \tag{2.25}$$

where $\dot{y}$ and $\dot{w}$ are abbreviations for $\partial y(t)/\partial t$ and $\partial w(t)/\partial t$ respectively.

The state (2.22)–(2.23) and auxiliary (2.24)–(2.25) direct sensitivity DAEs are integrated together with the state (2.13)–(2.14) and auxiliary (2.15)–(2.16) DAEs. Because all of these DAE equations are integrated together, only local time-step storage is required. Note that the differential variables $\partial y(t)/\partial u$ and $\partial w(t)/\partial u$ appear linearly in the equations. The dominant cost of the sensitivity computation is the n_u independent DAEs represented in (2.13)–(2.14) that must be solved. It is this set of equations that gives the direct approach its $O(n_u)$ complexity.

Within each time step iteration, approximations for the Jacobians $\partial c/\partial \dot{y}$, $\partial c/\partial y$ and $\partial c/\partial u$ for the main simulation code are required. If finite differencing is used, then direct sensitivity methods can be implemented using just what implicit transient simulation codes already provide. The simulation code is required to give up control of its time stepping algorithm and thereby just becoming a slave to the transient direct sensitivity solver. Note that any integration method can be used to solve the transient direct sensitivity equations, including explicit methods. For an explicit time integration method, only the action of the Jacobian matrices $\partial c/\partial \dot{y}$ and $\partial c/\partial y$ onto arbitrary vectors are required and these can be relatively cheaply (although inaccurately) computed using directional finite differences. In addition, higher levels of parallelism can be utilized where blocks of different direct sensitivities in $\partial y(t)/\partial u$ and $\partial w(t)/\partial u$ are solved on different clusters of processors. As long as there are sufficiently many right-hand sides to solve and the computational load is well balanced, then the overall parallel efficiency should be very good and should result in dramatic reductions in wall-clock time.

2.2.4 Transient adjoint sensitivities

The reduced gradient in (2.21) can also be computed using an adjoint approach. Unlike the steady-state case, we will not present a complete derivation of the adjoint approach since the concepts involved are more complicated. Instead, here we simply show the relevant equations in order to provide a context for the discussion that follows.

The adjoint approach for computing the reduced gradient in (2.21), after the solution of the forward

problem, begins with the solution of the adjoint equations

$$\frac{\partial h}{\partial \dot{w}}^T \frac{\partial Q}{\partial t} + \left[\frac{d}{dt} \frac{\partial h}{\partial \dot{w}}^T + \frac{\partial h}{\partial w} \right] Q = 0, \; t \in [0, t_f], \tag{2.26}$$

$$\left(\frac{\partial h}{\partial \dot{w}}^T Q \right) \Big|_{t=t_f} = \frac{\partial g}{\partial w(t_f)}^T \tag{2.27}$$

$$\frac{\partial c}{\partial \dot{y}}^T \frac{\partial P}{\partial t} + \left[\frac{d}{dt} \frac{\partial c}{\partial \dot{w}}^T + \frac{\partial c}{\partial w} \right] P = -\frac{\partial h}{\partial \dot{y}}^T \frac{\partial Q}{\partial t} - \left[\frac{d}{dt} \frac{\partial h}{\partial \dot{y}}^T + \frac{\partial h}{\partial y} \right] Q, \; t \in [0, t_f], \tag{2.28}$$

$$\left(\frac{\partial c}{\partial \dot{y}}^T P \right) \Big|_{t=t_f} = \frac{\partial g}{\partial y(t_f)}^T \tag{2.29}$$

where

$Q \in \mathbf{R}^{n_h \times n_g}$ are the adjoint variables for the auxiliary DAEs, and
$P \in \mathbf{R}^{n_y \times n_g}$ are the adjoint variables for the state DAEs.

The above adjoint equations can be derived in a number of ways but, in essence, the solution of these equations gives the sensitivity in the auxiliary functions of interest with respect the perturbations in the DAE equations. Note that the above adjoint DAEs have final conditions at $t = t_f$ and therefore must be integrated from the final time backwards. The adjoint DAE in (2.28) is expressed in such a way as to emphasize that it depends on the adjoint solution Q. Hence, the adjoint for the state equation in (2.28) depends on the independent solution to the adjoint for the auxiliary DAE in (2.26) which is just the opposite to the relationship between the state (2.13) and auxiliary (2.15) DAEs.

Also note that the adjoint equations require access to the state and auxiliary DAE solutions $y(t)$ and $w(t)$. This need to access the forward DAE solutions during the backward adjoint solve is perhaps the most challenging aspect to an adjoint approach for large scale transient problems. This is a critical issue that is discussed later chapters.

Given that the adjoint solutions $P(t)$ and $Q(t)$ are computed, the reduced auxiliary response derivative in (2.21) is evaluated as

$$\frac{\partial \hat{g}}{\partial u} = -\int_0^{t_f} \left(P^T \frac{\partial c}{\partial u} + Q^T \frac{\partial h}{\partial u} \right) dt + \frac{\partial g}{\partial u}. \tag{2.30}$$

The adjoint solution variables Q and P need not be stored for the entire time domain. This is because the integrals in (2.30) can be computed while the adjoint equations in (2.26)–(2.29) are being integrated backward and therefore the integrators only have to store the typical local time-step information. What is significant, of course, about the adjoint approach is that the dominate computation of the n_g simultaneous state adjoint DAEs in (2.28)–(2.29) can be computed with $O(n_g)$ complexity, independent of the number of auxiliary variables n_u.

Chapter 3

Offline/Online QP Solver Stratgies for Real Time Inversion

3.1 Introduction

Sensitivity computations are an important part of gradient-based optimization algorithms and, as described in the previous chapter, a variety of strategies can be considered to calculate the underlying derivatives. In this chapter, large-scale direct sensitivities are exploited to solve an inversion problem of source terms on the domain boundary subject to steady-state convection diffusion by matching only a small set of observations. This problem formulation can be applied to a variety of problems such as identifying contamination events or controling temperature in heating and ventilation systems. The goal of the inversion problem is to minimize the difference between the observations and predictions. Because the number of observations will typically be less than the possible discretization points on the boundary, the problem is ill-posed and needs to be regularized, which amounts to an additional penalty type term in the cost functional. Even though the regularization term causes a smoothing of the reconstruction, this formulation has proven to be effective in solving inversion problems [206, 7]. In our numerical experiments, a forward simulation with specified initial conditions and source terms is used to extract observations from a few locations on the boundary and then fed into the inversion problem as the target values.

Real-time performance is a critical goal in order for an inversion of contamination events or temperature control to have any practical use. A variety of approaches are discussed in this report to solve large-scale optimization problems (and compute the underlying derivatives) as efficiently as possible, but here we focus on decomposing the optimization problem into offline and online calculations. The general idea is to recognize that a significant portions of the calculations do not change within the optimization solution procedure and can therefore be extracted as offline computations. What remains is then a subset of the overall computation and can be performed online with real-time performance. We describe a tailored solution of a special class of quadratic programming (QP) problems that arise in least-squares problems. The offline computation involves the formation of a compressed state/source sensitivity matrix. This computation can be performed using a direct or adjoint approach and underscores the need for such computations. The online computation uses the precomputed compressed state/source sensitivity matrix to compute a reduced gradient and reduced Hessian and solve a reduced QP online.

The proposed offline/online reduced QP method is demonstrated on a prototype problem where a contamination event is simulated in an airport terminal. The transport of the chemical is based on the steady-state convection-diffusion equations and the velocities are calculated through a steady-state Navier Stokes model. A least-squares formulation attempts to reduce the difference between sparse concentration information and simulated values in an attempt to reconstruct the location and character of the contamination event. The main target application for this specialized QP method is the repeated online solution of inversion problems involving large-scale linear state constraints. The algorithmic strategy is presented in discrete and general form to emphasize the applicability of the formulation to other problems with similar characteristics.

3.1.1 Statement of the Problem: General QP Formulation

The offline and online calculation strategy is based on the recognition that our particular inversion problem can be formulated as a Quadratic Programming (QP) problem in which the cost functional is quadratic and the contraints are linear. This class of problems can take advantage of standard solution methods [92, 174]. To achieve real-time performance, however, we consider a new solution approach for QPs of the following form:

$$\begin{aligned} \text{Minimize} \quad & \tfrac{1}{2}(y-\tilde{y})^T Q_y (y-\tilde{y}) + \tfrac{1}{2} u^T Q_u u && (3.1)\\ \text{Subject to} \quad & Cy + Nu = 0 && (3.2)\\ & u_L \le u \le u_U && (3.3) \end{aligned}$$

where

$y \in \mathbf{R}^{n_y}$ are state variables,
$u \in \mathbf{R}^{n_u}$ are design (or inversion) variables,
$\tilde{y} \in \mathbf{R}^{n_y}$ is some target state vector,
$u_L, u_U \in \mathbf{R}^{n_u}$ are simple bounds on u,
$Q_y \in \mathbf{R}^{n_y \times n_y}$ is a diagonal positive semi-definite matrix with n_s non-zero entries,
$Q_u \in \mathbf{R}^{n_u \times n_u}$ is a positive definite matrix,
$C \in \mathbf{R}^{n_y \times n_y}$ is a nonsingular unsymmetric matrix known as the *state Jacobian*, and
$N \in \mathbf{R}^{n_y \times n_u}$ is a rectangular matrix known as the *design Jacobian*.

In the context of an inversion problem, this formulation solves for the inversion variables u that give the best possible match of the linear state model (3.2) for y to some target data $\tilde{y}$. Only incomplete information about the target state in $\tilde{y}$ is known and these are selected by the nonzero entries along the diagonal of Q_y. In this case, the number of data points n_s known in the target state $\tilde{y}$ is less than the number of inversion parameters n_u. This gives rise to an ill-posed inversion problem where an entire subspace of solutions for u exists. Some type of regularization is therefore added to the objective function in the form of the positive-definite matrix Q_u that results in a well-posed QP with a unique solution for u. For problems with many inversion parameters, enforcing physical bounds on the inversion parameters (for example, non-negative concentrations) in (3.3) result in much better inversion results in terms of quality of inversion. Instead of inverting for every location in a 3D domain, we consider only the boundary of the 3D domain

and in addition we parametrize the domain so that we can control the final number of inversion parameters. We focus therefore on the solution of QPs where n_u and n_s are not excessively large (i.e. n_u and n_y are a few thousand or less).

In addition, to achieve real-time performance we recognize that the data in Q_y and $\tilde{y}$ change from one QP to the next and not the state model matrices C or N. We are also interested in using approaches that allow for a fast solution time for repeated QP subproblems. It should be noted that for most problems, even a single iterative solve with the full state Jacobian C (or its adjoint) can not be performed online in real-time. Note that only the n_s entries in $\tilde{y}$ selected by the nonzero diagonals of Q_y are significant. We also assume that the structure of Q_y (that is, which diagonals are zero or nonzero) does not change from one QP to the next. What is critical about this approach is that the state equation matrices C and N are constant for all QPs that we consider. This allows one to perform perhaps some very expensive computations with C and N offline and then use these computed quantities in the repeated online solution of the QPs.

3.2 Reduced QP problem

For the class of QPs considered here, it is useful to form a reduced QP where the state variables y and the linear equality constraints in (3.2) are eliminated from the problem. It is this linear elimination that allows a significant portion of the computation to be performed offline and therefore will be the key to the success of the repeated online solution of this class of QPs.

The elimination of the linear equality constraints in (3.2) gives rise to the following implicit state function

$$y(u) = Du \tag{3.4}$$

where

$D = -C^{-1}N \in \mathbf{R}^{n_y \times n_u}$ is known as the direct sensitivity matrix.

The direct sensitivity matrix D can be computed offline since it does not depend on the data in Q_y or $\tilde{y}$.

To obtain the reduced QP, $y(u)$ is substituted into (3.1) to obtain

$$\begin{aligned} &\text{Minimize} && g^T u + \tfrac{1}{2} u^T G u && (3.5)\\ &\text{Subject to} && u_L \le u \le u_U && (3.6) \end{aligned}$$

where

$g = -D^T Q_y \tilde{y} \in \mathbf{R}^{n_y}$ is the *reduced gradient*, and
$G = D^T Q_y D + Q_u \in \mathbf{R}^{n_y \times n_y}$ is the *reduced Hessian*.

It is this reduced QP formulation in (3.5)–(3.6) that is the focus of the offline/online decomposition described in the sequel.

3.3 Decomposition of Reduced QP into Offline and Online Subproblems

Here we consider a decomposition of the reduced QP in (3.5)–(3.6) into offline and online subproblems to achieve real-time performance. An important concept of the approach is to realize that not all the values of the direct sensitivity matrix D are required in the optimization calculation. In the inversion problem, only the state concentration values at sensor locations need to be extracted, stored and then read in during the on-line computation. We therefore need to define an operator E that depends on the sensor locations and can map the required rows of D to a more compact operator $\hat{D}$. This avoids large storage requirements and IO costs during the online calculation process.

We provide a general overview of the decomposition algorithm and further define the individual calculations in subsequent sections. The general algorithm (which refers to some quantities $\hat{D}$, $\hat{Q}_y$ and $\hat{y}$ that are defined later) is as follows:

Algorithm 1. OFFLINE AND ONLINE COMPUTATIONAL PROCEDURE

1. *Offline Computations*
   - compute compressed direct sensitivity matrix $\hat{D}$

2. *Online Computations*
   - read in compressed direct sensitivity matrix $\hat{D}$
   - for each online QP problem
     - read in compressed data $\hat{Q}_y$ and $\hat{y}$
     - assemble reduced gradient g
     - assemble and factor reduced Hessian G
     - solve bound constrained QP (3.5)–(3.6)

In the offline computation, an operator is defined that compresses the full direct sensitivity matrix D to a compact matrix consisting of sensitivities of state variables only at observation points; the sensitivities of all other state variables do not impact the optimization problem in any way. We therefore define a *scattering matrix* $E \in \mathbf{R}^{n_y \times n_s}$ as follows:

$$Q_y = E^T \hat{Q}_y E, \tag{3.7}$$

$$\tilde{y} = E^T \hat{y}, \tag{3.8}$$

where

$\hat{Q}_y \in \mathbf{R}^{n_s \times n_s}$ is a diagonal positive definite matrix, and
$\hat{y} \in \mathbf{R}^{n_s}$ is a data vector.

The diagonal positive-definite matrix $\hat{Q}_y$ contains only positive diagonals while Q_y contains n_s positive diagonal entries and $n_y - n_s$ zero diagonal entries. Therefore, $\hat{Q}_y$ is the "significant" part of Q_y. Likewise, $\hat{y}$ is the "significant" part of $\tilde{y}$. In the sequel, $\hat{Q}_y$ and $\hat{y}$ will be referred to as the *compressed data* and it is this data that changes from one QP to the next. The scattering matrix E is the same for all QP subproblems.

Given the above definition of the scattering matrix E and the compressed data$\hat{Q}_y$ and $\hat{y}$, the reduced gradient and reduced Hessian become

$$g = \hat{D}^T \hat{Q}_y \hat{y}, \tag{3.9}$$
$$G = \hat{D}^T \hat{Q}_y \hat{D} + Q_u, \tag{3.10}$$

where

$\hat{D} = ED = -EC^{-1}N \in \mathbf{R}^{n_s \times n_u}$ is the *compressed direct sensitivity matrix.*

The next subsection describes various approaches for the offline computation of the compressed direct sensitivity matrix $\hat{D}$. This is followed Subsection 3.3.2 by a discussion of how, given a precomputed matrix $\hat{D}$, the reduced QP subproblem in (3.5)–(3.6) is formed and solved.

3.3.1 Offline Subproblem

The offline subproblem involves the precomputation of the compressed direct sensitivity matrix $\hat{D} = -EC^{-1}N$. This matrix can be computed by either first solving the set of n_u simultaneous systems $C^{-1}N$ (known as the forward approach), or by first solving the set of n_s simultaneous systems $C^{-T}E^T$ (known as the adjoint approach). There are many factors to consider when selecting which of these two approaches to use. Note that for some challenging applications, the number of state variables n_y may be in the millions or more and solving systems involving C and C^T may require the use iterative linear solvers (such as GMRES) on a distributed-memory massively parallel processing (MPP) computer. However, $\hat{D} = -EC^{-1}N \in \mathbf{R}^{n_s \times n_u}$ will always be a relatively small matrix since we are only considering problems where n_u and n_s are a few thousand at most. Therefore, while the computation of $\hat{D}$ may be able to effectively utilize a large MPP, the storage and use of $\hat{D}$ can not and a simple single-processor or symmetric multi-processor (SMP) is sufficient for all of the online computations described below.

These two approaches, and the issues involved with each, are described in the following subsections.

3.3.1.1 Computing Compressed Sensitivities using the Forward Approach

The forward approach for computing the compressed direct sensitivity matrix $\hat{D} = -EC^{-1}N$, which involves first solving the n_u simultaneous systems

$$D = -C^{-1}N \tag{3.11}$$

followed by simply selecting the rows by multiplying $\hat{D} = ED$, is potentially the most efficient approach for steady-state and transient problems when $n_u < n_s$. However, even when $n_u > n_s$, the forward approach may still be preferable since it only requires forward solves.

For a steady-state problem, solving $C^{-1}N$ requires a single block iterative linear solve with n_u simultaneous right-hand sides. For a transient problem, solving $C^{-1}N$ actually involves solving a set of n_u simultaneous linear ODEs from the initial time to the final time. For both the steady-state and transient problems, block linear solvers, such as block GMRES, may greatly reduce the cost of these expensive offline computations.

3.3.1.2 Computing Compressed Sensitivities using the Adjoint Approach

The adjoint approach for computing the compressed direct sensitivity matrix $\hat{D} = -EC^{-1}N$, which involves first solving the n_s simultaneous adjoint systems

$$T = C^{-T}E^T \tag{3.12}$$

followed by simply multiplying $\hat{D} = -T^T N$, is probably the more efficient approach when $n_s < n_u$. If an iterative linear solver such as GMRES is being used, then solving the adjoint system requires adjoint multi-vector products of the form $C^T V$. Note that there is no cheap way of approximating these adjoint products using directional finite differences.

For a steady-state problem, solving $C^{-T}E^T$ requires a single block adjoint iterative linear solve with n_s simultaneous right-hand sides. For a transient problem, a set of n_s simultaneous linear ODEs must be integrated from the final time backwards to the initial time. Since the adjoint sensitivity equations are linear and do not depend on the state, no transient state information needs to be stored and only local time-step information is needed. For both the steady-state and transient problems, block linear solvers, such as block GMRES, may greatly reduce the cost of these expensive offline adjoint computations.

3.3.2 Online Subproblem

The solution of the online problem first involves forming the reduced gradient g (3.9) and reduced Hessian G (3.10) given the compressed data $\hat{Q}_y$, and $\hat{y}$, and the precomputed compressed direct sensitivity matrix $\hat{D}$. Once these quantities are formed, the reduced bound-constrained QP (3.5)–(3.6) is then solved. All of these online calculations are relatively computationally inexpensive (compared to the offline computation) and can therefore be performed on a single processor or multi-processor SMP shared-memory machine. On an SMP, all of the level-2 and level-3 BLAS and LAPACK operations performed can exploit more than one thread of execution and therefore provide some parallel speedup.

The computation of the reduced gradient $\hat{g} = \hat{D}^T \hat{Q}_y \hat{y}$ is straightforward and relatively cheap. Computing $\hat{g}$ only requires an $O(n_s)$ diagonal scaling to form $\hat{Q}_y \hat{y}$ followed by an $O(n_u n_s)$ level-2 matrix-vector multiplication $\hat{g} = \hat{D}^T(\hat{Q}_y \hat{y})$. The cost of this computation is in the noise of the CPU time of the other online computations.

The reduced Hessian $G = \hat{D}^T \hat{Q}_y \hat{D} + Q_u$ is formed online as follows. First, the diagonal matrix $\hat{Q}_y$ is decomposed into

$$\hat{Q}_y = \hat{Q}_y^{1/2} \hat{Q}_y^{1/2} \tag{3.13}$$

by simply taking the square root of the diagonal of $\hat{Q}_y$ (which gives real components since $\hat{Q}_y$ is positive definite). This is only an $O(n_s)$ operation and is therefore very fast.

Next, the matrix

$$\hat{J} = \hat{Q}_y^{1/2} \hat{D} \tag{3.14}$$

is formed by simply scaling the rows of $\hat{D}$ using the diagonal entries of $\hat{Q}_y^{1/2}$. This is an $O(n_s n_u)$ operation.

After $\hat{J}$ is calculated, the $O(n_u (n_s)^2)$ level-3 BLAS operation

$$\hat{S} = \hat{J}^T \hat{J} \in \mathbf{R}^{n_u \times n_u} \tag{3.15}$$

is performed. For problems where $n_s \gg n_u$ (for example with least-squares problems), this computation is the dominate cost of the online solution of the reduced QP. For problems where $n_s < n_u$ (which is always the case in an ill-posed inversion problem for instance) then $\hat{S}$ will be singular with at least $n_u - n_s$ zero eigenvalues and this computation will not be a dominate component of the online solution.

Once the matrix $\hat{S}$ is formed, it is added to the matrix Q_u to form the reduced Hessian

$$G = \hat{S} + Q_u \tag{3.16}$$

which is an $O(n_u^2)$ operation. Note that since $\hat{S}$ is positive semi-definite and Q_u is positive definite, then G is guaranteed to be positive definite. However, the conditioning of G my be very poor if $\hat{S}$ is singular and dominates Q_u.

Finally, the active-set QP solver used to solve (3.5)–(3.6) requires many solves with G and this is facilitated by first performing an $O(n_u^3)$ Cholesky factorization of G. Of all of the online computations described up to this point, this $O(n_u^3)$ Cholesky factorization is by far the most expensive when $n_s \ll n_u$.

After the reduced gradient g has been formed and the reduced Hessian G has been calculated and factored, then the reduced QP subproblem (3.5)–(3.6) can be solved. There are many different approaches to solving such a QP but the one used in the results reported in Section 3.4 was QPSchur [26]. QPSchur solves convex QPs (that is, QPs where the Hessian is positive definite) using a dual method that starts with the unconstrained minimizer and then adds violated inequality constraints until the solution is found. Each addition to the working set involves at least one $O(n_u^2)$ level-2 BLAS backsolve with the Cholesky factors of G. Therefore, the online cost of solving a QP with n_{act} active inequality constraints is at least $O(n_u^2 n_{act})$. Note that the number of QP iterations can easily exceed n_u and each of these iterations can only utilize level-2 BLAS. Therefore, even through the number of flops in the QP solve is still usually just $O(n_u^3)$, the actual runtime can be much higher than the $O(n_u^3)$ Cholesky factorization of G which can utilize more efficient level-3 BLAS. Hence, if n_{act} is moderately large, then QPSchur will dominate the online solve.

3.4 Example Application: Source Inversion for Steady-State Convection-Diffusion in a 3D Airport Terminal

Here we describe a steady-state inversion problem with a 3D model of an airport terminal. The goal of the inversion is to attempt to invert for the location and intensity of a chemical, biological or radiological release given sparse sensor data. In these scenarios, it is critical to be able to quickly (in a matter of seconds) invert for the source location of a possible terrorist attack so that remediation procedures can be quickly put into place.

The model used for the results in this section was that of an airport terminal shown in Figure 3.1. The governing equations were comprised of a finite-element discretization of the convection-diffusion equations. The velocity field is calculated (offline) through a Navier Stokes solver with turbulence modeling. The resulting velocity is field $\mathbf{v}$ is based on inflow and outflow setting for a certain operating condition, which is assumed to be constant during the inversion calculation. The transport equation for

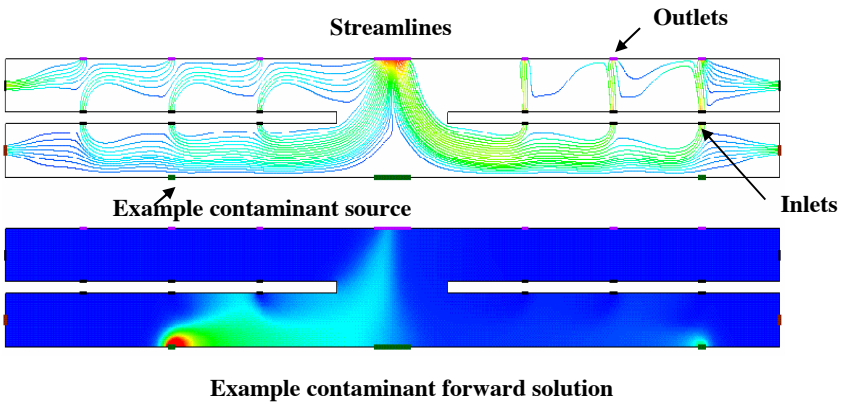


Figure 3.1. Two-D cross-section of the 3D airport terminal model. Also shown is an example flow pattern and contaminate release simulation.

contamination concentration y in a domain Ω is given by the standard convection-diffusion equation:

$$-k\Delta y + \mathbf{v} \cdot \nabla y = 0, \tag{3.17}$$
$$y(0) = y_0 \ \ on \ \ \Omega, \tag{3.18}$$
$$\partial y/\partial n = u \ \ on \ \ \Gamma_N, \tag{3.19}$$
$$\partial y/\partial n = 0 \ \ on \ \ \Gamma_D \tag{3.20}$$

where $k > 0$ is the molecular diffusion (assumed constant), Γ_N the portion of the boundary where the source u is injected, and Γ_D is the portion of the boundary where no source injection is allowed.

Applying a standard first-order finite-element discretization to the week form of the PDE gives the linear state equations in (3.2). Both the Navier Stokes and convection-diffusion equations were implemented in a production CFD code (MPSalsa). For additional details the intereted readers is referred to Shadid et al. [131]

In this model, possible source locations included only the bottom floor. The source was discretized using evenly spaced source locations. The chosen source discretization of the bottom floor neglects the possibility of the source emanating from other locations in the domain. While a more general source discretization could have been used, the choice of using only the bottom floor was for simplicity and to limit the number of inversion parameters n_u.

Figure 3.2 shows the inversion results compared to the input source field for an example run where $n_u = 100$ inversion parameters were used. This figure clearly shows that the inverted source matches the input source very well. Including the simple bounds (3.6), where $u_L \approx 0.0$, had a large impact on the quality of the solution and were critical for the qualitative accuracy of the inversion results.

Table 3.1 shows the wall-clock CPU times for the offline and online subproblems for $n_u = 100$ and $n_u = 500$ inversion parameters. These timings show that while the cost of computing the compressed direct sensitivity matrix $\hat{D}$ offline can be significant, the online cost is competitively small. In fact, the online inversion for only $n_u = 100$ took less that 0.05 seconds! In addition, the cost for the online

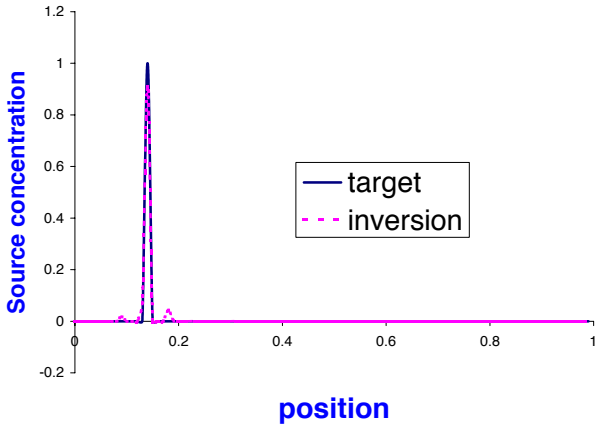


Figure 3.2. Plots for a 2D cross-section of the top bottom floor in the 3D airport terminal model (shown in Figure 3.1) of input source and inverted source for $n_u = 100$ discretized inversion parameters.

Computation	$n_u = 100$		$n_u = 500$	
Compute $\hat{D}$ offline	28.03	min	137.68	min
Compute/factor G online	0.016	sec	0.14	sec
Solve reduced QP online	0.038	sec	2.25	sec

Table 3.1. Offline and online times for inversion of 3D airport terminal model. Offline computation of compressed direct sensitivity matrix $\hat{D}$ performed using four nodes of a 1.7 MHz Beowulf cluster. Online computations performed on a single 1.7 MHz processor of same Beowulf cluster.

inversion for $n_u = 500$ took less than 2.5 seconds, but here one can see the increase in online complexity as n_u increases. Both of these QP subproblems had many active inequality constraints and therefore the online cost was dominated by the active-set QP solver QPSchur. When there are many active inequality constraints at the solution then QPSchur scales at about $O(n_u^3)$ which limits the applicability of this approach. Relative to the $n_u = 100$ case, a cubic increase in solution complexity would suggest a CPU time of $0.038(500/100)^3 = 4.75$ seconds for $n_u = 500$; however, the actual QPSchur time was only 2.25 seconds. Clearly there is some economy of scale occurring in QPSchur (i.e., same overhead but increased flops resulting in overall faster flop rate) what will tapper off as n_u increases.

3.5 Summary and Conclusions

Here described is an approach for decomposing the repeated solution of a class of QP problems into offline and online subproblems. The general formulation for the class of QPs we consider encompasses several different types of inversion and least-squares problems with both steady-state and transient state models. Our method exploits the properties that each QP to be solved has the same linear state constraint matrices $C \in \mathbf{R}^{n_y \times n_y}$ and $N \in \mathbf{R}^{n_y \times n_u}$, and has a diagonal objective function Hessian $Q_y \in \mathbf{R}^{n_y \times n_y}$ with the same nonzero structure along the diagonal (which is represented as a mapping matrix $E \in \mathbf{R}^{n_s \times n_y}$). We exploit these properties by splitting the computation of the solution of the QP into offline and online components. This approach allows for the "real time" solution of a relatively broad class of these QPs using modest computing resources (at least for the online computation).

Offline, a compressed sensitivity matrix $\hat{D} = -EC^{-1}N \in \mathbf{R}^{n_s \times n_u}$ is computed that only depends on the state constraint matrices C and N, and the structure of the state objective Hessian matrix represented by E. The offline formation of $\hat{D}$ can be very computationally expensive, can use either forward or adjoint linear solver methods, and can be computed using large-scale iterative linear solvers on a distributed-memory super computer.

The cost of the offline computation of $\hat{D}$ scales as $O(n_u)$ for the forward approach and as $O(n_s)$ for the adjoint approach. This means that for typical inversion problems where $n_u > n_s$ that adjoint methods may be preferable.

Each particular online QP can have a different compressed diagonal objective Hessian matrix $\hat{Q}_y \in \mathbf{R}^{n_s \times n_s}$, compressed target state vector $\hat{y} \in \mathbf{R}^{n_s}$, and regularization Hessian matrix $Q_u \in \mathbf{R}^{n_u \times n_u}$. Given this data, the reduced gradient $g = \hat{D}^T \hat{Q}_y \hat{y} \in \mathbf{R}^{n_s}$ and the reduced Hessian $G = \hat{D}^T \hat{Q}_y \hat{D} + Q_u \in \mathbf{R}^{n_s \times n_s}$ are formed online and then a bound-constrained reduced QP subproblem with n_u variables is solved.

The cost of the online computation scales in general as $O(n_u^3)$ and it independent of the size of the state space n_y. The online computations never require even a single iterative linear solve involving the state Jacobian C or its adjoint C^T. Most notably, the online problem can be solved on a single processor machine.

The above described approach was demonstrated on a steady-state inversion problem involving a 3D airport terminal model where the state constraint matrices were produced using the CFD code MPSalsa. Inversion results where reported for $n_u = 100$ and $n_u = 500$ inversion parameters. While the offline cost of computing the compressed sensitivity matrix $\hat{D}$ was considerable (e.g. one hour on four processors of a Beowulf cluster for $n_u = 500$), the online costs were less than 0.05 and 2.30 seconds for $n_u = 100$ and

$n_u = 500$, respectively, on a single 1.7 MHz Linux workstation.

These results clearly demonstrate that repeatably solving large-scale QP problems of the type considered here in "real time" is very tractable using the proposed offline/online approach given current computer technology.

3.6 Recommendations and Future Work

Several different avenues to pursue to continue this work which are described below.

- The offline/online decomposition approach for solving inversion QPs was demonstrated on a steady-state model of an airport terminal. However, the principles of the proposed approach are equally applicable for transient state models as well. Transient problems will require the use of time integrators for the forward (solving forward in time) and adjoint (solving backward in time) computation of the compressed sensitivity matrix $\hat{D}$. However, once $\hat{D}$ is computed, it can be used in the online computations in exactly the same way as for the steady-state case. Another challenge for transient problems will be in keeping the total number of transient inversion parameters n_u from getting too large which would increase the cost of the $O(n_u^3)$ online computations.
- For a large number of inversion parameters n_u (e.g. for transient problems), the $O(n_u^3)$ cost of solving the online reduced QP subproblem will become prohibitive and threaten the "real time" capability to solve these problems. In this cases, more attention needs to be focused on the solution of the reduced QP subproblem.

  Possible areas to pursue include:

  - The use of approximate reduced Hessians instead of recomputing them from scratch every time. This approach could be particularly effective when solving a sequence of related QPs for a transient inversion problem.
  - The use of warm-start estimates of the optimal active-set of inequality constraints from the last QP subproblem.
  - The use of a primal QP solver instead of the dual QP solver QPSchur. A primal QP solver may allow for better premature termination of the the QP algorithm that still results in sufficient quality inversion results.
  - The use of an interior-point algorithm to approximately solve the reduced QP. An interior-point method would require an $O(n_u^3)$ Cholesky factorization at each interior-point iteration but each of these factorizations would utilize level-3 BLAS for peak performance. Premature termination of the algorithm would guarantee that the bounds in (3.6) would not be violated.
- The computation of the compressed sensitivity matrix $\hat{D}$ can consume considerable computational resources, especially for transient problems. While this computation does not impact the cost of the online computations it does represent a real cost that can be a hindrance if it is too expensive. The forward and adjoint approaches that solve for $D = C^{-1}N \in \mathbf{R}^{n_y \times n_u}$ and $T = C^{-T}E^T \in \mathbf{R}^{n_y \times n_s}$, respectively, can both take advantage of block linear solver methods such as block GMRES. The use of a block solver may reduce the considerable cost of the offline computations by an order of magnitude or more in some cases.

Chapter 4

Large Scale Inversion of a Contaminant in a Regional Model using Convection-Diffusion

4.1 Introduction

We are interested in accurately characterizing airborne contamination events in regional models under the assumption that sparse observation data is available spatially and temporally. In particular, we aim to invert for initial conditions of contaminants from observations, using a convection-diffusion model for their transport. These types of problems arise from characterization of pollutants in the atmosphere, unintentional catastrophic accidents involving chemical plants, or terrorist type release of chemical or biological agents. Although adjoint formulations have been developed to characterize the sensitivity of chemical concentration with respect to source terms [156, 137, 132, 67], very little has been done to reconstruct the initial conditions via the solution of an inverse problem.

In the previous chapter the use of direct sensitivities was successfully demonstrated for a source inversion problem. The number of inversion parameters, however, was limited to O(100). In the case of a contamination inversion in a large regional model setting, the number of inversion parameters may be several orders of magnitude greater than that of the internal airflow problem. More efficient methods must therefore be developed to enable large-scale inversion in time scales appropriate for mitigation procedures. Two options can be considered: 1) full space approaches, and 2) reduced space method using adjoints.

Full space methods have been proven to efficiently scale to extremely large design spaces, even as large as the size of the underlying discretization [36]. Unfortunately, for a time dependent PDE constraint, the resulting KKT matrix involves the entire space–time discretization, is indefinite, and is generally very ill-conditioned.

Alternatively, we consider a reduced space method, which is equivalent to a block elimination on the three PDEs that comprise the first order optimality conditions: the state equation, the adjoint equation, and the inversion equation. By block-eliminating the state and adjoint variables, we are left with a reduced system in the space of just the inversion variables, which represent the initial condition field and are therefore of dimension of just the spatial discretization. The coefficient matrix of this system — the reduced Hessian — is dense as a result of the block elimination, and therefore iterative methods that require only the matrix-vector product must be used. These can be formed at each iteration by exploiting the structure of

the reduced Hessian to solve a pair of forward–adjoint convection diffusion problems. Not only does the reduced Hessian eliminate the time dimension of the KKT system, but with appropriate regularization it is also positive definite. However, for large systems, a scalable preconditioner will be necessary, which is particularly challenging when the reduced Hessian cannot be formed.

Our ultimate goal is to invert in real time for initial conditions using a well resolved terrain model while capturing the appropriate physics. We have organized our investigation into several phases: (1) develop inversion algorithms using a steady state velocity field from a laminar Navier Stokes solver and contaminant transport dynamics from a scalar convection-diffusion model, (2) prototype our algorithms for a large regional model with realistic topography, (3) incorporate additional physics into the model to address turbulence, which would require the ability to use unsteady velocity fields in the convection diffusion inversion algorithm, and (4) incorporate additional climate physics and invert for the velocity field from pointwise observations. Our investigations thus far have been limited to the first two items as we have worked on developing scalable inversion techniques. The representation of the underlying physics associated with contaminant transport in a regional model is simplified, but it is sufficiently complex (realistic terrain with Navier-Stokes-based velocity field, convection-diffusion in three dimensions plus time) to stress the inversion capabilities. To the best of our knowledge large scale inversion of contaminant transport in a regional model has not been attempted.

In previous work a formulation was developed to demonstrate inversion in simplified geometries that assumed a simple velocity field [7]. The purpose was to primarily develop inversion capabilities and investigate associated issues, such as regularization, convergence, and general computational issues. In this work, we extend our investigation to three dimensional time-dependent transport, reduced space methods, and realistic terrain topography. This chapter will first present our mathematical formulation of the time dependent convection diffusion equations, followed by the development of the inversion algorithm. The numerical results section discusses computational experiments for evaluating the effect of sensors and regularization constants, in addition to the effects of topography.

4.2 Convection-Diffusion and First Order Optimality Conditions

In order to reconstruct initial conditions using sparse concentration observations, we formulate a regularized least squares objective function constrained by convection diffusion. The goal is to reconcile the differences between observed and simulated concentration values. The strong form of the formulation is presented below:

$$\min_{c,c_0} \sum_j \int_\Omega \int_0^T (c - c^*)^2 \delta(x - x_j)\, dx\, dt + \frac{\beta}{2} \int_\Omega \nabla c_0 \cdot \nabla c_0\, dx \tag{4.1}$$

$$\text{s.t. } \partial c/\partial t - k\Delta c + \mathbf{v} \cdot \nabla c = 0 \quad \text{in} \quad \Omega \times (0,T), \tag{4.2}$$

$$c(0) = c_0 \text{ on } \Omega, \tag{4.3}$$

$$\partial c/\partial n = 0 \text{ on } \Gamma_N \times (0,T), \tag{4.4}$$

$$c = 0 \text{ on } \Gamma_D \times (0,T) \tag{4.5}$$

where c is the state variable or concentrations in this application context, $\hat{c}$ is the target concentration (pre-computed from a forward simulation for our numerical experiments; otherwise $\hat{c}$ corresponds to sensor readings), the delta function represents the location of the observations, the constant β determines the contribution of the regularization term, and c_0 is initial concentration or inversion variable.

The minimization of the objective function is subject to convection-diffusion where the velocity field $\mathbf{v}$ is calculated from a Navier Stokes model. The diffusion constant is specified as k. Γ_N denotes the portion of the boundary with homogeneous Neumann conditoins, and Γ_D denotes the portion of the boundary with homogeneous Dirichlet conditions. In follow-on work we will employ a turbulent Navier Stokes model to invert for velocities given sparse observations. At this algorithmic development stage of the project, however, we assume known velocity boundary conditions and use a steady state Navier Stokes system to calculate the velocity field.

For the remainder of the mathematical presentation we follow a discretize-then-optimize approach. We first present the formulation in weak form and discretize the forward convection diffusion equation with Streamline Upwind Petrov Galerkin (SUPG) approximation in space, and use the Crank-Nicholson method in time. The discrete form of the inverse problem is then discussed, and first order optimality condition (KKT conditions) are presented.

4.2.1 Contaminant Transport by Convection-Diffusion

The strong form of the convection-diffusion initial-boundary value problem is:

$$\begin{aligned} \partial c/\partial t - k\Delta c + \mathbf{v}\cdot\nabla c &= 0, \\ c(0) &= c_0 \quad \text{on } \Omega, \\ \partial c/\partial n &= 0 \quad \text{on } \Gamma_N, \\ c &= 0 \quad \text{on } \Gamma_D \end{aligned} \tag{4.6}$$

We apply the standard Galerkin finite element approximation in space to the appropriate weak form of (4.6). Let U be the space of admissible solutions, U_h be the finite element subspace of U and ω_h be a test function from that space. Then the finite element discretization of the weak form is written as follows: find $c_h \in U_h$ such that

$$\int_\Omega \frac{\partial c_h}{\partial t}\omega \, dx + \int_\Omega k\nabla c_h \cdot \nabla\omega_h \, dx + \int_\Omega \omega_h \mathbf{v}\cdot\nabla c_h \, dx = 0 \quad \forall\, \omega_h \in U_h \tag{4.7}$$

The discrete convection diffusion equation can be written as a system of ODEs:

$$M\dot{\mathbf{c}} + K\mathbf{c} = 0 \tag{4.8}$$

where the mass matrix $M_{ij} = \int_\Omega \phi_i\phi_j dx$ and the stiffness matrix $K_{ij} = \int_\Omega (k\nabla\phi_i\cdot\nabla\phi_j + \phi_i\mathbf{v}\cdot\nabla\phi_j)\, dx$

For simulations with high Peclet number, stabilization of the Galerkin scheme is required. Therefore, the discretized weak form with the additional SUPG stabilization term, is

$$\begin{aligned} &\int_\Omega \frac{\partial c_h}{\partial t}\omega_h \, dx + \int_\Omega k\nabla c_h \cdot \nabla\omega_h \, dx + \int_\Omega \omega_h \mathbf{v}\cdot\nabla c_h \, dx \\ &+\Sigma_{\text{elem}} \int_{\Omega^e} \tau\left(\mathbf{v}\cdot\nabla\omega_h\right)\left(\frac{\partial c_h}{\partial t} - k\Delta c_h + \mathbf{v}\cdot\nabla c_h\right) dx = 0 \;\; \forall\, \omega_h \in U_h \end{aligned} \tag{4.9}$$

where τ is the SUPG constant [43]. Assuming linear shape functions, the mass matrix and stiffness matrix are modified to

$$M_{ij} = \int_\Omega (\phi_i \phi_j + \tau \mathbf{v} \cdot \nabla \phi_i \phi_j)\, dx \tag{4.10}$$

$$K_{ij} = \int_\Omega [k \nabla \phi_i \cdot \nabla \phi_j + \phi_i \mathbf{v} \cdot \nabla \phi_j + \tau (\mathbf{v} \cdot \nabla \phi_i)(\mathbf{v} \cdot \nabla \phi_j]\, dx \tag{4.11}$$

In time, we discretize using a Crank-Nicholson scheme, i.e.,

$$M \frac{\mathbf{c}^{k+1} - \mathbf{c}^k}{\Delta t} + K \frac{\mathbf{c}^{k+1} + \mathbf{c}^k}{2} = 0 \tag{4.12}$$

where the superscripts refer to time iteration steps. To compute the concentration at time step $k+1$, we solve:

$$\left[\frac{M}{\Delta t} + \frac{K}{2}\right] \mathbf{c}^{k+1} = \left[\frac{M}{\Delta t} - \frac{K}{2}\right] \mathbf{c}^k \tag{4.13}$$

or

$$M^{'} \mathbf{c}^{k+1} = K^{'} \mathbf{c}^k. \tag{4.14}$$

Hence, the discrete forward convection diffusion equation in matrix form is:

$$\begin{bmatrix} M^{'} & 0 & & & & & \\ -K^{'} & M^{'} & & & & & \\ & -K^{'} & M^{'} & & & & \\ & & & . & & & \\ & & & & . & & \\ & & & & & . & \\ & & & & & & \\ & & & & & -K^{'} & M^{'} \end{bmatrix} \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \\ . \\ . \\ . \\ . \\ \mathbf{c}_N \end{bmatrix} = \begin{bmatrix} K^{'}\mathbf{c}^0 \\ 0 \\ . \\ . \\ . \\ . \\ 0 \end{bmatrix} \tag{4.15}$$

We represent the above matrix with the following simplified notation:

$$A\mathbf{c} - T\mathbf{c}_0 = 0 \tag{4.16}$$

If N_n is number of nodes in space, and N_t is number of time steps, then dimensions of discrete operators and vectors are:

$$\begin{aligned} A &: (N_n * N_t) \times (N_n * N_t) \\ \mathbf{c} &: (N_n * N_t) \\ \mathbf{c}_0 &: (N_n) \\ T &: (N_n * N_t) \times N_n \end{aligned} \tag{4.17}$$

4.2.2 First Order Optimality Conditions and Inversion Equations

The inversion problem (4.1) can be written in discrete form as:

$$\begin{aligned} \min_{\mathbf{c},\mathbf{c}_0} \ & \frac{1}{2}[B(\mathbf{c} - \mathbf{c}^*)]^T [B(\mathbf{c} - \mathbf{c}^*)] + \frac{\beta}{2} \mathbf{c}_0^T R \mathbf{c}_0 \\ & \text{s.t. } A\mathbf{c} - T\mathbf{c}_0 = 0 \end{aligned} \tag{4.18}$$

where R is the discrete Laplacian (i.e. $R_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, dx$), and B is a Boolean matrix with 1's corresponding to sensor nodal locations.

To obtain first order optimality conditions we first form the Lagrange function

$$\mathcal{L}(\mathbf{c}, \mathbf{c}_0, \lambda) = \frac{1}{2}[B(\mathbf{c} - \mathbf{c}^*)]^T[B(\mathbf{c} - \mathbf{c}^*)]) + \frac{\beta}{2}\mathbf{c}_0 R\mathbf{c}_0 + \lambda^T(A\mathbf{c} - T\mathbf{c}_0) \tag{4.19}$$

We derive the first order optimality conditions for this function by taking derivatives with respect to $\mathbf{c}$, $\mathbf{c}_0$, and the adjoint variable λ, which results in the state equation (4.16), the adjoint equation:

$$A^T\lambda + B^TB(\mathbf{c} - \mathbf{c}^*) = 0 \tag{4.20}$$

and the inversion equation:

$$-T^T\lambda + \beta R\mathbf{c}_0 = 0 \tag{4.21}$$

In matrix notation, the KKT system is as follows:

$$\begin{bmatrix} B^TB & 0 & A^T \\ 0 & \beta R & -T^T \\ A & -T & 0 \end{bmatrix} \quad \begin{bmatrix} \mathbf{c} \\ \mathbf{c}_0 \\ \lambda \end{bmatrix} \quad = \begin{bmatrix} B^TB\mathbf{c}^* \\ 0 \\ 0 \end{bmatrix} \tag{4.22}$$

We follow two methods to solve this large coupled system: (1) an all-at-once or full space method, where the KKT system is solved simultaneously via a Krylov method as one large system, and (2) a reduced space method in which the equations are reduced to the initial concentration space and solved via conjugate gradients.

In the full space method, we use two approaches. The first is LNKS (see [36] for details). In this method, one needs to store state, adjoint and inversion variables. The success of this approach depends on the quality of the reduced space preconditioner. In the second full space approach, we reorder the equations and unknowns as follows:

$$\begin{bmatrix} A & 0 & -T \\ B^TB & A^T & 0 \\ 0 & -T^T & \beta R \end{bmatrix} \quad \begin{bmatrix} \mathbf{c} \\ \lambda \\ \mathbf{c}_0 \end{bmatrix} \quad = \begin{bmatrix} 0 \\ B^TB\mathbf{c}^* \\ 0 \end{bmatrix} \tag{4.23}$$

and we use a domain decomposition preconditioned Krylov method to solve this system. In this approach, one needs to store the entire KKT matrix to construct the preconditioner. Furthermore, the convergence behavior of this method deteriorates as the regularization parameter decreases due to the increasing ill-conditioning of the KKT matrix.

In the reduced space method, the solution procedure consists of first eliminating $\mathbf{c}$ from the state equation,

$$\mathbf{c} = -A^{-1}T\mathbf{c}_0 \tag{4.24}$$

then eliminating λ from the adjoint equation,

$$\lambda = -A^{-T}BA^{-1}T\mathbf{c}_0 - A^{-T}B\mathbf{c}^* \tag{4.25}$$

to finally obtain the reduced system for the inversion parameter, $\mathbf{c}_0$.

$$[T^TA^{-T}BA^{-1}T + \beta R]\mathbf{c}_0 = -T^TA^{-T}B\mathbf{c}^* \tag{4.26}$$

Since the coefficient matrix (the reduced Hessian) of this system is positive definite, we solve for the inversion parameter $\mathbf{c}_0$ using the conjugate gradient method. Each CG matrix-vector product requires one forward solve (with A) and one adjoint solve (with A^T). Only the inversion variable c_0 needs to be stored. The reduced Hessian is generally better conditioned then the KKT matrix in the full space method, and often no preconditioner is needed. The disadvantage of the reduced space method is the need to solve one forward and one adjoint equation at each CG iteration.

4.3 Implementation and Numerical Results

In this section, we present typical results for source inversion using the reduced space algorithm described above. We give inversion results for the transport of a contaminant within a rectangular geometry as well as over a synthetic terrain.

4.3.1 Transport of a Source Contaminant

We first show results of a forward simulation. A synthetic terrain, shown in Figure 4.1, was created using a set of eight random Gaussian bumps located in the problem domain $\Omega = \{(x, y, z) \,|0 \leq x \leq 10, 0 \leq y \leq 10, 0 \leq z \leq 5\}$. The domain was discretized into a logically rectangular grid of $20 \times 20 \times 10$ hexahedra, which were each decomposed into six linear tetrahedra.

An initial velocity field was computed for this domain by solving the steady-state incompressible Navier-Stokes equations. The inflow velocity boundary condition was set to $v_x = z^2$ along the plane $x = 0$. Traction-free boundary conditions were assumed at $x = 10$. On all other boundary surfaces, nonslip conditions were assumed normal to the boundary surface and traction-free boundary conditions were assumed tangential to the boundary surface.

An initial source distribution was centered at $\{2, 5, 0\}$ with an initial spatial concentration given as

$$c_0 = 10e^{-4.0\left((x-2)^2+(y-2)^2+z^2\right)} \tag{4.27}$$

Based on the precomputed velocity field, transport of the initial source concentration was modeled for $t \in [0, 20]$. The time interval was discretized into 100 time steps. The diffusion coefficient was taken as 0.05, resulting in a maximum element Peclet number of 10.

Figure 4.1 illustrates the transport of the source contamination over time. Due to the low velocities near the surface, the contaminant is primarily diffusing early in the simulation. However, as the contaminant diffuses vertically, the greater velocity convects it.

4.3.2 Source Inversion Results

Based on the forward transport of the contaminant described above, in this section inversion results. As before, our problem domain is discretized into a logically rectangular grid of $20 \times 20 \times 10$ hexahedra, which were each decomposed into six linear tetrahedra. We show results for both the synthetic terrain described above and for the case where there is no topography.

Our target source concentration is the same as for the forward problem. However, here we are going to consider the time interval of $t \in [0, 20]$ discretized into 20 time steps.

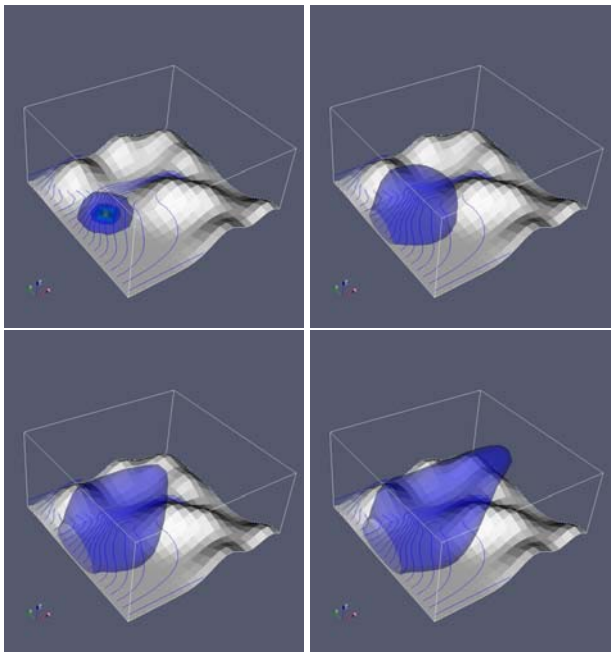


Figure 4.1. Transport of chemical over an 8 Gaussian synthetic landscape at initial time, 30, 60, and 90 timesteps.

4.3.2.1 Number of sensors

In Tables 4.1 and 4.2, the L_2 and ∞ norms of the "error" between and target initial concentration and inverted initial concentration, and the number of CG iterations to solve the system of inversion equations, are presented for the case of (1) no topography and (2) a synthetic terrain.

As expected, as the number of sensors located in the domain is increased, the predicted initial concentration more closely resembles the true initial concentration. Qualitatively, this is evident in Figure 4.2. Additionally, the system becomes better conditioned as evidenced by teh decreasing number of CG iterations.

Table 4.1. Comparison of CG iterations and error norms with varying number of sensors, when inverting for a Gaussian source concentration in a $20 \times 20 \times 10$ element rectangular domain.

# of sensors	$\|e\|_{L_2}$	$\|e\|_{\infty}$	iterations
$21 \times 21 \times 21$	0.0203	0.00354	42
$11 \times 11 \times 11$	0.0203	0.00354	42
$6 \times 6 \times 6$	9.02	2.53	91
$3 \times 3 \times 3$	37.94	4.49	124

Table 4.2. Comparison of CG iterations and error norms with varying number of sensors, when inverting for a Gaussian source concentration in a $20 \times 20 \times 10$ element domain with 8 Gaussian "mountains".

# of sensors	$\|e\|_{L_2}$	$\|e\|_{\infty}$	iterations
$21 \times 21 \times 21$	0.034	0.0041	58
$11 \times 11 \times 11$	0.034	0.0041	58
$6 \times 6 \times 6$	10.04	2.65	128
$3 \times 3 \times 3$	38.10	7.22	148

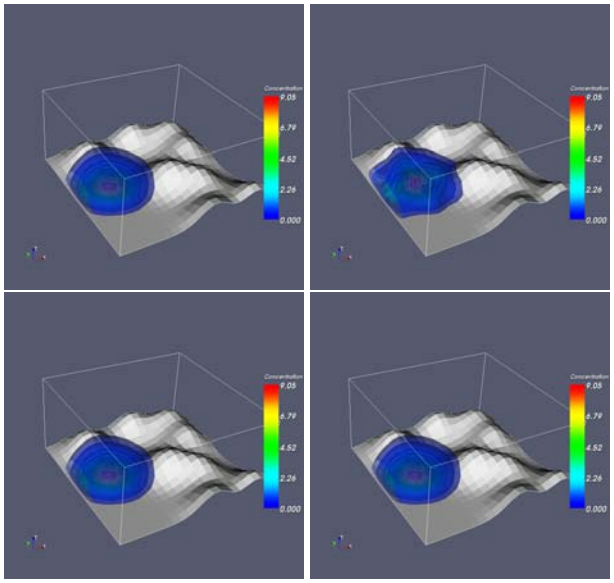


Figure 4.2. Inversion results: target concentration (top left), 6x6x6 sensor array inversion (top right), 11x11x11 sensor array inversion (bottom left), and 21x21x21 sensor array array (bottom right).

Table 4.3. Comparison of CG iterations and error norms with varying regularization parameter β, when inverting for a Gaussian source concentration in a $20 \times 20 \times 10$ element rectangular domain with a $6 \times 6 \times 6$ sensor array.

β	$\|e\|_{L_2}$	$\|e\|_{\infty}$	iterations
10^{-2}	9.68	2.42	47
10^{-3}	9.02	2.53	91
10^{-4}	8.34	2.36	185
10^{-5}	7.74	2.31	282

Table 4.4. Comparison of CG iterations and error norms with varying regularization parameter β, when inverting for a Gaussian source concentration in a $20 \times 20 \times 10$ element domain containing 8 Gaussian "mountains" and a $6 \times 6 \times 6$ sensor array.

β	$\|e\|_{L_2}$	$\|e\|_{\infty}$	iterations
10^{-2}	10.69	2.53	66
10^{-3}	10.04	2.65	128
10^{-4}	9.33	2.49	198
10^{-5}	8.73	2.44	328

4.3.2.2 Regularization Parameter

Regularization is required to eliminate the null space of the least squares inversion operator. As the regularization parameter β increases, the reduced Hessian becomes better conditioned, and the reduced system is easier to solve. However, the regularization modifies the range space as well, and therefore the inverted field is less accurate with increasing β. Tables 4.3 and 4.4 illustrate this for both the no topography and synthetic terrain domain.

4.4 Conclusions

We demonstrate the use of adjoint based methods on a transient inversion problem subject to convection-diffusion. We are interested in an inversion of initial conditions for a contamination event in a regional model and we assume that inversion parameters exist everywhere in the domain. The constrained optimization problem is reformulated as an unconstrained problem by forming a Lagrangian function. By taking derivatives of this function the KKT system results, which is solved by CG on a block-eliminated system in the reduced space of the inversion variable, i.e. the concentration field. The focus of this development has been on computational efficiency by using structured grids and matrix free implementations. We are able to solve the inversion problem for a reasonably sized dataset relatively efficiently. However, in order to scale up the problem so that we can apply this methodology to a large

regional dataset at fine resolution, we need an efficient preconditioning strategy for the solution of the reduced Hessian. As of the writing of this report, we have developed the basic interface for a multigrid approach and anticipate a successful implementation.

Chapter 5

Automatic Differentiation

5.1 Introduction

Sensitivity information is ultimately reliant on the fundamental derivative calculation. Here, our focus is on the differentiation of expressions implemented in the C++ programming language. There are many approaches available to implement derivative calculations, including symbolic methods, directional differentiation, complex step and automatic differentiation (AD). An appropriate calculation strategy depends on implementation issues, accuracy requirements, and the importance of computational efficiency. Symbolically derived analytical code provides the most accurate results, but can be time-consuming to derive, especially for complex physics. Furthermore, the implementation can be difficult and error prone. The directional finite-difference techniques are simpler to implement but result in less accurate derivative values. The selection of the perturbation step size is one of the fundamental problems associated with finite differencing and is difficult to set a priori. Setting the perturbation step too large or too small will exacerbate either truncation or round-off errors and no step size will remove these fundamental errors. Complex step finite differencing is a potentially implementation-efficient approach with results accurate to machine precision. Even though the implementation is relatively straightforward, involving the addition of a complex type, the disadvantage is in the redundancy of certain computations in comparison to automatic differentiation. Automatic differentiation can potentially provide the optimal combination of accuracy, implementation simplicity, and computational efficiency. Current source transformation tools are not quite as flexible as the template overloading libraries. The standard source transformation tools are somewhat sensitive to simulation code implementations and can generate a cumbersome development environment because of seperate recompilations to calculate the appropriate derivative values of new functions. Also, the computational efficiency of the transformed code depends entirely on the ability of the source transformation tools to be sensitive to the important hardware components. Obviously this is a difficult task and clearly not always achievable. Nevertheless, significant work has been done on tools that use source transformation [38, 100, 37] and can be very effective as a general or initial approach to calculating derivatives. An alternative strategy for applying AD in C++ based codes is to use template overloading. The template overloading stategy provides a mechanism to implement AD at various levels of complexity in functions. This removes certain code maintenance issues, provides machine precision derivative calculations, and most importantly provides an easy mechanism to control the level of intrusiveness of the AD calculation, which has implications to implementation effort and computational efficiency. The subject of this chapter is the use of hybrid strategies involving the application of AD at various levels of functional

complexity. We present the application of hybrid approaches to a relatively complicated function evaluation in gas dynamics.

We focus on the differentiation of vector functions of the form

$$f(x) \in \mathbf{R}^n \rightarrow \mathbf{R}^m \tag{5.1}$$

where it is assumed that $f(x)$ has at least one continuous derivative. Many different types of numerical algorithms, such as linear solvers, nonlinear solvers,and optimization methods, require the application of the Jacobian-vector product

$$\delta f = \frac{\partial f}{\partial x} \delta x \tag{5.2}$$

evaluated at a point x where $\delta x \in \mathbf{R}^n$ can be any particular vector. The application of the linear operator in (5.2) is required, for instance, in an iterative Krylov-Newton method for the solution of nonlinear equations of the form $f(x) = 0$ where $n = m$. The basic linear operator in (5.2) can be used directly as the operator application in a Krylov linear solver, such as GMRES, or can be used to generate an explicit matrix given the structure of the function $f(x)$. We generalize (5.2) to a form that involves the multiple simultaneous application of this linear operator for multiple right-hand sides

$$U = \frac{\partial f}{\partial x} S \tag{5.3}$$

where $S \in \mathbf{R}^{n \times p}$ and $U \in \mathbf{R}^{m \times p}$. Note that (5.3) can be used to generate the entire Jacobian matrix $\partial f / \partial x$ itself when $S = I$.

In this chapter we consider derivative computations of the form (5.2) and (5.3) for a broad class of ANSI C++ codes, in particular partial differential equation (PDE) based simulations. However, other models can also be considered such as differentiable algebraic equations (DAEs) based network simulators. Although we demonstrate certain derivate calculation strategies using a concrete example from compressible fluid dynamics, most of the statements and conclusions presented in this work are general and can be applied to a range of functions and numerical algorithms. The primary contribution of this work is the development of hybrid strategies that combines automatic and symbolic differentiation for complex functions to optimize the trade-off between implementation effort and the need for computational efficiency.

This chapter is organized as follows. Background information for various methods to differentiate ANSI C++ code is provided. Section 5.2 presents an overview of a large class of application areas where large-scale functions are assembled from a set of mostly independent "element" computations. This section uses the term "element" in the broadest sense and can be applied to PDE simulators as well as other types of models. A series of different levels of hybrid symbolic/AD methods for differentiation is defined. Section 5.4 provides a concrete example using compressible flow equations and presents numerical results that compare and contrast many of the different differentiation approaches. Finally, in Section 5.5 we give a number of conclusions and observations.

5.2 Background

There are a variety of methods that can be used to compute the linear operator in (5.2) for vector functions implemented in ANSI C++: 1) hand-coded symbolic derivatives (SD), 2) directional finite differences (FD), 3) complex-step (CS), and 4) automatic (or algorithmic) differentiation (AD). The main focus of this

work will be on the use of operator overloading methods for AD, but first other methods are reviewed so that the advantages of hybrid approaches can be fully appreciated.

The first method is referred to as symbolic differentiation (**SD**) and is based on symbolically deriving (5.2). The derivative expressions can either be derived and simplified by hand or by using tools such as Maple[1] or Mathematica[2]. This approach can yield very accurate and efficient derivative computations but can require a tremendous amount of manual labor and can be error prone. Even for moderately complicated functions it can be difficult to derive and implement the derivatives in a way that achieves good computer performance. However, because SD results in accurate analytic derivatives and because SD can be implemented potentially in a computational efficient manner (provided the target function is reduced to something manageable), this approach can be used in combination with automatic differentiation methods to produce excellent results, as discussed in later sections.

The second (and perhaps the most popular) method to compute an approximation to (5.2) is to use directional finite differencing (**FD**) of $x \rightarrow f(x)$. For example, a one-sided first-order finite difference approximation to (5.2) at a point x is given by

$$\delta f \approx \frac{f(x + \epsilon\delta x) - f(x)}{\epsilon} \tag{5.4}$$

where $\epsilon \in \mathbf{R}$ is the finite difference step length that must be selected to approximately minimize the sum of $O(\epsilon)$ truncation errors and roundoff cancellation errors. This approach requires minimal implementation effort because it depends only on the original function $x \rightarrow f(x)$ evaluation code, a single vector function evaluation $f(x + \epsilon\delta x)$ and several simple floating point operations. As a consequence of this simplicity, it is also a computationally efficient calculation. Higher-order finite-difference approximations can be used to reduce the truncation error. This allows larger finite difference step sizes ϵ to decrease roundoff error thereby reducing the overall approximation error. For example, the fourth-order central finite difference approximation

$$\delta f \approx \frac{f(x - 2\epsilon\delta x) - 8f(x - \epsilon\delta x) + 8f(x + \epsilon\delta x) - f(x + 2\epsilon\delta x))}{12\epsilon} \tag{5.5}$$

can be applied, yielding $O(\epsilon^4)$ truncation errors but at the cost of four evaluations of $x \rightarrow f(x)$. The disadvantages of finite-difference approaches are that i) it is difficult to select a priori the optimal finite difference step length ϵ such that the sum of truncation and roundoff errors are adequately minimized and ii) higher-order and therefore more accurate approximations, such as shown in (5.5), significantly increase the computational cost. In general, the unavoidable errors associated with the finite-difference approaches can result in poorly performing numerical methods.

A third approach to differentiate $x \rightarrow f(x)$ is the complex-step (**CS**) method. This method relies on the concept of *analytic extensions*, where the initially real f is extended into the complex plane in a neighborhood of x. The properties of analytic functions allows the approximation

$$\delta f \approx \frac{\text{Im}[f(x + i\epsilon\,\delta x)]}{\epsilon}, \tag{5.6}$$

where the right-hand side f denotes the extended function and Im denotes imaginary value. Note that there is no subtraction involved in this finite-difference approximation and thus no loss of significant digits in finite precision arithmetics. Approximation (5.6) requires f to be real analytic at x, which most operations performed on floating-point numbers in computer programs are with a few but important exceptions discussed below.

[1]Maple: `http://www.maplesoft.com`

[2]Mathematica: `http://www.wolfram.com`

In ANSI C++, the CS implementation amounts to replacing the floating-point real data type `double` with a floating-point complex data type using a type similar to `std::complex<double>`. Formula (5.6) with a very small ϵ, say $\epsilon = 10^{-20}$, will then provide very accurate derivative approximations free from the classical cancellation effect associated with finite differences. Therefore, in finite precision arithmetics, the CS method provides essentially exact derivative evaluations.

However, there are disadvantages associated with the use of the complex step method. First, complex arithmetics is significantly more expensive than real arithmetics. Second, the technique requires all operations involved in calculating $f(x)$ to be real analytic. The standard definition of the absolute value function $|z| = \sqrt{a^2 + b^2}$ for a complex number $z = a + ib$ is not analytic and not the analytic extension of the real absolute value. The analytic extension is $\sqrt{z^2}$, using the principal branch of the square root. Another complex extension of the real absolute value that is not the analytic, but one that also give the correct result using formula (5.6), is $\text{abs}(a + ib) = a + ib$ if $a \geq 0$ and $\text{abs}(a + ib) = -(a + ib)$ if $a \leq 0$. Another problem for the CS method are the relational operators, such as $<$ and $>$, which are not even defined for complex numbers (although can be defined by relations between associated real parts). For these reasons an existing (possibly optimized) C++ complex data type designed for complex arithmetic, such as the standard C++ data type `std::complex<>`, cannot be directly used for the CS method without making modifications to the underlying C++ code. The alternative, as advocated in [157], is to define a new C++ complex data type that properly defines the relational operators $<$ and $>$ and the absolute value function for use as a differentiation tool. The disadvantages of this approach is that an existing, possibly optimized, C++ complex data type cannot be used for the purpose of differentiation. The CS method, however, is similar in many ways to operator overloading based automatic differentiation. For this reason, the CS approach can be utilized as a simple verification of automatic differentiation calculations.

The fourth method for evaluating (5.2) is to use automatic differentiation (**AD**) [97]. It should be noted that automtatic differentiation is sometimes also referred to as algorithmic differentiation. The abstraction of automatic differentiation can be dated back to the development of code lists in the late 1950s and the early 1960s [30, 1, 215]. Considerable advancements have been made since then. Source transformation tools such as ADIFOR [37] and ADIC [38] have been the primary focus for fortran and C based codes. More recently, operator overloading methods using template expressions have been effectively applied to C++ codes [54].

AD exploits the chain rule and is based on the primitive rules for differentiation (e.g. $(a + b)' = a' + b'$, $(a - b)' = a' - b'$, $(ab)' = a'b + ab'$, $(a/b)' = (a'b - ab')/(b^2)$, $\sin(a)' = \cos(a)a'$ etc.). Therefore these methods are free of truncation and roundoff errors that plague FD methods. Only normal floating point roundoff errors are present in AD. In fact, it can be shown that the roundoff errors in the AD derivative computations are bounded by the roundoff errors involved in computing $x \rightarrow f(x)$ [97]. Therefore, a simple way to ensure that the AD derivatives are accurate is to ensure that the function evaluation is accurate. This is the first example of an important principle of AD which is "what is good for the function evaluation is also good for AD" [97].

AD can be performed in forward or reverse mode. The forward mode is the easiest to understand and is the most efficient for computing derivative objects of the form (5.2). The general process consist of decomposing the function into elemental steps, applying simple derivative rules to individual operations and then using the chain rule to provide a final derivative calculation of the overall function. The reverse mode of automatic differentiation was introduced by Linnainmaa [147] in 1976 and later used by Speelpenning [199]. It can be viewed as a topdown reconstruction of the code list and is equivalent to programming an adjoint sensitivity routine. Although the reverse mode of AD is an important capability, especially for computing gradients where n is large, there are currently very few tools available to perform

this type of calculation automatically; to the best of the authors knowledge there are no production quality C++ capabilities to perform reverse mode AD. As such, our efforts have focussed on the forward mode of AD. The forward mode of AD computes both the function value and Jacobian-vector products of the form (5.3) for one or more input or seed vectors. More specifically, forward-mode AD performs

$$(x, S) \rightarrow \left(f(x), \frac{\partial f}{\partial x} S \right) \tag{5.7}$$

where $S \in \mathbf{R}^{n \times p}$ as known as the seed matrix. This form of AD only computes the function value $f(x)$ once and uses it to propagate p different columns of sensitivities through the forward differentiation operator. One common choice for S is the $n \times n$ identity matrix I. When the seed matrix $S = I$ is used in the forward mode of AD, then the output is the Jacobian matrix $\partial f / \partial x$. When n and m are large, it is common that there is a lot of sparsity in the Jacobian $\partial f / \partial x$ and much work has been done to take advantage of this sparsity. Here we will only consider the direct use of the forward-mode of AD to explicitly form Jacobians when n and m are small and the Jacobians $\partial f / \partial x$ are fully dense. However, even in these smaller dimensional applications of forward-mode AD, it may still pay to take advantage of the structure of $f(x)$ when the seed matrix $S = I$ is used as many earlier derivative computations involving zeros can be avoided.

In general, there are two major approaches for implementing AD: source transformation and operator overloading. The source-transformation approach involves a compiler-like tool that reads an entire piece of source code implementing a function to be differentiated. The compiler-like tool parses the code, performs various analyzes and then writes a new set of source files defining new functions that implement the forward (and perhaps in some cases also the reverse) mode of AD. The operator overloading approach for forward AD is to use a derived abstract data-type that defines all of the operators of a floating point scalar but also carries along one or more derivative components. More specifically, one derivative component is carried along for each column in the seed matrix S in (5.7). The operator overloading approach performs elementary derivative computations in addition to the elementary function evaluation computations. A straightforward implementation of forward-mode AD using operator overloading handles piecewise defined functions by being able to easily navigate through conditional branches (i.e. `if` statements), and when combined with C++ function overloading, this approach can also be used to define rules to differentiate through non-C++ or third-party function calls.

Several different C++ classes use operator overloading to implement the forward mode of AD[3]. Although these classes use different approaches, we focus on the forward AD (Fad) suite of C++ classes [54]. These methods are templated on the scalar type and therefore allow great flexibility in their use (for example creating nested AD types to compute second and higher-order derivatives). The Fad classes use a C++ template programming technique called *expression templates* which results in object code that eliminates many of the temporary objects commonly created by operator overloading in C++. One of the classes (`TFad`) is also templated on the derivative components and therefore does not impose any runtime dynamic memory allocation when it is used.

As pointed out in [157], the CS and operator-overloading AD approaches have a lot in common with respect to computing the single vector right-hand side form of (5.2). They both use operator overloading and both carry two floating-point values (the function value and the single derivative component) with each scalar variable. Both can produce essentially analytic derivative objects. However, AD methods are more computationally efficient for a number of reasons. First, the CS method performs unnecessary floating

[3] `http://www.autodiff.org`

point operations in some cases. For example, consider the CS multiplication operation

$$(a + ia')(b + ib') = (ab - a'b') + i(ab' + a'b).$$

Here $i(ab' + a'b)$ carries the derivative component and $ab \approx ab - a'b'$ is essentially the value because $ab \gg a'b'$. The elementary multiplication and subtraction of $a'b'$ is unnecessary. AD does not perform these types of unnecessary computations and therefore is more efficient. Second, specially designed AD types, such as `TFad<>`, can carry multiple derivative components instead of just one which makes the computation of (5.3) for $p > 1$ right-hand sides more efficient since the function value $f(x)$ is only computed once and amortized over the p multiple derivative components. This is not the case for the CS method which must compute the function value over and over again for each of the p columns of S. Third, the use of multi-component AD types can result in better memory usage (i.e. cache) performance since the expressions for $f(x)$ are only moved through one time. However, in the CS method the function $f(x)$ must be evaluated independently for each derivative component p.

In addition to the performance issues, there are also some implementation differences. An off-the-shelf complex data type such as `std::complex<>` cannot be used without modifying underlying C++ source code for the logical operators $>$, $<$ or the absolute value function $|.|$. Another significant disadvantage of the CS method is that if $f(x)$ already uses complex arithmetic then it is unclear if the CS method can be used even through nested types such as `std::complex<std::complex<> >` are potentially allowed.

Based on the previously described points, AD should always be preferred over the CS approach in production ANSI C++ code. Given this conclusion, can AD however be improved upon by considering a hybrid approach? To answer that question, we first describe the relationship between the SD and AD methods. Given enough developer time and effort, SD approaches should always be more computationally efficient in memory and CPU time than any AD approach. This is because SD can simply mimic what AD does performing exactly the same operations and throw away unnecessary computations. Forward-mode AD always computes the function value in addition to the derivatives as shown in (5.7) even though this function value is usually thrown away (i.e. since it is already known). In the extreme case where all of the operations implementing $f(x)$ are linear (i.e. only $+$ and $-$), AD performs twice as many flops as a simple SD approach since the derivative computations such as $(a + b)' = a' + b'$ and $(a - b)' = a' - b'$ never need the values a or b. If the majority of operations in a particular function $f(x)$ are linear, then significant savings can be achieved by specializing the computations for the case where the final function value is not needed, only the derivatives. For example, consider the simple function in Figure 5.1. When performing forward AD on this function, only the value for the first intermediate variable $x_5 = x_1 x_2$ is needed to compute the derivative for $x'_6 = x'_5 x_4 + x_5 x'_4$ and the values for all of the other intermediate variables x_i, for $i = 6 \ldots 10$, can be ignored since they are not used in any later derivative computations. In this example, a simple SD approach can result in nearly half the floating point operations as the basic AD approach.

For general functions $f(x)$, specialized SD approaches can be very difficult to implement and maintain (as the function $f(x)$ changes due to new requirements) but in some specialized cases, such as discretization methods for PDE, the high-level structure of $f(x)$ can be exploited fairly easily. Furthermore, this high-level structure generally does not change significantly once the discretization method has been adopted. Such specialized vector functions $f(x)$ have a regular hierarchical structure with many different layers and, in these cases, one can symbolically differentiate one level at a time, going deeper and deeper until sufficient performance is achieved. Any code below the current symbolically differentiated level can of course be easily and effectively differentiated automatically using the operator overloading method. The ratio of SD to AD code depends on a trade-off between developer time and computational performance.

Input:

x_1, x_2, x_3, x_4

Computation:

$x_5 = x_1 x_2,\ x_6 = x_5 x_4,\ x_7 = x_6 - x_1,\ x_8 = x_7 + x_2,\ x_9 = x_8 - x_3,\ x_{10} = x_9 + x_4$

Output:

$f = x_{10}$

Figure 5.1. Example of a simple function $f(x) \in \mathbf{R}^4 \rightarrow \mathbf{R}$ with nonlinear operations up front followed by all linear operations.

5.3 Hybrid symbolic/AD approaches for element-based assembly models

We describe a common model for the assembly of vector functions of the form (5.1). This general assembly encompasses many different application areas from various discretization methods for PDEs (i.e. finite-element, finite-volume, discontinuous Galerkin etc.) to network models (i.e. electical devices, utility networks, processing plants etc.). The goal of this section is to describe levels of intrusive symbolic differentiation combined with automated methods that can be used to efficiently compute products such as (5.2). We start with an abstract model (shown in Algorithm 2) to establish notation and general methodology in preparation of discussing more complicated issues.

Algorithm 2. GENERIC ELEMENT-BASED ASSEMBLY MODEL OF THE GLOBAL FUNCTION $f(x)$

$x \rightarrow f(x)$

1. *Initialization*
   $f = 0$
2. *Element assembly*

   for $e = 1 \ldots N_e$
   Gather local variables
   $x_e = P_{x,e}\, x \in \mathbf{R}^{n_e}$
   Local computation
   $f_e = f_e(x_e) \in \mathbf{R}^{m_e}$
   Global scatter and assembly
   $f = f + P_{f,e}^T\, f_e$

The assembly model in Algorithm 2 shows a summation assembly of independent element computations. Here we use the term element in a most general sense that is not restricted to just finite-element methods. For each element, a relatively small number of variables are gathered for the local computation by the operation $x_e = P_{x,e}\, x$. This "local" vector of variables x_e is then used in a "local", relatively compact, element computation $x_e \rightarrow f(x_e)$ to form $f_e \in \mathbf{R}^{m_e}$. In general $n_e \ll n$ and $m_e \ll m$. Finally, for each element, the local function f_e is assembled into the global function as $f = f + P_{f,e}^T\, f_e$.

This assembly model uses abstractions of mapping matrices $P_{x,e}$, and $P_{f,e}$ to represent the indexing of local/global variables and functions respectively. These matrices contain only columns of identity (i.e. each row and column contains only one nonzero element) or zeros depending on the formulation of the problem. The conversion used here for a local/global mapping matrix P is for the non-transposed linear operator P to perform global-to-local mappings as

$$v_l = P v_g \tag{5.8}$$

and for the transposed operator P^T to perform the local-to-global mappings as

$$v_g = P^T v_l. \tag{5.9}$$

The element functions $f_e(x_e)$ $(e = 1 \dots N_e)$ may represent the physics computation and the discretization method of a PDE. Each computation can be processed independently and in parallel. In the case of a PDE discretization, the element functions $f_e(x_e)$ would represent either internal or boundary contributions. Obviously, these calculations vary greatly depending on the discretization method. Therefore, for a finite-element PDE method, the element loop in Algorithm 2 would involve both traditional internal element loops and one or more boundary loops. Typically, more data than just x_e is needed to define the functions $f_e(\dots)$ and it is assumed that this data is encapsulated in the mathematical functions themselves. Therefore, each function $f_e(\dots)$ may have different behavior based on the index e. In an actual implementation, extra data will undoubtedly be required in which case one may have the function $f_e(x_e, d_e)$ where d_e is a vector of extra data associated with each element e. This extra data d_e is generally considered "passive" data that does not define independent variables for which to differentiate through.

The assembly in Algorithm 2 can be compactly written as

$$f(x) = \sum_{e=1}^{N_e} P_{f,e}^T f_e(P_{x,e} x). \tag{5.10}$$

This form will be useful in deriving the the various global derivative objects in the following section.

5.3.1 Global derivative computations

In this section, we consider how local element-wise derivative computations can be used with the element assembly model in Algorithm 2 to compute the following global derivative objects:

1. *Sparse Jacobian matrix*:
$$J = \frac{\partial f}{\partial x} \tag{5.11}$$
2. *Jacobian-vector product*:
$$\delta f = \frac{\partial f}{\partial x} \delta x \tag{5.12}$$

These are the primary computations needed for a number of numerical algorithms including linear and nonlinear equation solvers, stability analysis methods, uncertainty quantification, and optimization. From the assembly model in (5.10), the Jacobian matrix is given by

$$\frac{\partial f}{\partial x} = \sum_{e=1}^{N_e} P_{f,e}^T \frac{\partial f_e}{\partial x_e} P_{x,e}. \tag{5.13}$$

Given the above expression for the Jacobian $\partial f/\partial x$, it is straightforward to see how to assemble the above desired global derivative objects.

5.3.2 Sparse Jacobian matrix and Jacobian-vector assembly

The assembly of the sparse Jacobian matrix $J = \partial f/\partial x$ follows directly from (5.13) and is given in Algorithm 3.

Algorithm 3. ASSEMBLY OF THE GLOBAL JACOBIAN MATRIX

$(y, u) \rightarrow J = \frac{\partial f}{\partial x}$

1. *Initialization*
   $J = 0$
2. *Element assembly*

   for $e = 1 \ldots N_e$
   $J = J + P_{f,e}^T \frac{\partial f_e}{\partial x_e} P_{x,e}$

The above algorithm requires that the element Jacobians $\partial f_e/\partial x_e$ be explicitly computed. The mapping matrices $P_{x,e}$ and $P_{f,b}$ simply define how the local element Jacobians $\partial f_e/\partial x_e$ are scattered and added into the global sparse Jacobian matrix J. Interfaces supporting this type of mapping are common in many different types of codes. Also note that from now on it is assumed the the element-level Jacobians $\partial f_e/\partial x_e$ are computed at the current x_e which should be obvious from the context.

The assembly of a Jacobian-vector product (5.2) follows directly from (5.13) as

$$\begin{aligned} \delta f &= \frac{\partial f}{\partial x}\delta x \\ &= \sum_{e=1}^{N_e} P_{f,e}^T \frac{\partial f_e}{\partial x_e} P_{x,e}\delta x. \end{aligned} \tag{5.14}$$

and is restated in Algorithm 4.

Algorithm 4. ASSEMBLY OF THE GLOBAL JACOBIAN-VECTOR PRODUCT

$(x, \delta x) \rightarrow \delta f = \frac{\partial f}{\partial x}\delta x$

1. *Initialization*
   $\delta f = 0$
2. *Element assembly*

for $e = 1 \ldots N_e$
$\quad \delta x_e = P_{x,e} \delta x$
$\quad \delta f_e = \frac{\partial f_e}{\partial x_e} \delta x_e$
$\quad \delta f = \delta f + P_{f,e}^T \delta f_e$

5.3.3 Precomputed-storage and storage-free approaches

Note that the forward assembly Algorithm 4 can be used in one of two ways. The first general class will be referred to as **precomputed-storage** approaches. These methods involve computing the Jacobian matrices $\partial f_e / \partial x_e$ upfront in a single loop and storing these element matrices in an element-wise data structure. Then the precomputed $\partial f_e / \partial x_e$ matrices can be used to assemble Jacobian-vector products in Algorithm 4 which involves local matrix-vector products. These repeated assembly loops only utilize the precomputed $\partial f_e / \partial x_e$ matrices and therefore do not touch the actual element functions $f_e(x_e)$ themselves. The main advantage of these approaches are that they can result in dramatic reductions in cost of repeated Jacobian-vector products. The disadvantages of these approaches are that some potentially expensive upfront AD computations are needed to compute the element Jacobian matrices and that these local matrices can require significant storage.

The second general class will be referred to as **storage-free** approaches. These methods do not involve any upfront computation or storage and instead just require applications of forward AD with $f_e(x_e)$ to compute Jacobian-vector products. The main advantages of these approaches are that no upfront computation or storage is needed. The disadvantage of these approaches over *precomputed-storage* approaches is that the formation of each product assembly is more expensive since the element functions $f_e(x_e)$ must be called repeatedly using AD datatypes.

5.3.4 Element derivative computations

The global derivative assembly computations described in the preceding sections require the following types of element-wise derivative computations:

1. *Element Jacobian matrix*:
$$J_e = \frac{\partial f_e}{\partial x_e} \in \mathbf{R}^{m_e \times n_e} \tag{5.15}$$
2. *Element Jacobian-vector product*:
$$\delta f_e = \frac{\partial f_e}{\partial x_e} \delta x_e \in \mathbf{R}^{m_e} \tag{5.16}$$

Note that only *storage-free* approaches require directional derivatives (5.16) for Algorithm 4.

As mentioned previously, we assume that the dimensions of each element computation n_e and m_e are relatively small (i.e. $O(100)$ for less) and that the mapping from the input element variables x_e to the output functions $f_e = f_e(x_e)$ is fairly dense (i.e. $\partial f_e / \partial x_e$ has few structurally zero entries). This is generally true for many different types of applications but there are some exceptions such as PDE problems involving many different chemical species with reactions. Given that we are assuming that n_e and m_e are

relatively small and $x_e \rightarrow f_e(x_e)$ is fairly dense, these element-wise derivatives can be computed in a variety of ways. One of the most efficient and simplest of approaches is to use AD. In C++ this is especially easy if the functions $f_e(x_e)$ (or the classes that implement these functions) are templated on the scalar types for the input arguments x_e and the output vector arguments for f_e. For example, suppose the following non-member C++ function computes $f_e(x_e, d_e)$:

```
void eval_ele_func(
  const double       x_e[],
  const ElementData  &d_e,
  double             f_e[]
  );
```

where `d_e` is an object that defines the rest of the element-specific data (e.g. nodal coordinates etc. for a PDE discretization) for the element computation. To facilitate the use of automatic differentiation, the above function can be templated as follows:

```
template<Scalar_x_e, Scalar_f_e>
void eval_ele_func(
  const Scalar_x_e   x_e[],
  const ElementData  &d_e,
  Scalar_f_e         f_e[]
  );
```

Dense Jacobians and forward Jacobian-vector products can be efficiently computed using the forward mode of AD and easily implemented using the small templated class `TFad<N,T>`. This type is templated both on the underlying scalar type `T` and the number of derivative components `N`. In templating by the number of derivative components, all memory allocation can be performed on the stack and therefore greatly speed up compute times. Using `TFad` is as simple as instantiating the code for $f_e(\ldots)$ using `TFad`, initializing the input independent `TFad` variables appropriately and then running the `TFad`-enabled function. On output, the desired derivatives are extracted from the output arguments. By using only one derivative component (i.e. `TFad<1,double>`), a Jacobian-vector product in (5.16) can be cheaply computed at a cost of less than twice the storage and less than five times the flops of the function evaluation. However, generating a Jacobian with respect the `N` variables requires using `N` derivative components (i.e. `TFad<N,double>`) and the resulting computation will, in theory, require between `2*N` and `5*N` more flops than the original function evaluation. However, certain types of operations, like square roots, can reduce the relative cost of each derivative component.

The following C++ functions give examples of the use of `TFad` for computing Jacbian-vector products (5.16) and Jacobian matrices (5.15) for the templated function `eval_ele_func(...)`.

```
//
// Compute an element Jacobian-vector product
//
void eval_ele_jac_vec(
  const double       x_e[],
  const ElementData  &d_e,
  const double       delta_x_e[],
  double             delta_f_e[]
  )
```

```
{
  const int N_x_e = 10; // Number components in x_e[]
  const int N_f_e = 5;  // Number components in f_e[]
  // Initialize AD argumets
  TFad<1,double> ad_x_e[N_x_e], ad_f_e[N_f_e];
  for( int k = 0; k < N_y_e; ++k ) {
    ad_x_e[k].val() = x_e[k];                  // Set indepenent var values
    ad_x_e[k].fastAccesDx(0) = delta_x_e[k];   // Load multiplying vector
  }
  // Run function in forward mode
  eval_ele_func( ad_x_e, d_e, ad_f_e );
  // Extract state Jacobian-vector product
  for( int k = 0; k < N_f_e; ++k ) {
    delta_f_e[k] = ad_f_e[k].fastAccessDx(0);
  }
}

//
// Compute an ''element'' (column-major) Jacobian matrix
//
void eval_ele_state_jac(
  const double       x_e[],
  const ElementData  &d_e,
  double             J_e[]
  )
{
  const int N_x_e = 10; // Number components in x_e[]
  const int N_f_e = 5;  // Number components in f_e[]
  // Initialize AD argumets
  TFad<N_x_e,double> ad_x_e[N_x_e];
  TFad<N_x_e,double> ad_f_e[N_f_e];
  for( in k = 0; k < N_x_e; ++k ) {
    ad_x_e[k].val() = x_e[k];           // Set independent var values
    ad_x_e[k].diff(k);                  // Mark independent vars
  }
  // Run function in forward mode to compute the entire state Jacobian
  eval_ele_func( ad_x_e, d_e, ad_f_e );
  // Extract state Jacobian matrix in column-major format
  for( in k1 = 0; k1 < N_f_e; ++k1 ) for( in k2 = 0; k2 < N_y_e; ++k2 )
    J_y_e[k1+k2*N_f_e] = ad_f_e[k1].fastAccessDx(k2);
}
```

As shown above code examples, using `TFad` is straightforward provided the number of independent variables is known at compile time. If the number of independent variables is not known at compile time then the (slightly) less efficient class `Fad` can be used instead. (Note that the while element derivative computations described above only used AD at the element level, these derivative computations can also be performed using a hybrid symbolic/AD approach.

5.3.5 Levels of hybrid symbolic/AD differentiation

We introduce a classification system of levels for hybrid differentiation of functions that exhibit hierarchal dependencies. Complicated models for $f(x)$ will have many computational dependencies where higher level calculations depend on lower level calculations. Hybrid differentation is based on that structure by selecting the appropriate amount of symbolic differentiation combined with AD for the remaining lower levels. Each level of this hierarchal structure defines a different possible level of a hybrid symbolic/AD differentiation method. Here we focus on the types of element assembly models shown in Algorithm 2 and focus on the different levels of symbolic/AD approaches for assembling $\delta x \rightarrow (\partial f/\partial x)\delta x$. For each level of hybrid method, higher-level expressions are symbolically differentiated and then all remaining lower-level expressions are differentiated using AD. These levels are described next and provide a classification of our hybrid differentiation strategy for systematic performance comparisons of our upcoming concrete example discussions.

Level 0 denotes the application of directional AD to the global C++ function that computes $x \rightarrow f(x)$ without any concern to underlying structure. This is the least intrusive level and therefore the easiest to use, requiring no knowledge of the structure of $f(x)$. The disadvantage of this level is the requirement that the entire function computing $x \rightarrow f(x)$ be templated on the scalar type which is not practical for certain code frameworks. Only *storage-free* methods are used with this most basic level.

Level 1 denotes the application of AD to the element level functions $f_e(x_e)$. Both the *storage-free* and *precomputed-storage* methods can be used for this level. In the case of the *storage-free* approaches, directional AD is applied at the element assembly level to compute $x_e \rightarrow (\partial f_e/\partial x_e)\delta x_e$, followed by an assembly of the global Jacobian-vector product as shown in Algorithm 4. In the case of *precomputed-storage* approaches, the seed matrix is set to identity $S = I$ and the element Jacobians $\partial f_e/\partial x_e$ are precomputed using the forward mode of AD applied to $f_e(x_e)$. These precomputed element Jacobian matrices are then used to assemble subsequent Jacobian-vector products. The main advantage of this level is that it requires no knowledge of the underlying structure of the expressions or mathematical problem and only involves manipulating the basic assembly process . In addition, the entire C++ function $x \rightarrow f(x)$ does not need to be templated on the scalar type; only the element functions for $f_e(x_e)$ need be templated on the scalar type.

Levels 2 and higher denotes symbolically differentiating into the element-level expressions $f_e(x_e)$ and applying AD on the remaining lower levels. Level 2 may only involve the least-invasive amount of symbolic differentiation that only symbolically differentiates the highest-level expressions in $f_e(x_e)$. Levels 3 and higher may involve symbolically differentiating deeper and deeper into the expressions for $f_e(x_e)$. The number of meaningful levels that are possible depends on the nature of the expressions for $f_e(x_e)$. These higher-level hybrid symbolic/AD methods may be aplied using *precomputed-storage* or *storage-free* approaches. Note that *precomputed-storage* approaches may not directly store the element Jacobians $\partial f_e/\partial x_e$ but instead may only store contributions to these Jacobians that may result in less overall storage and higher performance; The *level-2* approaches described in Section 5.4 are an example of this.

In general, for each increasing level of hybrid symbolic/AD method, the amount of developer work needed to implement and maintain the method will increase. However, higher-level, more invasive methods offer potential to greatly reduced storage (i.e. for *precomputed-storage* methods) and runtime CPU time (i.e. for *precomputed-storage* and *storage-free*) methods.

Section 5.4 describes a concrete example for the use of several different levels of *precomputed-storage* and

storage-free symbolic/AD approaches and demonstrates the potential improvements in storage cost and runtime performance for increasing levels of hybrid symbolic/AD methods.

5.4 Finite-volume discretization example

We present a concrete case study of various differentiation techniques and the application of different levels of hybrid symbolic/AD approaches. The particular application we present here is a finite-volume method for the Euler equations of gas dynamics which represents an example of a two-loop element assembly model. The single loop of independent elements shown in Algorithm 2 accurately applies for a surprisingly large number of application areas such as many finite-element methods for PDEs and many different types of network models. However, a single independent loop over a single set of elements is not sufficient in other areas and auxiliary variables and element loops must be added. One instance of this is the finite-volume example described in this section where two or more coupled loops are required.

We take a top-down approach to presenting the hybrid differentiation of the Euler equation example. We assume very little about the Euler equations and start the discussion with level 0. As additional levels of differentiation are considered, more details of the Euler equations will be revealed and discussed. This approach will demonstrate the utility of the hybrid strategy and provide an optimal approach to balancing implementation effort versus computational efficiency. In addition, we ignore many details of the discretization, and provide only enough algorithmic information to describe the use of the different differentiation techniques. For additional details, we refer to the general CFD literature [40].

5.4.1 The basic residual function and level-0 hybrid symbolic/AD differentiation

At the center of this finite-volume CFD code is the computation of the *residual function*

$$\mathbf{w} \rightarrow \mathbf{r}(\mathbf{w}) \in \mathbf{R}^{(n_d+2)n_N} \rightarrow \mathbf{R}^{(n_d+2)n_N} \tag{5.17}$$

where n_d is the spatial dimension ($n_d = 2$ or $n_d = 3$), n_N is the number of nodes in the computational mesh, and $\mathbf{w} \in \mathbf{R}^{(n_d+2)n_N}$ is a vector of fluid states (density, velocity, pressure) in so-called conservative form. Here we have changed the nomenclature of the global variables and global function from $x \rightarrow f(x)$ to $\mathbf{w} \rightarrow \mathbf{r}(\mathbf{w})$ in order to match previous notation.

Our C++ prototype code contains a function that implements (5.17), and this function is templated on the scalar types of each of the input vectors. It should also be noted that more data than just the vector of state variables $\mathbf{w}$ is passed into this C++ function but for the purposes of the application of *level-0* AD, this data is not important. Templating this C++ function on the scalar type for $\mathbf{w}$ facilitates the straightforward use of the AD type `TFad` at all levels of computation. While several differentiation calculations are needed for gas dynamics, we focus on the computation of the state Jacobian-vector product

$$\delta\mathbf{r} = \frac{\partial \mathbf{r}}{\partial \mathbf{w}} \delta\mathbf{w} \tag{5.18}$$

which is needed by various solution strategies for $\mathbf{r}(\mathbf{w}) = 0$ for example.

As previously mentioned, we take a top-down approach to describing this finite-volume code with the intent of highlighting our use of hybrid symbolic/AD methods. We start with the highest-level description of the

functions, shown as the high-level vector function (5.17), and the least-invasive level of AD, that is, *level-0* which applies AD to the entire residual evaluation code. We then symbolically differentiate deeper into the code and discuss the impact on storage and runtime for repeated derivative computations of the form (5.2). However, we stop using symbolic differentiation when we reach the most complicated algebraic expression in the code, and in this case we stop at the calculation of the artificial dissipation term of Roe type.

The residual evaluation code for (5.17) is templated on the scalar type. Thus, if we can assume that the function is first-order differentiable, then we do not need to know how the function (5.17) is evaluated in order to compute (5.18) using the forward mode of AD. The templated residual evaluation function with the scalar type `TFad<1,double>` for the input vector $\mathbf{w}$ and the output vector $\mathbf{r}$ only needs to be instantiated in order to compute (5.18) using the forward mode of AD. All of the other data types for the passive input data (which are inconsequential for a *level-0* method) are left as `double`. This selective template instantiation prevents wasted derivative computations for inactive input data.

In the following sections, we symbolically differentiate into the residual evaluation (5.17). For each level of the symbolic differentiation process, we describe the computations inside of (5.17) in only enough detail to be able to apply AD strategically to the lower levels.

5.4.2 Two-loop edge-based residual assembly

Implementing a basic *level-1* hybrid symbolic/AD method requires knowing something about the discritization method since this defines the element assembly process in Algorithm 2. The finite-volume residual assembly consists of two loops over all edges in the computational mesh. In a finite-volume discretization, a control volume is created around a mesh node where all the physics calculations are performed. A control volume consists of edges, which is considered the basic element of this discretization method. The finite-volume discretization described here uses an unstructured meshing—with tetrahedra or hexahedra—of the domain Ω occupied by the gas. We denote by $\mathcal{V}(\Omega)$ the set of mesh nodes in the strict interior of the domain and by $\mathcal{V}(\partial\Omega)$ the nodes on the boundary of the mesh. Thus, the set of all nodes is $\mathcal{V}(\overline{\Omega}) = \mathcal{V}(\Omega) \cup \mathcal{V}(\partial\Omega)$. Moreover, let $\mathcal{N}_i$ denote the set of nodes being *nearest neighbors* to mesh node i, that is, the nodes that are connected to node i with an edge. We associate a surface vector $\boldsymbol{n}_{ij}$ with each directed edge $\xrightarrow{i\,j}$ in the mesh. The normals are computed from a so-called dual mesh [40], and by construction,

$$\boldsymbol{n}_{ji} = -\boldsymbol{n}_{ij}. \tag{5.19}$$

Note that the normals are not of unit length and therefore embody both direction and magnitude. (The magnitude represents a part of the surface area of the finite volume surrounding each node point.) The residual in (5.17) is assembled in two edge-based loops as shown in Algorithm 5.

Algorithm 5. Two-loop edge-based residual assembly

Compute the residual vector $\mathbf{r}$ given the input vectors $\mathbf{w}$, $\boldsymbol{x}$, $\boldsymbol{n}$ and V.

 # Spatial gradient assembly

1. Set $\mathbf{g}_i \leftarrow \mathbf{0}$ for each $i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{i\,j}$:

(a) $\mathbf{g}_i \leftarrow \mathbf{g}_i + \hat{\mathbf{g}}(V_i, \mathbf{w}_i, \mathbf{w}_j, +\boldsymbol{n}_{ij})$

(b) $\mathbf{g}_j \leftarrow \mathbf{g}_j + \hat{\mathbf{g}}(V_j, \mathbf{w}_j, \mathbf{w}_i, -\boldsymbol{n}_{ij})$

3. + Boundary contributions

 # Residual assembly

1. Set $\mathbf{r}_i \leftarrow \mathbf{0}$ for each $i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{i\,j}$:

(a) $\hat{\mathbf{r}}_{ij} = \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}_i, \mathbf{g}_j, \boldsymbol{n}_{ij})$

(b) $\mathbf{r}_i \leftarrow \mathbf{r}_i + \hat{\mathbf{r}}_{ij}$

(c) $\mathbf{r}_j \leftarrow \mathbf{r}_j - \hat{\mathbf{r}}_{ij}$

3. + Boundary contributions

A single templated C++ function `assemble_grad(...)` is called for each edge in Algorithm 5 to simultaneously compute $\hat{\mathbf{g}}(V_i, \mathbf{w}_i, \mathbf{w}_j, +\boldsymbol{n}_{ij})$ and $\hat{\mathbf{g}}(V_j, \mathbf{w}_j, \mathbf{w}_i, -\boldsymbol{n}_{ij})$ in which some of the same computations are shared. A single templated C++ function `assemble_resid(...)` is called for each edge in Algorithm 5 to compute $\hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}_i, \mathbf{g}_j, \boldsymbol{n}_{ij})$. While the treatment of boundary conditions is critical to any discretization method, the computation usually does not contribute significantly to the computational cost of the state residual (5.17). Therefore, for the purposes of this discussion, we will gloss over details of boundary conditions, since our goal is to address the bulk computational work. Enforcement of boundary conditions would represent one or more element loops that would contribute to the residual assembly.

5.4.3 State Jacobian-vector product assembly

By inspecting the edge-based assembly in Algorithm 5 and using the multi-variable chain rule, the edge-based assembly loops for the Jacobian-vector product can be derived as shown in Algorithm 6.

Algorithm 6. Level-1 hybrid symbolic/AD Jacobian-vector product assembly

---

Compute $\delta\mathbf{r} = \dfrac{\partial \mathbf{r}}{\partial \mathbf{w}} \delta\mathbf{w}$ given input vector $\delta\mathbf{w}$

 # Linearized spatial gradient assembly

1. Set $\delta\mathbf{g}_i \leftarrow \mathbf{0}$ for each $i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{i\,j}$:

(a) $\delta\mathbf{g}_i \leftarrow \delta\mathbf{g}_i + \dfrac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_i}\delta\mathbf{w}_i + \dfrac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_j}\delta\mathbf{w}_j$

(b) $\delta \mathbf{g}_j \leftarrow \delta \mathbf{g}_j + \dfrac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_i}\delta \mathbf{w}_i + \dfrac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_j}\delta \mathbf{w}_j$

3. + Boundary contributions

 # Linearized residual assembly

1. Set $\delta \mathbf{r}_i \leftarrow \mathbf{0}$ for each $i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\overset{i\,j}{\longrightarrow}$:
   (a) $\delta \hat{\mathbf{r}}_{ij} = \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_i}\delta \mathbf{w}_i + \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j}\delta \mathbf{w}_j + \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_i}\delta \mathbf{g}_i + \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_j}\delta \mathbf{g}_j$
   (b) $\delta \mathbf{r}_i \leftarrow \delta \mathbf{r}_i + \delta \hat{\mathbf{r}}_{ij}$
   (c) $\delta \mathbf{r}_j \leftarrow \delta \mathbf{r}_j - \delta \hat{\mathbf{r}}_{ij}$
3. + Boundary contributions

In the above algorithm, we use the notation

$$\frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_i} = \frac{\partial}{\partial \mathbf{w}}\hat{\mathbf{g}}(V_j, \mathbf{w}, \mathbf{w}_j, \boldsymbol{n}_{ij})|_{\mathbf{w}=\mathbf{w}_i}, \quad \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_j} = \frac{\partial}{\partial \mathbf{w}}\hat{\mathbf{g}}(V_j, \mathbf{w}_i, \mathbf{w}, \boldsymbol{n}_{ij})|_{\mathbf{w}=\mathbf{w}_j}, \tag{5.20}$$

$$\frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_i} = \frac{\partial}{\partial \mathbf{w}}\hat{\mathbf{r}}(\mathbf{w}, \mathbf{w}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}_i, \mathbf{g}_j, \boldsymbol{n}_{ij})|_{\mathbf{w}=\mathbf{w}_i}, \quad \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j} = \frac{\partial}{\partial \mathbf{w}}\hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}_i, \mathbf{g}_j, \boldsymbol{n}_{ij})|_{\mathbf{w}=\mathbf{w}_j}, \tag{5.21}$$

$$\frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_i} = \frac{\partial}{\partial \mathbf{g}}\hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}, \mathbf{g}_j, \boldsymbol{n}_{ij})|_{\mathbf{g}=\mathbf{g}_i}, \quad \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j} = \frac{\partial}{\partial \mathbf{g}}\hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}_i, \mathbf{g}, \boldsymbol{n}_{ij})|_{\mathbf{g}=\mathbf{g}_j}. \tag{5.22}$$

The Jacobian-vector product assembly in Algorithm 6 constitutes a *level-1* hybrid symbolic/AD differentiation into the residual evaluation. At this level, AD only needs to be applied to the functions $\hat{\mathbf{g}}(\ldots)$ and $\hat{\mathbf{r}}(\ldots)$ that operate on objects associated with an edge and not the entire residual assembly function for (5.17).

We consider two *level-1* hybrid symbolic/AD strategies at the edge level for assembling the Jacobian-vector product in expression (5.18). The first is the *precomputed-storage* approach and the second is the *storage-free* approach. The *precomputed-storage* approach assembles Jacobian-vector products by computing edge-based Jacobian sub-matrices and storing them. The *storage-free level-1* approach simply applies AD at the edge level to local edge-based functions $\hat{\mathbf{g}}(\ldots)$ and $\hat{\mathbf{r}}(\ldots)$ using a single directional derivative. The *precomputed-storage level-1* approach initially applies AD at the edge-level to explicitly compute local Jacobians, store them, and then assemble the full Jacobian-vector product by matrix-vector multiplication through loops over the edges, utilizing the pre-stored local Jacobians.

When using the *precomputed-storage level-1* approach, the local edge-based Jacobians

$$\begin{aligned} &\frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, \quad \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, \\ &\frac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, \quad \frac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, \end{aligned} \tag{5.23}$$

are computed and stored in a loop over all edges. If nothing more was known at all about the function $\hat{\mathbf{g}}(\ldots)$ then these Jacobians alone would require the storage of $4((n_d+2)(n_d)(n_d+2)) = 4((3+2)(3)(5)) = 200$ `doubles` per edge in 3D. However, these matrices are actually diagonal, since the gradient is computed separately on each component of the conservative variables. The diagonal structure reduces the storage to $4((n_d+2)(n_d)) = 4((3+2)(3)) = 60$ `doubles` per edge. In addition, when using AD to generate these matrices, all of the variables in $\mathbf{w}_i$ and $\mathbf{w}_j$ can be perturbed simultaneously requiring just two columns in the seed matrix S.

After the Jacobians (5.23) have been computed, the edge-based Jacobians

$$\begin{array}{ll} \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2)\times(n_d+2)}, & \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2)\times(n_d+2)}, \\ \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_i} \in \mathbf{R}^{(n_d+2)\times(n_d+2)n_d}, & \dfrac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_j} \in \mathbf{R}^{(n_d+2)\times(n_d+2)n_d}, \end{array} \tag{5.24}$$

are computed and stored in a loop over all edges. Computing these edge-based Jacobians requires invoking forward AD (i.e. using `TFad<double>`) using a seed matrix with $(n_d+2)(2+2n_d) = (3+2)(2+2(3)) = 40$ columns in 3D (one column of identity for each component in $\mathbf{w}_i$, $\mathbf{w}_j$, $\mathbf{g}_i$ and $\mathbf{g}_j$). These edge-based Jacobians require the storage of $2(n_d+2)(n_d+2) + 2(n_d+2)(n_d+2)(n_d) = 200$ `doubles` per edge in 3D. Therefore, the total storage for edge-based Jacobians is $60 + 200 = 260$ `doubles` per edge in 3D. This is a significant amount of storage but, as shown in Section 5.4.4, the use of pre-computed Jacobians results in much more rapid evaluations of (5.18).

Symbolically differentiating deeper into the residual evaluation requires knowing more about the computations at the edge level. The next level of structure of these computations is more complicated, but great gains in speed and memory savings can be accomplished by symbolically differentiating deeper. First we describe the edge-based spatial gradient function $\hat{\mathbf{g}}(\ldots)$ and then the more complicated edge-based Euler residual function $\hat{\mathbf{r}}(\ldots)$.

The edge-based spatial gradient function $\hat{\mathbf{g}}(\ldots)$ takes the form

$$\hat{\mathbf{g}}(V_i, \mathbf{w}_i, \mathbf{w}_j, \boldsymbol{n}_{ij}) = \frac{1}{2|V_i|}(\mathbf{w}_i + \mathbf{w}_j) \otimes \boldsymbol{n}_{ij}. \tag{5.25}$$

The function $\hat{\mathbf{g}}(\ldots)$ is linear in $\mathbf{w}_i$ and $\mathbf{w}_j$ so simply passing in $\delta\mathbf{w}_i$ and $\delta\mathbf{w}_j$ into the `assemble_grad(...)` function returns the linearized spatial gradients $\delta\mathbf{g}_i$ and $\delta\mathbf{g}_j$. Clearly AD is not needed for this computation.

The edge-based residual function $\hat{\mathbf{r}}(\ldots)$ uses a Roe dissipation scheme together with linear reconstruction and a smooth limiter as follows:

$$\begin{aligned} \hat{\mathbf{r}}_{ij} &= \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \mathbf{g}_i, \mathbf{g}_j, \boldsymbol{n}_{ij}) \\ &= \frac{1}{2}\left(\mathbf{f}(\mathbf{w}_{ij}^{i+}) \cdot \boldsymbol{n}_{ij}\right) + \frac{1}{2}\left(\mathbf{f}(\mathbf{w}_{ij}^{j-}) \cdot \boldsymbol{n}_{ij}\right) + \mathbf{d}\left(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \boldsymbol{n}_{ij}\right) \end{aligned} \tag{5.26}$$

where:

$$\mathbf{w} = \begin{pmatrix} \rho \\ \rho \boldsymbol{u} \\ \rho E \end{pmatrix}, \tag{5.27}$$

$$\mathbf{f}(\mathbf{w}) = \begin{pmatrix} \rho \boldsymbol{u} \\ \rho\, \boldsymbol{u} \otimes \boldsymbol{u} + \boldsymbol{I} p \\ \boldsymbol{u}(\rho E + p) \end{pmatrix}, \quad \text{where} \tag{5.28}$$

$$p = (\gamma - 1)(\rho E - \frac{1}{2}|\boldsymbol{u}|^2),$$

$$\mathbf{w}_{ij}^{i+} = \mathbf{w}^{+}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{p}_{ij}^{i}), \tag{5.29}$$

$$\mathbf{w}_{ij}^{j-} = \mathbf{w}^{-}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{p}_{ij}^{j}), \tag{5.30}$$

$$\mathbf{p}_{ij}^{i} = \mathbf{g}_i \cdot (\boldsymbol{x}_j - \boldsymbol{x}_i) \in \mathbf{R}^{(n_d+2)}, \tag{5.31}$$

$$\mathbf{p}_{ij}^{j} = \mathbf{g}_j \cdot (\boldsymbol{x}_j - \boldsymbol{x}_i) \in \mathbf{R}^{(n_d+2)}. \tag{5.32}$$

$$\tag{5.33}$$

Above, $\mathbf{f}(\mathbf{w})$ in (5.28) is the flux function of the Euler equations $\nabla \cdot \mathbf{f}(\mathbf{w}) = 0$. Moreover, ρ, $\boldsymbol{u} \in \mathbf{R}^d$, p, and E are the density, velocity vector, pressure, and total energy per unit volume, respectively. The vectors $\mathbf{w}_{ij}^{i+}$ and $\mathbf{w}_{ij}^{j-}$ are the reconstructed left and right states. These reconstructions use the the spatial gradients $\mathbf{g}_i$ and $\mathbf{g}_j$ and are intended to increase the accuracy in regions where the solution is smooth at little additional computational cost but at the expense of effectively increasing the stencil, or footprint, of the operator. Embedded in the reconstruction functions $\mathbf{w}^{+}(\ldots)$ and $\mathbf{w}^{-}(\ldots)$ is a limiter that is needed to handle the very steep gradients around shocks. Several different limiters can be used. For the purposes of this discussion the details of the limiter are not important, except that we use limiters that are piecewise differentiable, such as the van Albada limiter. Finally, $\mathbf{d}(\ldots)$ is a dissipation term (of Roe type) that provides stability and dominates the computational cost of the method.

From the form of $\hat{\mathbf{r}}_{ij}$ in (5.26), its linearization with respect to $\mathbf{w}_i$, $\mathbf{w}_j$, $\mathbf{g}_i$ and $\mathbf{g}_j$ immediately follows as:

$$\delta\hat{\mathbf{r}}_{ij} = \mathbf{J}_{ij}^{i+}\delta\mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-}\delta\mathbf{w}_{ij}^{j-} \tag{5.34}$$

where

$$\mathbf{J}_{ij}^{i+} = \frac{1}{2}\frac{\partial\left(\mathbf{f}_{ij}^{i+} \cdot \boldsymbol{n}_{ij}\right)}{\partial \mathbf{w}_{ij}^{i+}} + \frac{\partial \mathbf{d}\left(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \boldsymbol{n}_{ij}\right)}{\partial \mathbf{w}_{ij}^{i+}}, \tag{5.35}$$

$$\mathbf{J}_{ij}^{j-} = \frac{1}{2}\frac{\partial\left(\mathbf{f}_{ij}^{j-} \cdot \boldsymbol{n}_{ij}\right)}{\partial \mathbf{w}_{ij}^{j-}} + \frac{\partial \mathbf{d}\left(\mathbf{w}_{ij}^{j-}, \mathbf{w}_{ij}^{j-}, \boldsymbol{n}_{ij}\right)}{\partial \mathbf{w}_{ij}^{j-}}, \tag{5.36}$$

$$\delta\mathbf{w}_{ij}^{i+} = \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i}\delta\mathbf{w}_i + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j}\delta\mathbf{w}_j + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^{i}}\delta\mathbf{p}_{ij}^{i}, \tag{5.37}$$

$$\delta\mathbf{w}_{ij}^{j-} = \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i}\delta\mathbf{w}_i + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j}\delta\mathbf{w}_j + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^{j}}\delta\mathbf{p}_{ij}^{j}, \tag{5.38}$$

$$\delta\mathbf{p}_{ij}^{i} = \delta\mathbf{g}_i \cdot (\boldsymbol{x}_i - \boldsymbol{x}_j), \tag{5.39}$$

$$\delta\mathbf{p}_{ij}^{j} = \delta\mathbf{g}_j \cdot (\boldsymbol{x}_j - \boldsymbol{x}_i), \tag{5.40}$$

and where we have used a notation for the derivatives analogous to expressions (5.20)–(5.22).

We now describe how this additional knowledge of the structure of $\hat{\mathbf{r}}(\ldots)$ is used to implement a *level-2* hybrid symbolic/AD Jacobian-vector product more efficiently. The *storage-free level-2* approach stores nothing up front and computes local Jacobian-vector products at the edge level symbolically for all terms except for the dissipation term $\mathbf{d}(\ldots)$. In all cases, we used AD with `TFad<double>` to perform all derivative computations with $\mathbf{d}(\ldots)$.

The *precomputed-storage level-2* approach initially computes and stores the sub-Jacobian matrices

$$\begin{array}{ll} \mathbf{J}_{ij}^{i+} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} & \mathbf{J}_{ij}^{j-} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} \\ \dfrac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} & \dfrac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} \\ \dfrac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} & \dfrac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} \\ \dfrac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^{i}} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} & \dfrac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^{j}} \in \mathbf{R}^{(n_d+2)\times(n_d+2)} \end{array} \tag{5.41}$$

and performs matrix-vector multiplication with these sub-matrices to assemble subsequent Jacobian-vector products of the form (5.18). The sub-Jacobians $\mathbf{w}_{ij}^{i+}$ and $\mathbf{w}_{ij}^{j-}$ are diagonal since the reconstructions and the limiters are component-wise operations. All of these Jacobian evaluations were manually derived and coded in C++ except for the dissipation term $\mathbf{d}(\ldots)$. This scheme requires the storage of $2(n_d+2)(n_d+2)+4(n_d+2)+2(n_d+2) = 2(3+2)(3+2)+4(3+2)+2(3+2) = 80$ `doubles` per edge in 3D. Once these edge-based sub-Jacobians are computed, they are utilized in the edge-based loops shown in Algorithm 7 to assemble the Jacobian-vector product (5.18).

Algorithm 7. Level-2 hybrid symbolic/AD Jacobian-vector product assembly

---

Compute $\delta\mathbf{r} = \dfrac{\partial \mathbf{r}}{\partial \mathbf{w}}\delta\mathbf{w}$ given input vector $\delta\mathbf{w}$

 # Linearized spatial gradient assembly

1. Set $\delta\mathbf{g}_i \leftarrow \mathbf{0} \qquad \forall i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{ij}$:
   (a) $\delta\mathbf{g}_i \leftarrow \delta\mathbf{g}_i + \hat{\mathbf{g}}(V_i, \delta\mathbf{w}_i, \delta\mathbf{w}_j, +\boldsymbol{n}_{ij})$
   (b) $\delta\mathbf{g}_j \leftarrow \delta\mathbf{g}_j + \hat{\mathbf{g}}(V_j, \delta\mathbf{w}_j, \delta\mathbf{w}_i, -\boldsymbol{n}_{ij})$
3. + Boundary contributions

 # Linearized residual assembly

1. Set $\delta\mathbf{r}_i = \mathbf{0} \qquad \forall i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{ij}$:

Method	Precomputed-storage	Storage-free
Level-0	-	0
Level-1	260	0
Level-2	80	0

Table 5.1. Storage of edge-based Jacobian contributions in number of `doubles` per edge

(a) $\delta\mathbf{r}_{ij} \leftarrow \mathbf{J}_{ij}^{i+}\delta\mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-}\delta\mathbf{w}_{ij}^{j-}$, where and $\delta\mathbf{w}_{ij}^{i+}$ and $\delta\mathbf{w}_{ij}^{j-}$ are computed using (5.37)-(5.40).

(b) $\delta\mathbf{r}_i \leftarrow \delta\mathbf{r}_i + \delta\mathbf{r}_{ij}$,

(c) $\delta\mathbf{r}_j \leftarrow \delta\mathbf{r}_j - \delta\mathbf{r}_{ij}$,

3. + Boundary contributions

5.4.4 Results for various Jacobian-vector product assembly methods

Tables 5.1 and 5.2 give computational results for *precomputed-storage* and *storage-free level-0*, *level-1* and *level-2* hybrid symbolic/AD schemes used to compute Jacobian-vector products of the form (5.18) described above. Strategies to assemble Jacobian-vector products of the form (5.18) are summarized below:

Level-0: Apply direction forward AD (i.e. using the seed matrix $S = \delta\mathbf{w}$) at the global residual function call level (i.e. AD the entire C++ function for (5.17)).

Level-1: AD is applied in a black-box way at the edge assembly level.

Precomputed-storage Level-1: AD is applied at the edge level in a black-box way to generate the sub-matrices in (5.23) and (5.24) and then these sub-matrices are used to assemble Jacobian-vector products using Algorithm 6.

Storage-free Level-1: Directional AD is applied in a black-box way at the edge assembly level on the functions $\hat{g}(\ldots)$ and $\hat{r}(\ldots)$.

Level-2: Hybrid symbolic/AD is applied at the edge level where symbolically derived derivatives are used for everything except the dissipation term $\mathbf{d}(\ldots)$.

Precomputed-storage Level-2: The sub-matrices in (5.41) are computed symbolically up front with AD used only for the dissipation term $\mathbf{d}(\ldots)$. Once these edge-level sub-matrices are computed and stored then Algorithm 7 is used to assemble Jacobian-vector products.

Storage-free Level-2: Edge-level Jacobian-vector products are computed using symbolically derived and hand-coded derivatives for all expressions except for the dissipation term $\mathbf{d}(\ldots)$ where directional AD is used.

Method	Precomputation	Precomputed-storage Jac-vec	Storage-free Jac-vec
Level-0	-	-	3.14
Level-1	24.8	0.21	3.14
Level-2	4.73	0.19	0.1.93

Table 5.2. CPU time relative to residual evaluation for small mesh example.

For the timing results reported here, we used a small and non-physical test mesh where all of the computations easily fit in cache. All computations were run several times in a loop for a total of about 1.0 CPU seconds in order to obtain reliable timings. These tests therefore compare the floating point efficiency of the implemented code and not the quality of the data structures with respect the memory usage (e.g. cache misses). For all AD computations, the class `TFad<double>` was used as the scalar type for the active variables. All derivative computation CPU times reported are relative to the time for the residual evaluation which used the scalar type `double`. All of the results reported in this section were preformed by compiling the C++ code using GNU g++ version 3.1 and run under Red Hat Linux 7.2 on a 1.7 GHz Intel Pentium IV processor machine.

First we compare and contrast the different levels of *storage-free* approaches that assemble Jacobian-vector products on the fly requiring no upfront storage or computation. These approaches may be preferred when memory storage is at a premium. The *storage-free Level-0* method required about 3.14 times the CPU time compared to the CPU time to evaluate the residual function $\mathbf{r}(\ldots)$ using `double`. The *storage-free Level-0* is only about 50% more expensive than a central FD approximation. In addition to being computationally efficient in comparison to FD approximations, the Jacobian-vector products are accurate to machine precision.

The *storage-free Level-1* method gives exactly the same relative CPU time of 3.14 as the *storage-free Level-0* approach. This should not be surprising since this approach performs almost exactly the same computations as applying AD to the entire residual assembly function. In fact, in our C++ implementation, the Jacobian-vector products for the *storage-free Level-0* and *storage-free Level-1* approaches gave the same resultant vector down to the last bit. The primary advantage of the *storage-free Level-1* approach over the *storage-free Level-0* approach is that the entire residual evaluation function does not have to be templated on the scalar type.

The *storage-free Level-2* approach exploits the gradient and residual computations and thereby reduces the relative CPU time from 3.14 to just 1.93. This is now faster than a central FD but again gives exact Jacobian-vector products. However, this hybrid SD/AD approach is still almost twice as expensive as one-sided finite differences. To obtain better performance we need to consider methods that perform some initial computations up front. Next, we therefore compare and contrast the various levels of *precomputed-storage* approaches that compute and store Jacobian sub-matrices at the edge level and then use them to assemble repeated Jacobian-vector products. These methods require more storage but can significantly speed up repeated Jacobian-vector product assemblies.

The *precomputed-storage Level-1* approach requires 24.8 times the CPU time of the residual assembly in order to compute the edge-level matrices and store 260 `double`s per edge. Once these matrices are computed and stored, the relative CPU time for each subsequent Jacobian-vector product assembly is only

0.21 times the cost of a residual assembly. This is almost five times faster than a one-side FD approximation. Note that the relative CPU time of 24.8 is actually quite impressive considering that `TFad<double>` is used on the function $\hat{\mathbf{r}}(\ldots)$ with 40 derivative components. Therefore, the cost for each derivative component is less than the cost of the function evaluation. The relative CPU time of 24.8 for the creation of these matrices, however, may still seem too high but in many situations where Jacobian-vector products are performed at a particular point $\mathbf{w}$ (such as in an iterative linear solver like GMRES in a Krylov-Newton method) the overall reduction in runtime over even one-side FDs can be significant. If the memory capacity is limited or the reduction in CPU is insufficient, then additional symbolic differentiation can be considered where Jacobian sub-matrices at the edge level are stored and all derivative computations, except for the dissipation term, are derived symbolically and hand coded.

We show results for the *precomputed-storage Level-2* approach. The Jacobian sub-matrices for the dissipation term $\mathbf{d}(\ldots)$ in (5.35) and (5.36) are computed using `TFad<double>` (using 10 columns in the seed matrix). This approach results in dramatic reductions in relative CPU time for pre-computing edge-based Jacobian sub-matrices and storage of these sub-matrices over the *precomputed-storage Level-1* approach. The relative CPU time is reduced from 24.8 to 4.73 and the storage for the Jacobian sub-matrices is reduced from 260 to 80 `double`s per edge. However, we only see a minor reduction in relative CPU time from 0.21 to 0.19 for subsequent Jacobian-vector product assemblies. Even through the reduction in Jacobian-vector product computations is not significantly reduced, the dramatic reductions in upfront cost and storage makes this scheme much more efficient in many cases.

We experimented with several different approaches to automatically differentiate the Roe-type dissipation term $\mathbf{d}(\ldots)$ shown in (5.26) and also attempted to derive the Jacobians symbolically. It required approximately a month to derive expressions for the Jacobians of $\mathbf{d}(\ldots)$ in (5.35) and (5.36) and after approximately two weeks of implemention the resulting hand-derived and hand-coded Jacobian evaluation C++ code was 2.5 times slower than the automatically generated AD code (using `TFad<double>`) and was not giving the correct Jacobians. The likely reason for the degradation in performance of the hand-coded Jacobian evaluation function is that there are many common expressions involved in the Jacobian entries that are automatically exploited in the AD code that were not exploited in the hand-coded derivative function. The attempt to symbolically derive and implement these Jacobians was therefore abandoned to avoid expending inpractical amounts of effort.

Finally, results are presented for the use of several different automated strategies to compute the Jacobian $\partial\mathbf{d}/\partial\mathbf{w}_i \in \mathbf{R}^{(n_d+2)\times(n_d+2)}$. When $n_d = 3$ (3D) then the Jacobian is of size 5×5. Table 5.3 shows relative CPU times on a variety of platforms for computing the 5×5 Jacobian $\partial\mathbf{d}/\partial\mathbf{w}_i$ using one-sided finite difference (FD), automatic differentiation (AD) using `TFad<double>`, and complex-step (CS) using `std::complex<double>`.

An interesting thing to note is that the relative CPU time of performing one-side finite differences varies by as much as 6.2 for icl to 7.4 for sgi even through exactly the same C++ source code was compiled on each of these platforms and similar optimization levels where used. A relative CPU time of 6.0 for the one-sided finite difference would be ideal since six evaluations of $\mathbf{d}(\ldots)$ are performed (one for the base point and five others for each perturbation in $\mathbf{w}_i$). Addition overhead is also incurred for performing the vector subtraction and scaling such as shown in (5.4).

These results also show that the CS method using `std::complex<double>` is much less efficient than the AD method using `TFad<double>`. However, all of our tests showed that the Jacobian entries for AD and CS agreed to all by the last two significant digits. As mentioned earlier, there are several factors contributing to the inefficiency of the CS method. First, the CS method performs some extra unnecessary

Compiler/platform	One-sided finite differences	Automatic differentiation	Complex step
g++	7.0	7.5	39.9
kcc	7.0	12.5	31.6
icl	6.2	10.3	59.3
sgi	7.4	19.5	39.0
cxx	6.7	57.4	197.5

Platforms
g++ : GNU g++ version 3.1, Red Hat Linux 7.2, 1.7 GHz Intel P IV
kcc : KAI C++ version 4.0e, Red Hat Linux 7.2, 1.7 GHz Intel P IV
icl : Intel C++ version 5.0, MS Windows 2000, 1.0 GHz Intel P IV
sgi : MipsPro C++ version 7.3.1
cxx : Compaq C++ version, CPlant[4]

Table 5.3.
Relative CPU times for the computation of the 5×5 Jacobian of the dissipation term $\partial \mathbf{d}/\partial \mathbf{w}_i$ using one-sided finite difference (FD), automatic differentiation (AD) using `TFad<double>` and the complex-step (CS) using `std::complex<double>`. All CPU times are relative to evaluation of the dissipation term $\mathbf{d}(\ldots)$.

float-point computations that AD does not. Second, the CS method recomputes the value of the dissipation function $\mathbf{d}$ over and over again for each separate column of $\partial \mathbf{d}/\partial \mathbf{w}_i$ while the AD method only computes the value of $\mathbf{d}$ once independent of the number of columns in the Jacobian being computed. A third factor is that none of the vendors provided implementations of `std::complex<>` uses the expression template (ET) technique to reduce the number of temporaries which are typically generated by operator overloading but not when ET is used.

The last interesting issue to discuss is how radically different the performance varies for the handling of the scalar types `TFad<double>` and `std::complex<double>` as compared to `double` among these different compilers. The relative performance of `TFad<double>` varies all the way from 7.5 for g++ to 57.4 for cxx. The relative performance of 7.5 for g++ is amazingly good considering that in terms of flops, the lower bound is $1 + 5 = 6$ in the best case (i.e. all '$+$' and $-$ operations) and as high as $1 + 5(5) = 26$ in the worst case (all '$/$' operations). We did not get an exact flop count for the AD approach but we suspect that the relative number of flops is significantly greater than 7.5 (since there are a lot of multiplications and these multiplications double the number of multiplications in the derivative evaluations) so the AD code actually exceeds what would be predicted from the idea flop count. The case of 57.4 for cxx is almost double the worst-case flop count. Clearly a good C++ compiler is needed to make the use of C++ programming constructs, such as the handling of derived data types and expression templates, efficient.

5.5 Conclusions

We have considered various approaches to compute Jacobian-vector products for a ANSI C++ implementation of a finite-volume discretion of the Euler equations. Non-invasive and invasive hybrid

symbolic/AD strategies that pre-compute and store Jacobian sub-matrices at the mesh-object level and then assemble Jacobian-vector products at the mesh-object level may result in computations nearly five times faster than one-sided finite differences. In addition, these derivative computations are accurate to machine precision. We also found that low storage on-the-fly black-box uses of AD as the edge level can result in derivative computations that are very competitive in speed with FD methods and require little knowledge of the actual edge-level computations. However, exploiting the structure of the computations at a lower level and combining symbolically derived and coded derivatives with AD resulted in significant improvements in CPU time and storage. Conclusions and observations are summarized as follows:

1. Hybrid symbolic/AD approaches are well suited to discretization methods for PDEs due to the hierarchal structure of such methods. Hybrid symbolic/AD methods can result in very accurate and affordable derivative computations without having to rewrite an entire code but instead focus differentiation tools on lower-level computations based on mesh objects.

2. These types of applications of AD generally do not require sophisticated features such as the special sparsity and check-pointing handling for storage management that are the focus of many of the current groups performing research in AD [38, 100]. This type of use of AD, therefore, allows for more effort to be focused on low level efficiency where many of the current AD tools for ANSI C++ can improve.

3. One should start an effort to provide derivative computations by applying AD at the least intrusive level possible where it is computationally realistic. This results in an implementation of derivative computations that are accurate to machine precision very quickly which then allows efforts to be directed to other types of functionality as needed. Only when more efficient derivative computations are needed should an attempt be made to symbolically differentiation deeper into a code. Too much time can be spent trying to prematurely optimize certain computations without good economic or other justification and the desire to prematurely symbolically differentiate and hand code derivatives for every expression is one such area. If nothing else, having a correct derivative computation using a non-invasive AD approach provides a means to test later efforts to implement more efficient derivative computations.

4. Any new or existing C++ project that is considering derivative computations should template on the scalar type as much computational C++ code as possible. Templating code by the scalar type not only allows the use of AD but other techniques such as interval analysis, extended precision arithmetic and complex arithmetic.

5. Symbolic differentiation is not always efficient or affordable even though theoretically such an approach should result in the best quality differentiation code. Too much implementation effort is necessary to provide symbolic derivatives that are more efficient than those generated automatically by AD.

6. The complex-step method for differentiation is analogous to automatic differentiation but less efficient (and technically less accurate). Therefore, since many good AD classes for C++ are available, the complex-step method for the computation of derivatives should never be seriously considered for production use in ANSI C++ codes. However, as an extra validation approach, the use of the complex-step method in ANSI-C++ codes may be very reasonable. In other languages that lack support for operator overloading but yet support complex arithmetic, such as Fortran 77, the complex-step method is a much more attractive alternative. CS performs more flops than AD, even for a single derivate propagation. In addition, complex types are not well optimized in C++ compilers.

Chapter 6

Comparison of Operators for Newton-Krylov Method for Solving Compressible Flows on Unstructured Meshes

6.1 Introduction

Newton Krylov methods (NK) represent an 'inexact' variant of Newton's method in which the linear solve (inner iteration) is performed iteratively to within a prescribed tolerance. This approach is appealing for solving large sparse nonlinear systems due to the excellent convergence behavior of Newton's method combined with the scalability and less severe memory constraints of iterative linear solvers compared to direct linear solvers. A potential drawback of NK, however, lies in the inexact nature of the linear solve reducing the quadratic rate of convergence associated with exact application of Newton's method. Other sources of inexactness which could prevent quadratic convergence and which are explored in this study consist of approximate and infrequent construction of the Jacobians and/or their corresponding preconditioners. Oftentimes, the penalty of reduced nonlinear convergence rate can be more than compensated by much cheaper linear solves. This is dependent on details of the construction and solution of the linear system, can be problem-dependent, and is the focus of our study presented here.

We examine several ways of implementing NK for solving steady-state compressible flows on unstructured three-dimensional (3D) meshes. The implementations differ in the way the Jacobian and preconditioner operators of the linear systems are constructed and applied and are described more fully in the Nonlinear Solvers section. The operators arise from discretizing the governing equations using a node-centered finite-volume method. Control-volumes and control-volume surfaces are defined by the median dual of the primal finite element mesh. Node and edge-based data structures are defined in which volumes are associated with nodes and control-surfaces are associated with edges and boundary nodes [19] [20] [152] [197]. Residual equations are constructed by looping over nodes and edges. First-order accurate upwind spatial discretizations are based on piece-wise constant values within a cell and require only a single pass over nodes and edges to construct the residual equations. Second-order spatial accuracy typically involve MUSCL extrapolation which requires one pass over the mesh to construct and store nodal gradients and

possibly gradient limiters, and a second pass for constructing the residual. This approach allows us to perform the residual computations by looping over edges and nodes only.

From a global perspective, the structure of the Jacobian for second-order accurate upwind schemes requires distance-two connectivity information for the nodes within the mesh. This represents a dramatic increase in memory for storage and in flops for filling these entries compared to a first-order accurate discretization. Moreover, several choices exist for computing nodal gradients and their limiters, and some of these can be quite complex and even nondifferentiable. For these reasons, we have chosen to base all Jacobian and preconditioner constructions on distance-one (first-order) connectivity information only. Second order information is taken into account using a matrix-free (MF) implementation, where instead of forming and then using the Jacobian matrix, the matrix-vector product required in the Krylov linear solver is approximated by a finite-difference. Since the finite-difference is formed by differencing two residual vectors, only two residual vectors need occupy memory at any given time. This leads to a dramatic reduction in memory requirements.

While a matrix-free implementation of NK obviates the need to construct a Jacobian matrix, convergence of the linear solve (inner iteration) is typically extremely slow without the use of preconditioning. Hence, there is still a need to form, if even approximately, a matrix to be used as a preconditioner. Our experience as well as that of others has shown this issue to be critical in making NK a viable means of reducing overall CPU cost relative to other methods.

We describe the implementation and performance of the various operators applied to finding steady-state solutions on unstructured meshes. The next section briefly summarizes the governing equations followed by a section describing the discretization of these equations over an unstructured mesh via the finite volume method and the approximate analytic preconditioner operators that result from the discretized equations. Next, we give details of the nonlinear solver algorithm and operator choices provided by the NOX nonlinear solver library. This is followed by a presentation of the results. Finally, we summarize our results and discuss current and future research.

6.2 Governing Equations

In integral form the governing equations can be written as

$$\int_{\mathcal{V}} \frac{\partial \mathbf{W}}{\partial t}\, d\mathcal{V} + \oint_{\Gamma} (\mathbf{F}^c - \mathbf{F}^v) \cdot \mathbf{n}\, d\Gamma = 0 \tag{6.1}$$

where $\mathbf{W} = \{\rho,\, \rho u,\, \rho v,\, \rho w,\, \rho E\}^T$ is the solution vector of conserved variables;

$$\mathbf{F}^c \cdot \mathbf{n} = \left\{ \begin{array}{c} (\rho u_i) n_i \\ (\rho u_1 u_i + p\delta_{i1}) n_i \\ (\rho u_2 u_i + p\delta_{i2}) n_i \\ (\rho u_3 u_i + p\delta_{i3}) n_i \\ ((\rho E + p)\, u_i) n_i \end{array} \right\}$$

is the inviscid flux vector; and

$$\mathbf{F}^v \cdot \mathbf{n} = \left\{ \begin{array}{c} 0 \\ \tau_{i1} n_i \\ \tau_{i2} n_i \\ \tau_{i3} n_i \\ (-q_i + u_j \tau_{ij}) n_i \end{array} \right\}$$

is the viscous flux vector. In the above equations, ρ is the density; $u_i = \mathbf{u}$, where $u = u_1$, $v = u_2$, and $w = u_3$ are velocity components in each Cartesian direction, x_j; p is the pressure; E is the total energy, defined by $E = e + (u^2 + v^2 + w^2)/2$; e is the specific internal energy, defined by $e = C_v T$; C_v is the specific heat at constant volume; T is the temperature, defined for an ideal gas by $p = \rho R T$; R is the gas constant, $n_i = \mathbf{n}$, is the surface normal vector, and δ_{ij} is the Kronecker delta. We also define a the primitive state vector as $\mathbf{U} = \{\rho, u, v, w, p\}^T$.

The viscous stress tensor is given by,

$$\tau_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij} \tag{6.2}$$

where μ is the molecular viscosity coefficient. The heat flux vector is assumed to follow Fourier's Law,

$$q_i = -\kappa \frac{\partial T}{\partial x_i} \tag{6.3}$$

and a constant Prandtl number, Pr is used.

On a solid impermeable surface in the case of inviscid flow, the velocity vector must be tangent to the surface, $\mathbf{u} \cdot \mathbf{n} = 0$ and there is no mass or heat flux crossing the boundary and so the the flux function becomes,

$$\mathbf{F}^b = \mathbf{F} \cdot \mathbf{n}^b = \left\{ \begin{array}{c} 0 \\ p\mathbf{n}^b \\ 0 \end{array} \right\} .$$

On solid no-slip boundaries, $\mathbf{u} = 0$ is inforced in a strong manner, in addition if the surface temperature is desired a bounary temperature is specified. Farfield boundary conditions are defined by an assumed infinity state $\mathbf{U}_\infty$ and applied as the right state to the Reimann solver resulting in a boundary flux.

6.3 Algorithmic Formulation

The compressible flow code, Premo, uses the SIERRA framework [77], which provides common services to application codes such as user-input parsing, bulk mesh-data input/output, interfaces to linear and nonlinear solvers, inter-mesh transfers, parallel communications, dynamic load balancing, and adaptivity.

Premo is a node-centered, edge-based, finite-volume, algorithm similar the formulations of Haselbacher and Blazek [107] and Luo *et al*. [152].

Control volumes and surfaces are defined by the median dual of the input finite-element mesh. This construction results in non-overlapping, space-filling, control volumes that are associated with nodes. Control-surface area vectors are area-weighted averages and are stored on the edge.

If we define volume-averaged values ($\mathbf{W}$), we can write the semi discrete form of Eq. (6.1) as,

$$\frac{\partial \mathbf{W}_I}{\partial t} = \frac{-1}{\delta \mathcal{V}_I} \sum_{J \in NE(I)} (\mathbf{F}^c_{IJ} - \mathbf{F}^v_{IJ}) |\delta \Gamma_{IJ}| = 0 \tag{6.4}$$

where I is the current node (*i.e.*, current control volume), $\delta \mathcal{V}_I$ is the control volume, $NE(I)$ is the set of nodes connected to node I by edge IJ, J is the second node on the edge, $|\delta \Gamma_{IJ}|$ is the magnitude of the control-surface area. The convective and viscous flux functions are evaluated at the edge midpoint and depend on the reconstructed data, normal vector and nodal gradients $\mathbf{F}^c_{IJ} = \mathbf{F}(\mathbf{U}^+_I, \mathbf{U}^-_J, \mathbf{n}_{IJ})$ and $\mathbf{F}^v_{IJ} = \mathbf{F}(\mathbf{U}_I, \nabla \mathbf{U}_I, \mathbf{U}_J, \nabla \mathbf{U}_J, \mathbf{n}_{IJ})$.

On slip boundaries,

$$\frac{\partial \mathbf{W}_I}{\partial t} = \frac{-1}{\delta \mathcal{V}_I} \sum_{J \in NE(I)} (\mathbf{F}^c_{IJ} - \mathbf{F}^v_{IJ}) |\delta \Gamma_{IJ}| + \mathbf{F}^b |\delta \Gamma^b_I| = 0 \, . \tag{6.5}$$

The advection scheme that Premo currently uses is the approximate Riemann solver of Roe [183] with the entropy fix of Liou and van Leer [148] and the implementation closely follows that of Whitaker [216]. This schemes require a left (+) and right(-) state in order to evaluate the interface fluxes.

First order accuracy is achieved by assuming a piecewise constant state within control volumes and interface values are just the node values themselves,

$$\mathbf{U}^+_I = \mathbf{U}_I \tag{6.6}$$
$$\mathbf{U}^-_J = \mathbf{U}_J \, . \tag{6.7}$$

For higher-order schemes, a piecewise linear variation within the cells is assumed. Reconstruction of the interface values use a multi-dimensional MUSCL extrapolation [207] [19] along with the average nodal gradient. The Green-Gauss volume-averaged gradient is obtained by solving a surface integral

$$\int_{\mathcal{V}} \nabla \phi \, d\mathcal{V} = \delta \mathcal{V} \nabla \phi = \oint_{\Gamma} \phi \mathbf{n} \, d\Gamma \, .$$

The discrete form is given by,

$$\nabla \phi_I \approx \frac{1}{\delta \mathcal{V}_I} \sum_{J \in NE(I)} \left(\frac{\phi_I + \phi_J}{2} \right) \mathbf{n}_{IJ} \, |\delta \Gamma_{IJ}| \tag{6.8}$$

which is an edge-based formula. For boundary nodes,

$$\nabla \phi_I \approx \frac{1}{\delta \mathcal{V}_I} \sum_{J \in NE(I)} \left(\frac{\phi_I + \phi_J}{2} \right) \mathbf{n}_{IJ} \, |\delta \Gamma_{IJ}| + \phi_I \mathbf{n}^b_I \left| \delta \Gamma^b_I \right| \, . \tag{6.9}$$

Linear reconstruction using MUSCL on either side of an interface is obtained by the following,

$$\mathbf{U}^+_I = \mathbf{U}_I + \Phi(\Delta^+_I, \Delta^-_I) \left(\nabla \mathbf{U}_I \cdot \frac{\Delta \mathbf{r}}{2} \right) \tag{6.10}$$
$$\mathbf{U}^-_J = \mathbf{U}_J - \Phi(\Delta^+_J, \Delta^-_J) \left(\nabla \mathbf{U}_J \cdot \frac{\Delta \mathbf{r}}{2} \right)$$

where $\Delta\mathbf{r} = \mathbf{x}_J - \mathbf{x}_I$ is a displacement vector, and $\Phi = [0, 1]$ is a gradient limiter. Gradients are limited by the van Albada limiter, similar to that used by Luo *et al.* [152] which is computed locally on each edge. The van Albada limiter is written as,

$$\Phi(a, b) = \frac{ab + |ab| + \epsilon}{a^2 + b^2 + \epsilon} \tag{6.11}$$

and the arguments are defined by,

$$\begin{aligned} \Delta_I^+ &= \mathbf{U}_J - \mathbf{U}_I \\ \Delta_I^- &= 2\nabla\mathbf{U}_I \cdot \Delta\mathbf{r} - (\mathbf{U}_J - \mathbf{U}_I) \\ \Delta_J^+ &= 2\nabla\mathbf{U}_J \cdot \Delta\mathbf{r} - (\mathbf{U}_J - \mathbf{U}_I) \\ \Delta_J^- &= \mathbf{U}_J - \mathbf{U}_I \, . \end{aligned} \tag{6.12}$$

The Roe flux function can be written as,

$$\mathbf{F}_{IJ}^c = \frac{1}{2}\left(\mathbf{F}(W_I^+, \mathbf{n}_{IJ}) + \mathbf{F}(W_J^-, \mathbf{n}_{IJ}) - |\tilde{A}|(W_J^- - W_I^+)\right) \tag{6.13}$$

where $A = \frac{\partial F}{\partial W}$ is the flux Jacobian matrix and $|\tilde{A}| = |A(\tilde{W}_I^+, \tilde{W}_J^-, \mathbf{n}_{IJ})|$ is evaluated using an average state $(\tilde{\cdot})$ defined by Roe [183]. We have implemented three approximations to the Jacobian based on linearizing eq. 6.13. The first is referred to as **AD** which is the exact differentiation of eq. 6.13 based on edge data and is evaluated using automatic differentiation [32] [98]

$$\begin{aligned} \frac{\partial F_I^{\mathbf{AD}}}{\partial \mathbf{W}_I} &= \mathbf{I}\frac{\delta\mathcal{V}}{\Delta t} + \frac{1}{2}\left(A(W_I, \mathbf{n}_{IJ}) + |\tilde{A}| - \frac{\partial|\tilde{A}|}{\partial \mathbf{W}_I}(W_J - W_I)\right) \\ \frac{\partial F_I^{\mathbf{AD}}}{\partial \mathbf{W}_J} &= \frac{1}{2}\left(A(W_J, \mathbf{n}_{IJ}) - |\tilde{A}| - \frac{\partial|\tilde{A}|}{\partial \mathbf{W}_J}(W_J - W_I)\right) \, . \end{aligned} \tag{6.14}$$

The second approximation denoted (**ROE**) was suggested by Luo *et al.* [154], neglects the variation of the dissipation matrix with respect to $\mathbf{W}$,

$$\begin{aligned} \frac{\partial F_I^{\mathbf{ROE}}}{\partial \mathbf{W}_I} &= \mathbf{I}\frac{\delta\mathcal{V}}{\Delta t} + \frac{1}{2}\left(A(W_I, \mathbf{n}_{IJ}) + |\tilde{A}|\right) \\ \frac{\partial F_I^{\mathbf{ROE}}}{\partial \mathbf{W}_J} &= \frac{1}{2}\left(A(W_J, \mathbf{n}_{IJ}) - |\tilde{A}|\right) \end{aligned} \tag{6.15}$$

and the third approximation (**SD**) is the Jacobian with respect to a simplified flux function that assumes symmetric artificial dissipation, also suggested by Luo *et al.* [153],

$$\mathbf{F}_{IJ} = \frac{1}{2}\left(F(W_I, \mathbf{n}_{IJ}) + F(W_J, \mathbf{n}_{IJ}) - |\lambda_{IJ}|(W_J - W_I)\right) \tag{6.16}$$

and the approximate Jacobian terms are,

$$\begin{aligned} \frac{\partial F_I^{\mathbf{SD}}}{\partial \mathbf{W}_I} &= \mathbf{I}\frac{\delta\mathcal{V}}{\Delta t} + \frac{1}{2}\left(A(W_I, \mathbf{n}_{IJ}) + |\lambda_{IJ}|\mathbf{I}\right) \\ \frac{\partial F_I^{\mathbf{SD}}}{\partial \mathbf{W}_J} &= \frac{1}{2}\left(A(W_J, \mathbf{n}_{IJ}) - |\lambda_{IJ}|\mathbf{I}\right) \end{aligned} \tag{6.17}$$

where $|\lambda_{IJ}|$ is the spectral radius of the flux Jacobian matrix and $\mathbf{I}$ is the identity matrix.

Evaluating the viscous flux vector requires constructing primitive variable gradients at edge midpoints. Gradients are constructed by blending edge node differences with nodal gradients,

$$\begin{aligned} \nabla U_{IJ} &= \frac{(\nabla U_I + \nabla U_J)}{2} \\ &+ \left(U_J - U_I - \frac{(\nabla U_I + \nabla U_J)}{2} \cdot \Delta \mathbf{r}_{IJ} \right) \frac{\Delta \mathbf{r}_{IJ}}{|\Delta \mathbf{r}_{IJ}|^2} \, . \end{aligned} \tag{6.18}$$

Viscosity, conductivity and velocity components appearing in the energy dissipation terms are evaluated as the average of the two node values. The support for these gradients include distance-two nodes. Therefore, exact discrete Jacobians cannot be contained within the edge graph. Instead, only the node differences are considered in the analytic representation of the Jacobian terms,

$$\nabla U_{IJ} \approx (U_J - U_I) \frac{\Delta \mathbf{r}_{IJ}}{|\Delta \mathbf{r}_{IJ}|^2} \, . \tag{6.19}$$

This results in a very efficient algorithm for evaluating the terms but is exact only on Cartesian grids.

In all cases, whether it be the evaluation the spatial residual equations, eq. 6.4 or the nodal gradients or an approximation to the Jacobian operator, assembly only requires looping over edges and nodes.

6.4 Steady-State Solution Strategies

6.4.1 Newton-based Strategies

For this work, steady state solutions are sought. This amounts to seeking a solution for eq. (6.1) for which the time derivative term is zero. With the exception of explicit time integration, all methods in this work are implicit-based and employ Newton-Krylov solutions (discussed later) of systems of nonlinear equations to drive the discretized compressible fluid flow equations to steady-state. Spatial discretization of eq. (6.1) leads to a system of nonlinear algebraic equations given generally by

$$\mathbf{F}(\mathbf{W}) = \frac{\partial \tilde{\mathbf{W}}}{\partial t} + \mathbf{R}(\mathbf{W}) = \mathbf{0} \tag{6.20}$$

where $\mathbf{W}$ is the vector of unknowns, and $\mathbf{R}(\mathbf{W})$ is the spatial residual equations which have no explicit time derivative dependencies. A straightforward approach to solving for steady-state would seek a solution to $\mathbf{R}(\mathbf{W}) = \mathbf{0}$. However, such an approach proves intractable in practice due to the poor quality of any linearization of the problem and the difficulty iterative linear solvers have in seeking their solutions. Consequently, ways of making the problem easier are sought by selecting forms for $\frac{\partial \tilde{\mathbf{W}}}{\partial t}$. Within the context of a Newton-Krylov method, these forms typically produce what amounts to a diagonal mass matrix which can dramatically facilitate the linear solves.

In this work, we seek solutions of the steady-state problem $\mathbf{R}(\mathbf{W}) = \mathbf{0}$ by solving a sequence of problems of the form of eq. (6.20) in which the stabilizing term represented by $\frac{\partial \tilde{\mathbf{W}}}{\partial t}$ is gradually removed. Each

Table 6.1. Solution Strategies for Steady-State.

Method	Residual, $\tilde{\mathbf{F}}$	Jacobian, $\tilde{\mathbf{J}}$
Transient	$\frac{\mathbf{W}-\mathbf{W}_{t-\Delta t}}{\Delta t}+\mathbf{R}$	$\frac{1}{\Delta t}\mathbf{I}+\frac{\partial \mathbf{R}}{\partial \mathbf{W}}$
Pseudo Transient	$\mathbf{R}$	$k\mathbf{I}+\frac{\partial \mathbf{R}}{\partial \mathbf{W}}$
Homotopy	$(1-\lambda)(\mathbf{W}-\mathbf{W}_a)$ $+\lambda\mathbf{R}$	$(1-\lambda)\mathbf{I}$ $+\lambda\frac{\partial \mathbf{R}}{\partial \mathbf{W}}$

problem in the sequence is solved by one of several variations of a Newton-Krylov method which differ in the choice of Jacobian and preconditioner operators. Efficient testing of the many Jacobian/preconditioner operator choices (along with associated options) is enabled by interfacing the compressible flow code, Premo, to a library of nonlinear solver algorithms, NOX [136], currently being developed at Sandia National Laboratories.

A sequence of nonlinear problems is generated by choosing a strategy to manipulate the time derivative term of eq. (6.20). This work considers three possibilities summarized in Table 6.1.

6.5 Transient

This strategy simply attempts to find a steady-state solution by integrating in time until all transients have vanished. Using an implicit time integration scheme allows for unconditional stability with respect to the time-step size, Δt. Table 6.1 represents implicit time integration using backward Euler, the simplest and most stable choice. Though adversely affecting the accuracy of the time-dependent solutions, choosing large time steps can lead to faster steady-state solutions being obtained. It should be noted that explicit time integration can be applied to the same equations to march the solution to steady-state without requiring the solution of a nonlienar system of equations at each time step. However, the time step size is constrained by stability requirements.

6.6 Pseudo-Transient

On the surface, this technique appears to be a modification of the Transient approach with the residual equation consisting of the true steady-state residual and the Jacobian corresponding to that for the steady-state residual with a user-controlled modification to the diagonal. In fact, the method can be obtained by treating time t as simply a parameter (pseudo-time), evaluating the spatial residual of eq. (6.20) at the updated state of the unknowns with this residual approximated to first order and using a finite-difference approximation for the (pseudo-) time-derivative. Denoting pseudo-time steps using subscripts, let

$$\Delta \mathbf{W}^n = \mathbf{W}^{n+1} - \mathbf{W}^n \quad . \tag{6.21}$$

We then have

$$\mathbf{R}(\mathbf{W}^{n+1}) \approx \mathbf{R}(\mathbf{W}^n) + \frac{\partial \mathbf{R}(\mathbf{W}^n)}{\partial \mathbf{W}} \Delta \mathbf{W}^n \tag{6.22}$$

which can be substituted for $\mathbf{R}$ into eq. (6.20) along with a discretization of the pseudo-time derivative to yield

$$\frac{\Delta \mathbf{W}^n}{\Delta t} = -\mathbf{R}^n - \frac{\partial \mathbf{R}^n}{\partial \mathbf{W}} \Delta \mathbf{W}^n \tag{6.23}$$

where $\mathbf{R}^n \equiv \mathbf{R}(\mathbf{W}^n)$. Letting $k = 1/\Delta t$ and rearranging gives

$$\tilde{\mathbf{J}} \Delta \mathbf{W}^n = -\tilde{\mathbf{F}}^n \tag{6.24}$$

as defined in Table 6.1.

Our implementation of the Pseudo-Transient technique is also different from a strict extension of the Transient technique. Whereas each time step in the Transient approach can require several nonlinear iterations for convergence, we apply one and only one nonlinear iteration for each Pseudo-Transient step. This is due to the fact that Pseudo-Transient is in reality an ineact Newton method for the steady-state problem in which the Jacobian does not correspond exactly to the residual but is instead adjusted to improve the quality of the Jacobian matrix and thereby facilitate the linear solve. We also choose to increase the pseudo-time step (decrease k) at each step as follows:

$$\Delta t^n = f_{CFL}^n \Delta t^0 \tag{6.25}$$

where $f_{CFL} > 1$ is a ramp factor. For some problems with complex physics, *eg.*, shocks, turbulence, etc., use of a monotonically increasing Δt can lead to catastrophic failures such as NaN, resulting from the nonlinear solver attempting to sample inadmissible solution states (*i.e.* negative density). The nonlinear solver library NOX provides checks for such failures and allows the application code, Premo, opportunity to recover from these without terminating the simulation. In this work, we choose to simply reduce the pseudo-time step by half and resume the solution process with the last good solution vector. Hence, for catastrophic failures,

$$\Delta t^n = 0.5 \Delta t^{n-1} \tag{6.26}$$

$$\mathbf{W}^n = \mathbf{W}^{n-1} \quad . \tag{6.27}$$

6.7 Homotopy

Homotopy originates from ideas associated with parameter continuation and is closely related to Pseudo-Transient. The idea is to introduce a fictitious continuation parameter λ and a representative solution state $\mathbf{W}_a$. A simple linear problem corresponds to $\lambda = 0$ and the final steady-state problem to $\lambda = 1$. A sequence of problems begins with $\lambda = 0$ and steps through values of λ, choosing step sizes adaptively based on how difficult the previous nonlinear problem in the sequence was to obtain.

We employ Homotopy using an additional Library Of Continuation Algorithms (LOCA), also being actively developed at Sandia National Laboratories, which builds on top of NOX and uses NOX for its nonlinear solves. More information about continuation methods (for both real and homotopy parameters) can be found from Salinger et al [192].

6.8 Newton-Krylov Based Solver

Regardless of which of the strategies of Table 6.1 is adopted, all require systems of nonlinear algebraic equations to be solved for each problem in the sequence leading to steady-state. Each nonlinear iteration is of the form

$$\tilde{\mathbf{J}}^n \Delta \mathbf{W}^n = -\tilde{\mathbf{F}}^n \tag{6.28}$$

where the expressions of the Jacobian $\tilde{\mathbf{J}}$ and residual $\tilde{\mathbf{F}}$ are given in the table. In accordance with Newton's method, the Jacobian reflects sensitivities of each equation $\tilde{F}_i$ with respect to each field variable W_j,

$$\tilde{\mathbf{J}}^n_{ij} = \frac{\partial \tilde{F}_i(\mathbf{W}^n)}{\partial W^n_j}, \quad 1 \leq i, j \leq N \tag{6.29}$$

for a system with N equations and N variables. Iterations are continued until a user-specified convergence is achieved, eg $||\tilde{\mathbf{F}}^n|| \leq \epsilon_a$, $||\tilde{\mathbf{F}}^n||/||\tilde{\mathbf{F}}^0|| \leq \epsilon_b$, $||\Delta \mathbf{W}^n|| \leq \epsilon_c$, etc.

Each nonlinear iteration of eq. (6.28) requires a linear system of equations to be solved. For very large problems, eg. $N \geq 10^5$, memory and operation counts become prohibitive for direct solvers. Iterative methods circumvent these and are more naturally suited to implementation on massively parallel architectures where data is distributed among $O(1000)$ processors. However, iterative methods can exhibit extremely slow convergence behavior for ill-conditioned or poorly scaled problems. The success of solving eq. (6.28) with iterative methods therefore relies on adequate preconditioning which reduces the number of linear solve iterations without incurring excessive cost in forming and applying the preconditioner.

The work presented here is based on solving eq. (6.28) using the iterative method GMRES [190] and various means of preconditioning. We select GMRES in order to accommodate nonsymmetric Jacobian operators arising from upwinding methods. We consider several choices for forming and/or applying the action of the Jacobian and preconditioner operators. These choices are described next.

6.9 Matrix-Free Operator (MF)

The matrix-free operator is used exclusively as a Jacobian operator and is based on the matrix-free Newton-Krylov ideas described by Brown and Saad [44]. Matrix-free provides the action of the Jacobian on a vector using a directional derivative approximated via finite differences. This is accomplished using two function evaluations to approximate the matrix-vector product needed for each iteration of the linear solve in eq. (6.28). Hence, the linear solve involves repeated action of the Jacobian on a vector, *eg*., $\mathbf{p}$, (used to generate each Krylov vector), which can be approximated as follows:

$$\tilde{\mathbf{J}}\mathbf{p} \approx \frac{\tilde{\mathbf{F}}(\mathbf{W} + \epsilon \mathbf{p}) - \tilde{\mathbf{F}}(\mathbf{W})}{\epsilon} \quad . \tag{6.30}$$

Because the matrix-free operator is based on a directional derivative in which all equation sensitivities to all problem variables are determined together, all terms described in the Algorithmic Formulation section which contribute to the residual, including those from second-order spatial discretization, are taken into account. This operator is the only one we consider in which second-order spatial effects are taken into account.

6.10 Finite-Difference Coloring Operator (FDC)

Whether used as a numerical approximation to the Jacobian or as a preconditioner or both, the matrix $\tilde{\mathbf{J}}$ can be explicitly formed using divided differences in which each component, $\tilde{J}_{ij}$, is formed by perturbing variable W_j, evaluating equation $\tilde{F}_i$ and evaluating a divided difference,

$$\tilde{J}_{ij} \approx \frac{\tilde{F}_i(\mathbf{W} + \epsilon \mathbf{e}_j) - \tilde{F}_i(\mathbf{W})}{\epsilon} \quad . \tag{6.31}$$

Forming the numerical approximation in a brute-force fashion would require N^2 function evaluations which quickly becomes prohibitive for $N \geq O(10^3)$. An alternative is to group variables into colors in which every function depends on *at most* one variable of a particular color. This allows all variables of a given color to be perturbed simultaneously leading to a cost for numerically forming $\tilde{\mathbf{J}}$ of cN function evaluations with c being the number of colors. The number of colors reflects the degree of variable interaction within a problem discretization and, for the discretization of this work, results in $c < 100$ regardless of mesh refinement. The grouping into colors represents a one-time overhead which is amortized over the course of the simulation. Application of matrix coloring transforms eq. (6.31) into the following,

$$\tilde{J}_{ij(m)} \approx \frac{\tilde{F}_i(\mathbf{W} + \epsilon \mathbf{c}_m) - \tilde{F}_i(\mathbf{W})}{\epsilon} \quad , \tag{6.32}$$

where $\mathbf{c}_m$ is the m^{th} color vector of length N with ones in positions associating a variable with this color, and subscript $j(m)$ reflects a mapping from locations in the color vector to columns in the Jacobian matrix indicating to which column the divided difference value computed with this color belongs.

Our application of finite-difference coloring is restricted to matrix structure and values corresponding to first-order spatial discretization. This results in proper accounting of all equation sensitivities to problem variables for spatially first-order problems but represents an approximation to spatially second-order problems. For this reason, the finite-difference coloring operator is used only as a preconditioner for second-order problems and as both the Jacobian and preconditioner operators in spatially first-order problems. The coloring algorithm is based on the greedy algorithm [191] and is implemented in NOX along with other refinements such as adaptive selection of the perturbation parameter ϵ for each variable in order to mitigate the effects of poor problem scaling.

We note that it is possible to use the explicitly computed matrix based on first-order spatial discretizations as both the Jacobian and preconditioner for spatially second-order problems, a technique sometimes referred to as *Deferred Correction*. While this represents another type of inexact Newton method which we could and will explore in future studies, we do not address it here to prevent our presentation from becoming overly lengthy.

6.11 Approximate Analytic Operator (AD)

A third type of operator having potentially many forms consists of analytic expressions for the partial derivatives of eq. (6.29). In general, complexity of the expressions, non-differentiability of some terms and needed flexibility to interchange many choices of limiters, gradient operators, etc. makes computing complete analytic expressions for all anticipated scenarios impractical or impossible. We consider three approximations to the analytic Jacobian of the first-order discretization including the exact expressions

evaluated using automatic differentiation [98]. The other two approximations neglect terms that are contained in the exact edge expressions.

6.12 Preconditioning

Having summarized the choices for both preconditioning and Jacobian operators we wish to briefly comment here on our means of preconditioning. When either the approximate analytic operator (sometimes evaluated using automatic differentiation) or the finite-difference coloring operator is used as the preconditioner, eq. (6.28) is solved using right-preconditioning [190] as follows,

$$\tilde{\mathbf{J}}\mathbf{M}^{-1}\mathbf{y} = -\tilde{\mathbf{F}} \tag{6.33}$$

$$\mathbf{M}\Delta\mathbf{W} = \mathbf{y} \tag{6.34}$$

where $\mathbf{M}$ is the preconditioning operator. When preconditioning is applied to solving eq. (6.28) using the matrix-free Jacobian operator, eq. (6.30) takes the modified form

$$\tilde{\mathbf{J}}\mathbf{M}^{-1}\mathbf{p} \approx \frac{\tilde{\mathbf{F}}(\mathbf{W} + \epsilon\mathbf{M}^{-1}\mathbf{p}) - \tilde{\mathbf{F}}(\mathbf{W})}{\epsilon}. \tag{6.35}$$

and each finite-difference based matrix-vector operation now requires application of the preconditioning operator $\mathbf{M}$. For this work, we use incomplete LU factorization with no fill-in (ILU0) of the matrix $\mathbf{M}$ as the preconditioner for the iterative linear solver. The linear solve is performed using the generalized minimum residual (GMRES [190]) iterative method. Both GMRES and application of the preconditioner are performed using the AztecOO linear solver library [15] via an interface provided by NOX.

Results

Spatially First-Order Subsonic Inviscid Flow

We begin by presenting a comparison of explicit and implicit time integrations to a steady-state for a relatively simple subsonic inviscid flow discretized spatially to first-order. Specifically, we consider subsonic flow over a NACA 0012 airfoil at zero angle-of-attack. The freestream conditions are $Ma = 0.6$, $T_\infty = 300$ K, $P_\infty = 101325$ N/m^2, $R = 287.0$ J/kgK, and $\gamma = 1.4$. The ideal gas assumption is used. An O-type mesh containing $129 \times 33 \times 2$ nodes is used. Our flow solver requires that the mesh contain at least one element in each of three dimensions so that what in this case could be solved using only two-dimensions is solved here on a three-dimensional mesh (though only a single element thick). The infinity boundary was placed approximately 25 chord lengths from the airfoil. All test cases have been run in serial on a 2.4GHz Pentium4 based PC running Linux with Gnu gcc3.1 compilers. For this configuration, the flow field is relatively smooth and no shock waves form. The explicit time integration results employ a one-stage Runge-Kutta scheme with time step size based on $CFL = 1$. The implicit time integrations use a backward Euler temporal discretization and are distinguished by the Jacobian and preconditioner operators used to solve each nonlinear problem occurring a each time step.

Figure 6.1 shows convergence histories for five different simulations. The five cases can be distinguished by the following symbols; **AD_NK_O1** uses the exact analytic Jacobian for the preconditioning matrix and the Jacobian operator; **AD_MF_O1** uses the exact analytic Jacobian for the preconditioning matrix and the matrix-free option for the Jacobian operator; **FDC_MF_O1** generates the preconditioning matrix using

Table 6.2. Euler 1^{st} order Results Summary I.

Method	**Jac/Prec**	**Time**	**Final** $||R||$
Exp Δt	N/A.	32.1 hrs	$1.8e^{-8}$
Imp Δt	AD/AD	40 min	$1.0e^{-12}$
Imp Δt	AD/MF	70 min	$1.0e^{-12}$
Imp Δt	FDC/MF	115 min	$1.0e^{-12}$

Table 6.3. Euler 1^{st} order Results Summary II.

Method	**Jac/Prec**	**Time**	**Final** $||R||$
Exp Δt	N/A.	32.1 hrs	$1.8e^{-8}$
Pseudo Δt	FDC/FDC	49.5 min	$8.9e^{-13}$
HomoT	FDC/FDC	34.4 min	$6.5e^{-16}$
HomoT	AD/AD	14.2 min	$6.5e^{-14}$

finite-difference-coloring and uses the matrix-free option for the Jacobian operator, **NO_PREC_MF_O1** uses no preconditioner and the matrix-free option for the Jacobian operator; and finally, **EXP_O1** used explicit time integration.

All implicit time integrations use an adaptive time step based on the ratio of L_1 spatial residual norms $||\mathbf{R}||_1$ and are constrained to lie within the range $[1 : 5000]$, i.e.

$$\Delta t^{n+1} = \Delta t^n \frac{||\mathbf{R}^{old}||_1}{||\mathbf{R}^{new}||_1} \tag{6.36}$$

where old and new are typically $n-1$ and n time states respectively. The perturbation parameter ϵ used in the matrix-free differencing is $\epsilon = 10^{-6}$. For each nonlinear iteration, the linear problem, solved using GMRES, is considered converged when the linear residual has been reduced to 10^{-4} of its initial value, and the nonlinear iteration is considered converged when the nonlinear residual norm is either reduced by a factor of 10^8 or attains a value of $5E-1$. All preconditioning involves right preconditioning using the ilu(0) factors of the preconditioning matrix.

The results of fig. 6.1 are summarized in Table 6.2 with the exception of the unpreconditioned implicit time integration simulation which terminated when the nonlinear solve failed to reach convergence. This comparison of explicit and implicit time integration to steady-state demonstrates the dramatic improvement in time to solution achievable using an implicit-based approach as well as the necessity of adequate preconditioning in these methods.

Using the same inviscid flow problem, we next extend our comparison to strategies that no longer preserve time-accurate solutions but instead seek significant savings in cpu time. These are the strategies of Table 6.1, and their performance relative to the time integration stategies can be seen in Table 6.3. All of these techniques use an explicitly formed Jacobian (as opposed to matrix-free) and utilize the same matrix as the preconditioner. Notably, all demonstrate comparable or better reductions in cpu time compared to explicit time integration.

Spatially Second-Order Flows

For this and the remainder of the results section, we use second-order spatial discretizations. Among other things, this has the effect of expanding the influence of variables at a node to its second nearest neighbors (by virtue of the gradient computation). For our flow code, Premo, based on unstructured meshes, this presents a difficult challenge in trying to determine all equation sensitivities to all variables as needed for a complete Jacobian matrix. However, the nature of the directional derivative used in the matrix-free Jacobian operator does account for these sensitivities. Consequently, we utilize this Jacobian operator for the remainder and focus our attention on the performance of preconditioners in terms of cost and robustness. A further result of this choice is that the absence of an explicitly formed Jacobian matrix precludes our present implementation of Homotopy which requires the matrix in order to modify its diagonal. We will rectify this deficiency and report new results in the future. So, all subsequent results utilize a matrix-free Jacobian operator.

In fig. 6.2 the convergence histories for five different transient simulations are shown for the second-order discretization. The second-order cases are referred to by the following symbols; **AD_MF_O2** uses the exact first-order analytic Jacobian for the preconditioning matrix (evaluated by automatic differentiation) and the matrix-free option for the Jacobian operator, **ROE_MF_O2** uses the first two term of the exact first-order analytic Jacobian for the preconditioning matrix (eq. 6.15) and the matrix-free option for the Jacobian operator, **SD_MF_O2** uses the first term and a symmetric dissipation term based on the spectral radius of the flux Jacobian matrix for the preconditioning matrix (eq. 6.17) and the matrix-free option for the Jacobian operator, **FDC_MF_O2** uses the finite-difference-coloring of the first-order discretization to form the preconditioning matrix and the matrix-free option for the Jacobian operator, and **EXP_O2** uses explicit time integration. Again, a significant gain in efficiency compared to the explicit method is achieved for all of the implicit methods. The **FDC_MF_O2** matrix is the most expensive to compute, requiring ~ 60 spatial residuals and this reflects in the overall cpu time. It should be noted that we have tried using a "modified preconditioning" technique where the preconditioning matrix is recomputed less frequently than once per nonlinear iteration with some success, however, we have not used that option in the present case. Of the three cases using an analytic preconditioning matrix, the exact edge-based (**AD**) is significantly more expensive to evaluate than the other two approximate methods, however it is a better estimate of the true Jacobian and results in lower overall cpu times.

In addition to using a matrix-free Jacobian operator, all subsequent results of this work are obtained using Pseudo-Transient to achieve steady-state solutions.

Transonic Inviscid Flow at Angle-Of-Attack

We now present results for a transonic flow over the NACA0012 airfoil obtained using three different preconditioners. The flow is represented by $Ma = 0.8$ with the airfoil at 1.25^o angle of attack. We use the van Albada limiter with limiter coefficient of $1.0E-8$. For each nonlinear iteration the linear problem convergence is set to 10^{-4}, and Pseudo-Transient stepping is continued until the steady-state residual norm is reduced to 10^{-12} of its initial value, $||\mathbf{R}^n||_2/||\mathbf{R}^0||_2 \leq 10^{-12}$. Two analytic-based preconditioners are considered: AD represents a complete accounting of all terms for a first-order spatial discretization of the Euler equations computed using forward automatic differentiation (eq. 6.14), and ROE represents only the first two terms computed analytically (eq. 6.15). The third preconditioner operator, FDC, computes the same matrix as AD but via finite-differnces (using coloring). Note that AD and FDC preconditioner matrices are the same as those for the previous results for first-order subsonic Euler but now provide an inexact representation of the Jacobian operator. Use of FDC with the matrix-free (MF) Jacobian operator differs from AD and ROE in a couple of ways. First, it is recomputed infrequently, once every 20 Pseudo-Transient steps for this problem. This is done to amortize the cost of computing the matrix.

Table 6.4. Transonic Inviscid flow over NACA 0012 airfoil; Ma=0.8, $\alpha = 1.25$ deg.

Prec.	f_{CFL} (max)	Freq.	L Its.	NL Its.	CPU sec.
FDC	1.20 (5E3)	20	27312	307	20820
AD	1.05 (1E4)	1	24128	377	11226
ROE	1.05 (1E3)	1	43145	*1000	17940

Second, the sensitivity of the transonic Euler flow to numerical error requires that FDC (and the MF) use second-order accurate centered differences to achieve convergence to 12 orders reduction in residual norm. The convergence history associated with the preconditioner choices is shown in fig. 6.3 and solution statistics are summarized in Table 6.4. The rate of convergence using FDC is steeper than for AD and ROE due to the more aggressive ramp in step size f_{ROE}. However, the accrual of numerical error associated with FDC allows the more accuratre AD to catch up. The cheaper cost of computing the preconditioner matrix using AD as well as its better quality (due to absence of numerical error) results in a reduction in total time to solution to almost half of that for FDC. The quality can be seen by the average number of inner iterations (each iteration for the linear solve) being 64 for AD and 89 for FDC. The ROE preconditioner is very cheap to compute but insufficiently captures the character of the MF Jacobian such that it ultimately fails to reach steady-state within the alloted number of Pseudo-Transient steps.

Supersonic Inviscid Flow We now increase the flow to $Ma = 2.0$, revert to zero angle of attack and repeat the comparison of preconditioner operators just described. We again define linear solve tolerance to be 10^{-4} but now only require a reduction in spatial residual norm of 10 orders of magnitude for convergence of the Pseudo-Transient run. We found the ROE preconditioner to be inadequate to obtain converged steady-states for this more difficult problem. Hence, we report performance statistics for AD and FDC preconditioners in Table 6.5. Convergence behavior is shown in fig. 6.4.

Though AD is inexpensive to comopute relateive to FDC, we attempt to improve the cpu time by infrequent computation of the AD preconditioner. However, this is observed to actually worsen overall solution time by making the linear solves more difficult. Accrued errors in the linear solves also leads to a few more Pseudo-Transient steps being required. The convergence beavior exhibits noise toward the end of the run and is most likely due to chatter of the limiter, which for all but the last run in Table 6.5 uses a limiter coefficient of $1.0E-8$. The last FDC run in the table is marked by an asterisk and uses a limter coefficient of 0.1. The effect of this limiter coefficient value is clearly seen in the convergnce plot in which the chatter is eliminated.

Laminar Viscous Flow

The next test case is laminar flow over NACA 0012 airfoil at zero angle-of-attack. The Flow conditions were; $Ma = 0.2$, and $Re = 5000$. Linear solver tolerance was $1.0E-4$ and the nonlinear tolerance was $1.0E-8$. The minimum CFL was $CFL_{min} = 1.0$. The mesh was $193 \times 65 \times 2$, with normal spacing equals $1.0e-4$ in chord units. There were 129 nodes on the airfoil surface. The wake region contained 10

Table 6.5. Inviscid flow over NACA 0012 airfoil with Ma=2.0 and zero angle-of-attack.

Prec.	f_{CFL} (max)	Freq.	L Its.	NL Its.	CPU sec.
AD	1.05 (5E3)	1	7772	231	3732
AD	1.05 (5E3)	20	9510	255	4053
FDC	1.2 (5E3)	20	7256	136	3347
*FDC	1.2 (5E3)	20	6892	141	2896

Table 6.6. Laminar flow over NACA 0012 airfoil at zero angle-of-attack.

Prec.	f_{CFL} (max)	Freq.	L Its.	NL Its.	CPU sec.
FDC	1.1 (1E7)	20	13347	137	21002
AD	1.1 (1E7)	20	11861	137	22493
AD	1.1 (1E7)	1	11209	137	18373
ROE	1.1 (1E5)	1	13949	153	25860

chords with 33 mesh points. The farfield was placed 25 chords away from the airfoil. The predicted drag coefficient was $Cd = 0.0505$ compared with a value $Cd = 0.0547$ obtained by Knight [135]. Table 6.6 presents a summary of the solver parameters and fig. 6.5 presents the history of convergence. While the nonlinear convergence is amost identical in all three cases, because the quaility of the AD preconditioner, fewer linear iterations were required and the total cpu time was a minimum.

Laminar Viscous Flow At Angle-Of-Attack The next solver test the solution laminar flow over the NACA 0012 airfoil at angle-of-attack. The same mesh as was used in the previous section was also used here. The flow conditions were $Ma = 0.8$, $Re = 5000$ and $\alpha = 10$ degrees. Linear tolerance was set at $1.0E-4$ and the nonlinear tolerance was $1.0E-8$. The minimum CFL was $CFL_{min} = 10.0$. The force coefficients were compared with Forsyth and Jiang [83], case A2. These authors report $Cl = 0.4393$ and $Cd = 0.2708$. Our predictions are, $Cl = 0.4197$ and $Cd = 0.1891$. Table 6.7 presents a summary of the solver parameters. Note that the table reports statistics for each run to the point it reaches the same residual norm reduction as the ROE case. Figure 6.6 presents the history of convergence and fig. 6.7 presents the history of convergence of the force coefficients. This shows that a large CFL initially is benificial however, later if it is too large, it becomes detrimental to the linear convergence.

Turbulent Flow At Angle-Of-Attack The final test of the solver is the simulation of turbulent flow over a NACA 0012 airfoil at angle-of-attack. The flow conditions were; $Re = 1.86 \times 10^6$, $Ma = 0.3$ and

Table 6.7. Laminar flow over NACA 0012 airfoil at $\alpha = 10$ deg.

Prec.	f_{CFL} (max)	**Freq.**	**L Its.**	**NL Its.**	**CPU** sec.
FDC	1.2 (1E6)	25	11322	83	37171
AD	1.1 (5E4)	1	13705	220	114930
ROE	1.1 (1E4)	1	43587	765	56156

$\alpha = 3.59$ degrees. The Splart-Allmaras model equation was solved fully coupled with the Navier-Stokes equations. The model equation is presented in the appendix. For this test case, a c-mesh was used containing $225 \times 81 \times 2$ nodes, with normal spacing next to the surface, $\Delta n = 1.0 \times 10^{-5}$ and the initial tangential spacing at the leading edge was $\Delta t = 5.0 \times 10^{-4}$ in chord units. There were 161 nodes on the surface, and the wake region was 10 chord lengths and contained 33 grid nodes. Experimental data was obtained from Luo *et al*. [151], and the experimental data can also be found in [5]. The FDC preconditioner operator was used without any modification to solve this flow problem. Convergence histories from two simulations with $CFL_{max} = 500, 1000$ and modified preconditioner frequency, 10 and 20 respectively, are presented in fig. 6.8. The convergence is nearly identical until the maximum timestep size is reached. Soon after, the simulation having the larger CFL diverges. As the eddy viscosity evolves near the surface, stiffness from source terms make it harder to solve. The linear tolerance was loosened to $1.0E-3$ and the total cpu time was 63419 sec. Six orders-of-maginitude reduction in residual was adequate to converge the lift coefficient as seen in fig. 6.9. This behavior is similar to that reported by Anderson and Bonhaus [10]. A better solution strategy may be one that employs an adaptive CFL like that used in the transient results reported eariler. The prediction with experimental data in the form of surface pressure coefficient is presented in fig. 6.10. The suction peak is significantly under predicted though the prediction aft of the quarter chord is in good aggreement. We anticipate that the agreement will improve with grid refinement. The important point of this test is that the FDC/MF solver required no modifications in order to solve a problems with additional equations and volume source terms.

Conclusions

Our work has demonstrated the feasibility of various strategies to obtain steady-state solutions for a broad array of compressible flow problems on unstructured meshes. We demonstrated significant reductions in time to solution using strategies other than explicit or implicit time integration including Pseudo-Transient and Homotopy. The success of these depends strongly on the choice of Jacobian and preconditioner operators available, the details of their implentations and some degree of user-tuning. A vast array of choices is provided by interfacing to a nonlinear solver package, NOX, and enhancements such as Homotopy are available by easy expansion of the interface to LOCA, a Library of Continuation Algorithms.

We have shown a small sampling of the many trade-offs between robustness and efficiency for a representative set of compressible flows. Our experience has been that Pseudo-Transient stepping employing matrix-free Jacobian operator with an adequate preconditioner provides the most robustness with efficiency primarily determined by the choice and implementation of the preconditioner operator. Much work remains to optimize the performance of our curent approaches, *eg*., better perturbation parameter selection, using the preconditioning matrix with more (ILUT) or less (block diagonal) detail, using automatic/adaptive strategies for Pseudo-Transient step sizes, linear solver tolerance, etc. We plan on

making progress in these directions with results forthcoming.

Several options are not explored in this work such as Deferred Correction, in which an inexact Jacobian is used as the Jacobian operator, as well as the performance of Homotopy using a matrix-free Jacobian operator. Issues associated with parallel implementations represents another level that will recieve future attention. This work represents a good starting point for such investigations and provides in its current form a capability for addressing challenging problems of interest both to Sandia and the compressible flow community.

Appendix

This appendix shows the Spalart-Allmaras turbulence model used in the turbulent flow simulations [198]. The model equation is given by,

$$\begin{aligned}&\tfrac{\partial \bar{\rho}\hat{\nu}}{\partial t} + \tfrac{\partial}{\partial x_i}(\bar{\rho}\tilde{u}_i\hat{\nu}) \\ &= -C_{\omega 1} f_\omega \bar{\rho} \left(\tfrac{\hat{\nu}}{d}\right)^2 + C_{b1}\bar{\rho}\hat{\nu}\hat{S} \\ &+ \tfrac{\partial}{\partial x_j}\left(\mu_{eff}\tfrac{\partial \hat{\nu}}{\partial x_j}\right) + \tfrac{C_{b2}}{\sigma}\bar{\rho}\tfrac{\partial \hat{\nu}}{\partial x_j}\tfrac{\partial \hat{\nu}}{\partial x_j}\end{aligned}$$

where the turbulent viscosity is given by,

$$\mu_t = \bar{\rho}\hat{\nu} f_{\nu 1} \tag{6.37}$$

and the source terms are,

$$f_{\nu 1} = \frac{\chi^3}{\chi^3 + C_{\nu 1}^3}, \quad f_{\nu 2} = 1 - \frac{\chi}{1 + \chi f_{\nu 1}} \tag{6.38}$$

$$\begin{aligned} f_\omega &= g\left(\frac{1 + C_{\omega 3}^6}{g^6 + C_{\omega 3}^6}\right)^{1/6}, \quad \chi = \frac{\hat{\nu}}{\nu} \\ g &= r + C_{\omega 2}(r^6 - r), \quad r = \frac{\hat{\nu}}{\hat{S}k^2 d^2} \\ \hat{S} &= (2\Omega_{ij}\Omega ij)^{1/2} + \frac{\hat{\nu} f_{\nu 2}}{k^2 d^2} \\ \Omega_{ij} &= \frac{1}{2}\left(\frac{\partial \tilde{u}_i}{\partial x_j} - \frac{\partial \tilde{u}_j}{\partial x_i}\right) \end{aligned} \tag{6.39}$$

with the following list of constants,

$$\begin{aligned} \mu_{eff} &= \bar{\rho}\frac{\nu + \hat{\nu}}{\sigma} \\ k = 0.41, \quad C_{b1} &= 0.1355, \quad C_{b2} = 0.622 \\ \sigma = 2/3, \quad C_{\omega 1} &= \frac{C_{b1}}{k^2} + \frac{1 + C_{b2}}{\sigma} \\ C_{\omega 2} = 0.3, \quad C_{\omega 3} &= 2.0, \quad C_{\nu 1} = 7.1. \end{aligned} \tag{6.40}$$

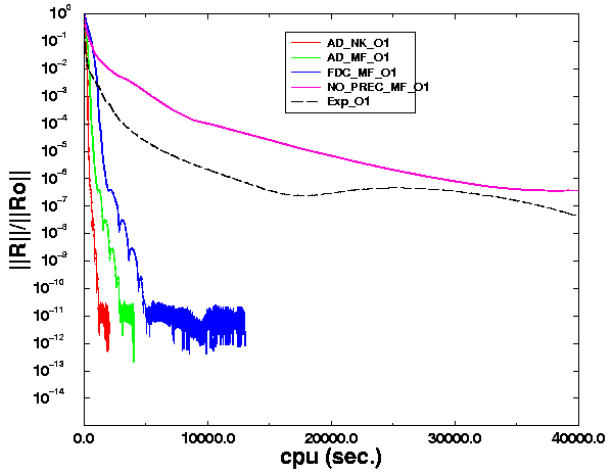


Figure 6.1. Convergence history vs. cpu time for the first-order simulations.

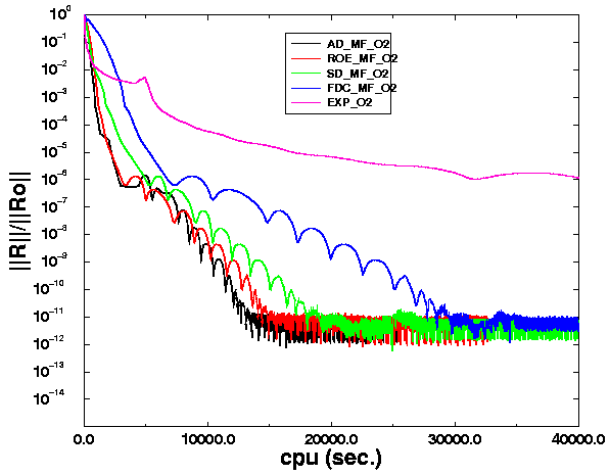


Figure 6.2. Convergence history vs. cpu time for the second-order simulations.

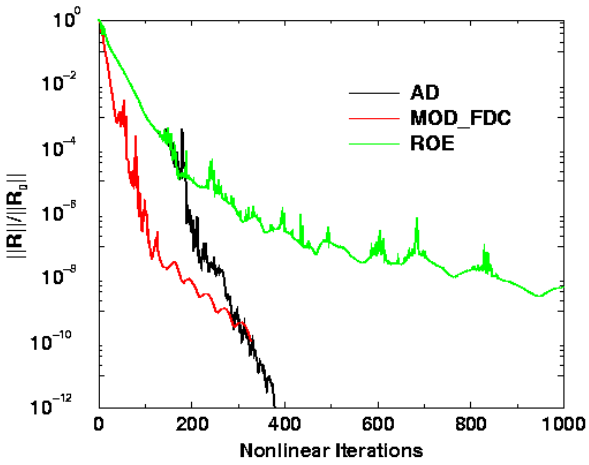


Figure 6.3. Convergence history vs. # nonlinear iterations for inviscid transonic flow over a NACA 0012 airfoil.

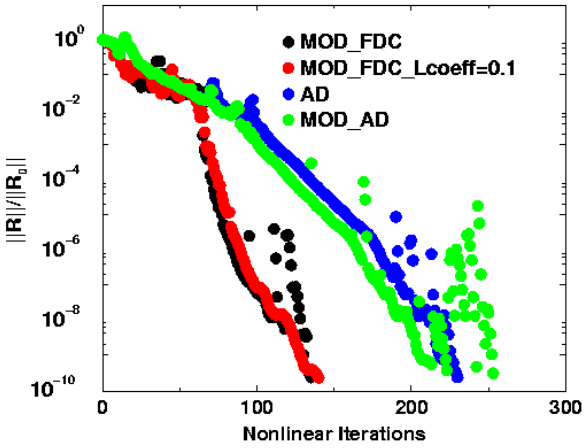


Figure 6.4. Convergence history vs. # nonlinear iterations for the supersonic inviscid flow over a NACA 0012 airfoil with Ma=2.

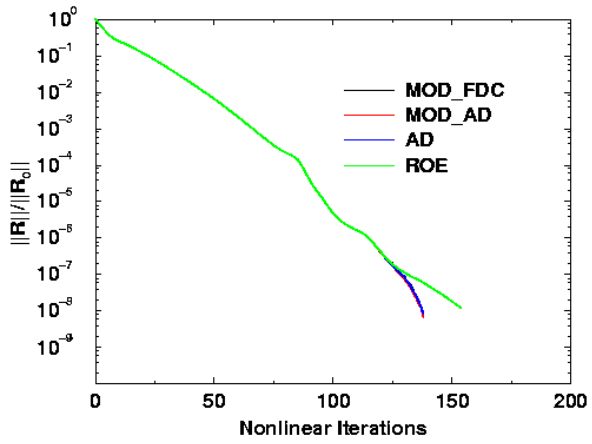


Figure 6.5. Convergence history vs. # nonlinear iterations for laminar flow over NACA 0012 airfoil with $\alpha = 0$ deg.

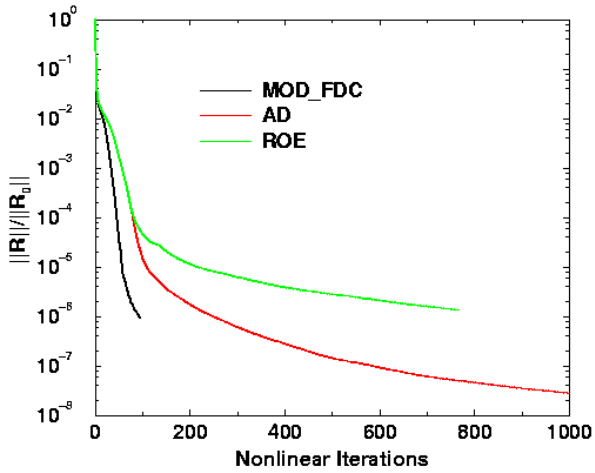


Figure 6.6. Convergence history vs. # nonlinear iterations for laminar flow over NACA 0012 airfoil with $\alpha = 10$ deg.

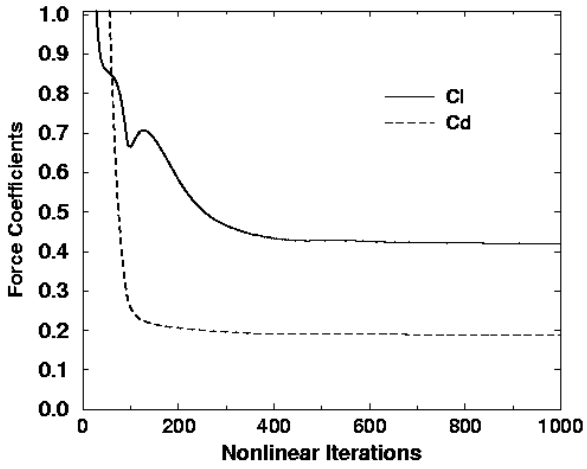


Figure 6.7. Convergence history of the force coefficients vs. # nonlinear iterations.

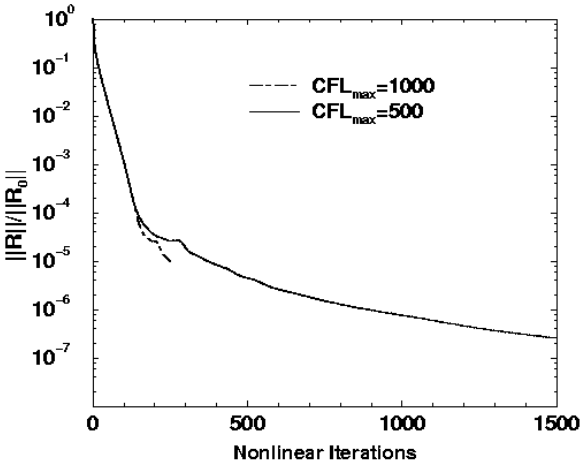


Figure 6.8. Convergence history vs. # nonlinear iterations for turbultent flow over NACA 0012 airfoil.

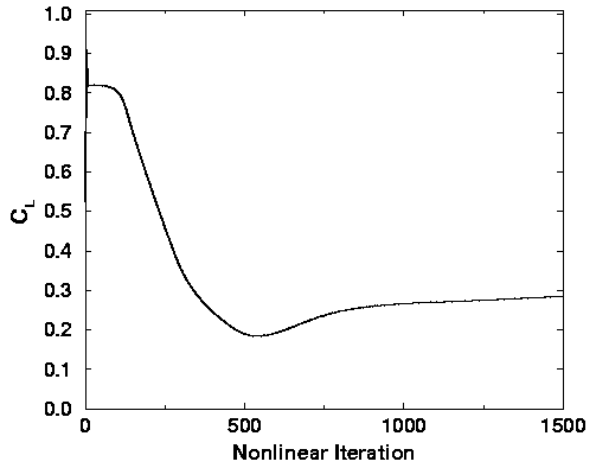


Figure 6.9. Convergence history vs. # nonlinear iterations of the lift coefficient.

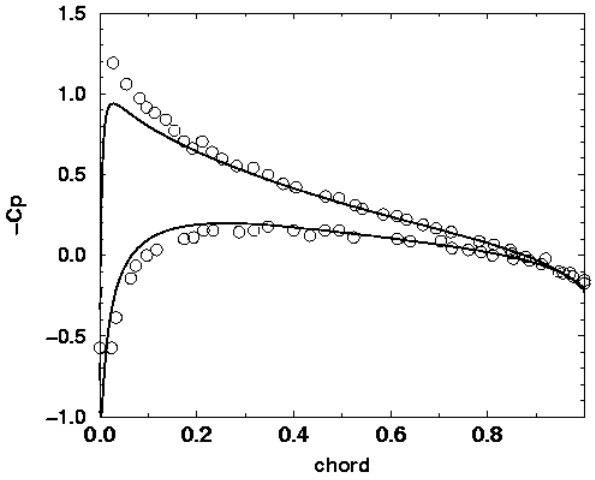


Figure 6.10. Comparison of surface pressure coefficient to experimental data (symbols) found in AGARD, 1979.

Chapter 7

Towards Shape Optimization

7.1 Introduction

The previous chapters discussed the development of hybrid symbolic/AD[1] methods and the computation of an exact first-order approximation for an efficient preconditioner. In this chapter, forward and adjoint linear operators are described for the second-order approximation of the MUSCL[2] scheme. Although actual shape optimization results were not available at the writing of this report, the adjoint formulation for the second-order scheme has been completed and the implementation into Premo has been started. We consider issues involved in formulating and implementing the basic functions and sensitivities for the optimization of a second-order finite-volume method for gas dynamics using the Euler equations.

The general notation for an optimization problem has already been presented in a previous chapter, but is repeated here in a different notation to accommodate additional constraints and derivative terms. Also, our presentation is intended to be precise so that this formulation can be mapped directly to code. In fact, we have developed a Matlab code that uses the same notation as our C++ prototype implementation that uses the hybrid automatic differentiation strategies.

The discretized shape optimization problem can be stated in general form as:

$$\min_{\mathbf{w},\boldsymbol{\alpha}} \quad \phi(\mathbf{w},\boldsymbol{\alpha}) \tag{7.1}$$

$$\text{s.t.} \quad \mathbf{r}(\mathbf{w},\boldsymbol{\alpha}) = 0 \tag{7.2}$$

$$\eta_{l,L} \le \eta_l(\mathbf{w},\boldsymbol{\alpha}) \le \eta_{l,U}, \quad \text{for } l = 1 \ldots N_\eta \tag{7.3}$$

where:

$\mathbf{w} \in \mathbf{R}^{n_w}$ are the state variables,
$\boldsymbol{\alpha} \in \mathbf{R}^{n_\alpha}$ are the shape parameters,
$r(\mathbf{w},\boldsymbol{\alpha}) \in \mathbf{R}^{n_w+n_\alpha} \rightarrow \mathbf{R}^{n_w}$ is the state constraint residual function,
$\phi(\mathbf{w},\boldsymbol{\alpha}) \in \mathbf{R}^{n_w+n_\alpha} \rightarrow \mathbf{R}$ is the objective function,
$\eta_l(\mathbf{w},\boldsymbol{\alpha}) \in \mathbf{R}^{n_w+n_\alpha} \rightarrow \mathbf{R}$ is the l^{th} auxiliary constraint ($l = 1 \ldots N_\eta$), and

[1] automatic differentiation
[2] Monotonic Upwinding Centered Scheme for the Conservation Law Equations

$\eta_{l,L}, \eta_{l,U} \in \mathbf{R}$ are the bounds on the l^{th} auxiliary constraint ($l = 1 \ldots N_\eta$).

Vectors such as $\mathbf{w}$ and $\boldsymbol{\alpha}$ can be viewed as simple objects where each component is a scalar value. The shape optimization process will require shape changes from some reference mesh with node positions $\boldsymbol{x}^r$ to a new mesh $\boldsymbol{x} = \boldsymbol{x}^r + \mathbf{u}$ where $\boldsymbol{x}$ are the new node positions and $\mathbf{u}$ are the displacements from the reference mesh. The mesh nodes are partitioned into boundary and interior nodes with positions $\boldsymbol{x}_I$ and $\boldsymbol{x}_B$, and displacements $\mathbf{u}_I$ and $\mathbf{u}_B$ respectively. The shape of the boundary variables $\mathbf{u}_B$ is changed through a set of implicitly related "shape" parameters $\boldsymbol{\alpha}$.

A range of optimization methods can be considered to solve the shape optimization problem, however, the number of optimization variables are likely to be large enough to warrant an adjoint-based method. For additional details on different approaches to continuous optimization methods we refer the reader to the following reference [174]. In addition, in previous work we identified various levels of optimization implementation approaches and presented computational comparisons that highlight the computational advantages of adjoint and full-space methods [206]. One of the critical computations of the optimization problem is the formation of the reduced gradient of some function $\psi(\mathbf{w}, \boldsymbol{\alpha}) \in \mathbf{R}^{n_w + n_\alpha} \rightarrow \mathbf{R}$

$$\frac{\partial \hat{\psi}}{\partial \boldsymbol{\alpha}} = \frac{\partial \psi}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \boldsymbol{\alpha}} + \frac{\partial \psi}{\partial \boldsymbol{\alpha}} \in \mathbf{R}^{1 \times n_\alpha} \tag{7.4}$$

where

$$\frac{\partial \mathbf{w}}{\partial \alpha} \equiv -\frac{\partial \mathbf{r}}{\partial \mathbf{w}}^{-1} \frac{\partial \mathbf{r}}{\partial \boldsymbol{\alpha}} \in \mathbf{R}^{n_w \times n_\alpha} \tag{7.5}$$

and $\psi(\mathbf{w}, \boldsymbol{\alpha})$ represents the objective function $\phi(\mathbf{w}, \boldsymbol{\alpha})$ or one of the auxiliary constraints $\boldsymbol{\eta}(\mathbf{w}, \boldsymbol{\alpha})$, for $l = 1 \ldots N_\eta$, and $\hat{\psi}(\alpha) = \psi(w(\alpha), \alpha)$

To compute (7.4), the adjoint variables must be first solved as

$$\mathbf{z} = \frac{\partial \mathbf{r}}{\partial \mathbf{w}}^{-T} \frac{\partial \psi}{\partial \mathbf{w}}^{T} \in \mathbf{R}^{n_w} \tag{7.6}$$

followed by

$$\frac{\partial \hat{\psi}}{\partial \boldsymbol{\alpha}} = \mathbf{z}^T \frac{\partial \mathbf{r}}{\partial \boldsymbol{\alpha}} + \frac{\partial \psi}{\partial \boldsymbol{\alpha}} \in \mathbf{R}^{1 \times n_\alpha} \tag{7.7}$$

which requires the adjoint matrix-vector product

$$\boldsymbol{q} = \frac{\partial \mathbf{r}}{\partial \boldsymbol{\alpha}}^{T} \mathbf{z} \in \mathbf{R}^{n_w}. \tag{7.8}$$

Note that there are often auxiliary constraints $\boldsymbol{\eta}(\mathbf{w}, \boldsymbol{\alpha})$ that only involve the design variables (i.e. $\frac{\partial \eta_l}{\partial \mathbf{w}} = 0$). In these cases the adjoint problem in (7.6) need not be solved and the reduced gradient for these constraints is just $\frac{\partial \hat{\eta}_l}{\partial \boldsymbol{\alpha}} = \frac{\partial \eta_l}{\partial \boldsymbol{\alpha}}$. In general, these types of constraints do not significantly increase the expense in solving the optimization problem.

There are several different types of computations that go into the evaluation of the residual, the objective function and the auxiliary constraints. These computations, listed in the order of evaluation for the forward problem, are:

1. **Shape parameterization**

$\mathbf{S}(\boldsymbol{\alpha}, \mathbf{u}_B) : \alpha \rightarrow \mathbf{u}_B$
This procedure specifies changes in shape u_B of the boundary given current values of the shape parameters α. This mapping is performed by an implicit function $\mathbf{S}(\boldsymbol{\alpha}, \mathbf{u}_B)$ where $\frac{\partial \mathbf{S}}{\partial \mathbf{u}_B}$ is nonsingular and $\frac{\partial \mathbf{S}}{\partial \boldsymbol{\alpha}}$ also exists. Note that the number of shape parameters in $\boldsymbol{\alpha}$ must be less than or equal to the number of boundary node positions $\mathbf{u}_B$.

2. **Mesh movement**
$\mathbf{M}(\mathbf{u}_B, \boldsymbol{x}) : \mathbf{u}_B \rightarrow \boldsymbol{x}$
This is a smooth procedure that moves mesh nodes x given a change from the reference mesh boundary shape u_B where $\frac{\partial \mathbf{M}}{\partial \boldsymbol{x}}$ is nonsingular and $\frac{\partial \mathbf{M}}{\partial \mathbf{u}_B}$ exists. Typically a spring and mass procedure is used here to move the mesh nodes, but mesh quality is compromised in certain situations. We have developed a more stable method based on a Laplacian and a Bi-Laplacian mesh movement.

3. **Dual mesh**
$\boldsymbol{n} = \boldsymbol{D}_n(\boldsymbol{x}) : \boldsymbol{x} \rightarrow \boldsymbol{n}$
$V = D_V(\boldsymbol{x}) : \boldsymbol{x} \rightarrow V$
The purpose of the dual mesh is to map characteristics of a finite-element mesh into a finite-volume mesh. These functions must be able to take the new positions of the nodes $\boldsymbol{x}$ and be able to determine the edge normals $\boldsymbol{n}$ (edge-based) and control volumes V (node-based) given some description of the mesh (i.e. different elements). For this mapping, the Jacobian matrices $\partial \boldsymbol{D}_n / \partial \boldsymbol{x}$ and $\partial D_V / \partial \boldsymbol{x}$ must exist.

4. **Residual evaluation**
$\mathbf{r}(\mathbf{w}, \boldsymbol{x}, \boldsymbol{n}, V) : (\mathbf{w}, \boldsymbol{x}, \boldsymbol{n}, V) \rightarrow \mathbf{r}$
This is the evaluation of the node-based residual r given the state variables $\mathbf{w}$, the nodal positions $\boldsymbol{x}$, the edge normals $\boldsymbol{n}$ and the control volumes V. Note that the Jacobian $\frac{\partial \mathbf{r}}{\partial \mathbf{w}}$ must be nonsingular and the Jacobians $\frac{\partial \mathbf{r}}{\partial \boldsymbol{x}}$, $\frac{\partial \mathbf{r}}{\partial \boldsymbol{n}}$ and $\frac{\partial \mathbf{r}}{\partial V}$ must also exist.

Given the above structure of the residual evaluation, using the chain rule, and substituting all the derivative requirements for shape parameterization, mesh movement and the dual mesh, we obtain the expression

$$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \boldsymbol{\alpha}} &= \left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{x}} + \frac{\partial \mathbf{r}}{\partial \boldsymbol{n}} \frac{\partial \boldsymbol{n}}{\partial \boldsymbol{x}} + \frac{\partial \mathbf{r}}{\partial V} \frac{\partial V}{\partial \boldsymbol{x}} \right) \frac{\partial \boldsymbol{x}}{\partial \mathbf{u}_B} \frac{\partial \mathbf{u}_B}{\partial \boldsymbol{\alpha}} \\ &= \left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{x}} + \frac{\partial \mathbf{r}}{\partial \boldsymbol{n}} \frac{\partial \boldsymbol{D}_n}{\partial \boldsymbol{x}} + \frac{\partial \mathbf{r}}{\partial V} \frac{\partial D_V}{\partial \boldsymbol{x}} \right) \frac{\partial \mathbf{M}}{\partial \boldsymbol{x}}^{-1} \frac{\partial \mathbf{M}}{\partial \mathbf{u}_B} \frac{\partial \mathbf{S}}{\partial \mathbf{u}_B}^{-1} \frac{\partial \mathbf{S}}{\partial \boldsymbol{\alpha}}. \end{aligned} \tag{7.9}$$

Consequently the adjoint solve (7.8) can be performed as a set of elementary operations:

$$\begin{aligned} \boldsymbol{q} &= \frac{\partial \mathbf{r}}{\partial \boldsymbol{\alpha}}^T \mathbf{z} \\ &= \frac{\partial \mathbf{S}}{\partial \boldsymbol{\alpha}}^T \frac{\partial \mathbf{S}}{\partial \mathbf{u}_B}^{-T} \frac{\partial \mathbf{M}}{\partial \mathbf{u}_B}^T \frac{\partial \mathbf{M}}{\partial \boldsymbol{x}}^{-T} \left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{x}}^T + \frac{\partial \boldsymbol{D}_n}{\partial \boldsymbol{x}}^T \frac{\partial \mathbf{r}}{\partial \boldsymbol{n}}^T + \frac{\partial D_V}{\partial \boldsymbol{x}}^T \frac{\partial \mathbf{r}}{\partial V}^T \right) \mathbf{z}. \end{aligned} \tag{7.10}$$

From (7.10) it is obvious that many different types of separate adjoint computations (both solves and mat-vecs) are involved in the computation of the overall adjoint matrix-vector product in (7.8). Therefore, we can focus on each atomic adjoint computation separately.

7.2 Residual and state computations

The general formulation for the shape optimization problem as described in the previous section applies to any set of physics (such as Euler, Navier Stokes, structural dynamics, etc). However, we have applied this formulation to the Euler equations and in the following sections we describe the formulation for a node-centered, finite-volume, higher-order, upwinding scheme. This formulation is the same as the in the previous chapter without the viscous forces. It should be noted that the notation is slightly different, primarily because of different authoring efforts and also as a result of specific needs for shape optimization.

7.2.1 The Euler equations

The Euler equations in differential form are

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot \rho \boldsymbol{u} &= 0, \\ \frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) + \nabla p &= 0, \\ \frac{\partial \, \rho E}{\partial t} + \nabla \cdot [\boldsymbol{u}(\rho E + p)] &= 0, \end{aligned} \tag{7.11}$$

in which ρ, $\boldsymbol{u}$, p, and E denotes the density, velocity vector, pressure, and total energy per unit volume, respectively. The coordinate invariant formulation (7.11) holds for space dimension $N_d = 1$, 2, or 3. The equation of state closes the system; for a perfect gas, we have

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho \left|\boldsymbol{u}\right|^2\right).$$

Introducing the conservative variables $\mathbf{w}$ and the flux function $\mathbf{f}$ given by

$$\mathbf{w} = \begin{pmatrix} \rho \\ \rho \boldsymbol{u} \\ \rho E \end{pmatrix}, \quad \mathbf{f} = \begin{pmatrix} \rho \boldsymbol{u} \\ \rho \, \boldsymbol{u} \otimes \boldsymbol{u} + \boldsymbol{I} p \\ \boldsymbol{u} \rho H \end{pmatrix}, \tag{7.12}$$

where $\rho H = \rho E + p$, the Euler equations (7.11) may be written in the compact form

$$\frac{\partial \mathbf{w}}{\partial t} + \nabla \cdot \mathbf{f} = 0. \tag{7.13}$$

Note the local vector $\mathbf{w}$ here can be thought of as one component of the global vector $\mathbf{w}$ in the discretized form; see (7.1)–(7.3). This should not cause confusion since later each local vector will have a nodal designation such as $\mathbf{w}_i$ in the discrete case. Bold italic letters, such as $\boldsymbol{u}$ and $\boldsymbol{I}$, are used for tensors of first and second order, whereas bold upright letters are used for matrices of tensors, such as $\mathbf{w}$ and $\mathbf{f}$. The fluid velocity $\boldsymbol{u}$ is a "vector" in the continuum-mechanics meaning (a first-order tensor), whereas $\mathbf{w}$ and $\mathbf{f}$ are (column) "vectors" in the linear-algebra meaning (matrices with one column). The use of the symbols $\mathbf{w}$ and $\mathbf{f}$ is just for convenience and does not indicate a tensor structure. To prevent confusion, we will minimize, as much as possible, the use of the term "vector", instead referring to *first-order tensors* (for objects like $\boldsymbol{u}$) and *3-by-1 matrices* (for objects like $\mathbf{w}$).

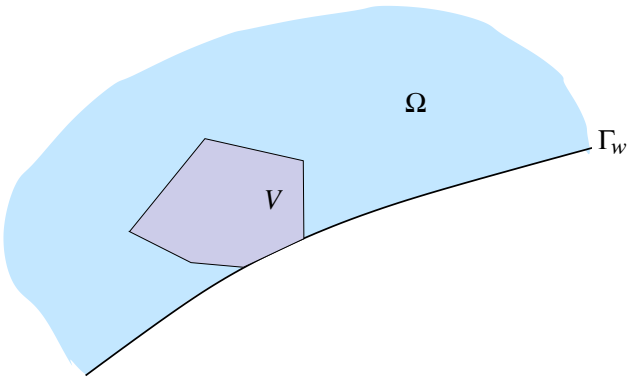


Figure 7.1. A control volume having a nonzero intersection with a solid wall Γ_w.

The flux function may be split in two parts, $\mathbf{f} = \mathbf{f}^{\mathrm{N}} + \mathbf{f}^{\mathrm{w}}$, where

$$\mathbf{f}^{\mathrm{N}} = \begin{pmatrix} \rho \boldsymbol{u} \\ \rho \boldsymbol{u} \otimes \boldsymbol{u} \\ \boldsymbol{u} \rho H \end{pmatrix}, \qquad \mathbf{f}^{\mathrm{w}} = \begin{pmatrix} \mathbf{0} \\ \boldsymbol{I} p \\ \mathbf{0} \end{pmatrix}. \tag{7.14}$$

A solid wall is characterized by the condition of no mass flux through the boundary, that is, $\boldsymbol{n} \cdot \boldsymbol{u} = 0$ ($\boldsymbol{n}$ is the outward-directed unit normal). Thus,

$$\boldsymbol{n} \cdot \mathbf{f} = \boldsymbol{n} \cdot \mathbf{f}^{\mathrm{w}} \quad \text{at wall boundaries.} \tag{7.15}$$

(The dot product of a first-order tensor, such as the boundary normal $\boldsymbol{n}$, with the flux function $\mathbf{f}$ is defined as the dot product applied on each row of $\mathbf{f}$.)

Consider an arbitrary control volume V in the fluid domain Ω; the boundary of V may have a nonempty intersection with a solid-wall boundary Γ_w, as in figure 7.1. Integrating equation (7.13) over V, using the divergence theorem and expression (7.15) yields the integral form of the Euler equations

$$\frac{d}{dt} \int_V \mathbf{w} \, dx + \int_{\partial V \setminus \Gamma_w} \boldsymbol{n} \cdot \mathbf{f} \, d\Gamma + \int_{\partial V \cap \Gamma_w} \boldsymbol{n} \cdot \mathbf{f}^{\mathrm{w}} \, d\Gamma = 0; \tag{7.16}$$

this is the basis for the finite-volume discretization below.

7.2.2 A node-centered finite-volume discretization of MUSCL type

Consider a meshing of the fluid domain Ω consisting of tetrahedrals or hexahedrals. (In 2D, the mesh consists of triangles or quadrilaterals.) Assume that the boundary $\partial\Omega$ can be partitioned into a solid-wall part Γ_w and a free-stream part Γ_∞. Our formulation will support hybrid meshes because we essentially translate the mesh into a set of nodes and edges. Let $\mathcal{V}(\Omega)$ denote the set of mesh nodes in the strict interior of the domain, and let $\mathcal{V}(\Gamma_w)$ and $\mathcal{V}(\Gamma_\infty)$ denote the set of mesh nodes at the wall and at the free-stream, respectively. The set of all boundary nodes may be denoted $\mathcal{V}(\partial\Omega) = \mathcal{V}(\Gamma_w) \cup \mathcal{V}(\Gamma_\infty)$ and the set of all nodes $\mathcal{V}(\overline{\Omega}) = \mathcal{V}(\Omega) \cup \mathcal{V}(\partial\Omega)$.

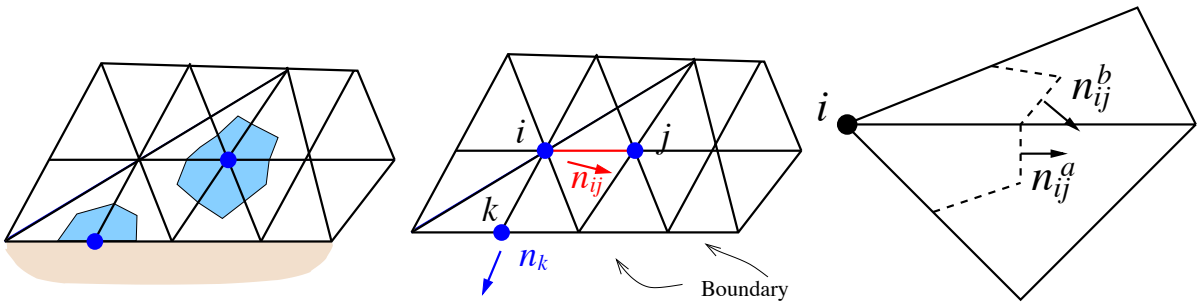


Figure 7.2. Control volume norms in 2D

Left: A control volume associated with an internal node and a control volume associated with a boundary node.

Middle: There are surface vectors n_{ij} associated with each edge, and surface vectors n_k associated with each boundary node.

Right: The surface vectors are defined as the vector sum of pairs of boundary normals associated with the dual mesh: $\boldsymbol{n}_{ij} = \boldsymbol{n}^a_{ij} + \boldsymbol{n}^b_{ij}$, $\boldsymbol{n}_k = \boldsymbol{n}^a_k + \boldsymbol{n}^b_k$. Note that the lengths of the normals are equal to the lengths of corresponding tangents.

Moreover, let $\mathcal{N}_i$ denote the set of nodes being *nearest neighbors* to mesh node i, that is, the nodes that are connected to node i with an edge. We associate a surface vector $\boldsymbol{n}_{ij}$ with each edge $\xrightarrow{ij}$, and we associate a surface vector $\boldsymbol{n}_i$ with each boundary node i. The normals are computed from a dual mesh; in 2D the dual mesh is obtained by joining the edge midpoints with the cell centroid. Figure 7.2 exemplifies the construction of control volumes, surface and boundary normals in 2D. Similar constructions are used in 3D and for other shapes of the control volumes. Note that, by construction,

$$\boldsymbol{n}_{ji} = -\boldsymbol{n}_{ij}. \tag{7.17}$$

Also note that the normals are not of unit length.

Using the control volumes associated with the dual mesh, a node-centered finite-volume discretization may be obtained from the integral form (7.16). In the steady case, we consider

$$\begin{aligned}
&\sum_{j\in\mathcal{N}_i}\left[\frac{1}{2}\left(\mathbf{f}^{i+}_{ij}\cdot\boldsymbol{n}_{ij}\right)+\frac{1}{2}\left(\mathbf{f}^{j-}_{ij}\cdot\boldsymbol{n}_{ij}\right)+\mathbf{d}\left(\mathbf{w}^{i+}_{ij},\mathbf{w}^{j-}_{ij},\boldsymbol{n}_{ij}\right)\right]=\mathbf{0} \qquad \forall i\in\mathcal{V}(\Omega),\\
&\sum_{j\in\mathcal{N}_i}\left[\frac{1}{2}\left(\mathbf{f}^{i+}_{ij}\cdot\boldsymbol{n}_{ij}\right)+\frac{1}{2}\left(\mathbf{f}^{j-}_{ij}\cdot\boldsymbol{n}_{ij}\right)+\mathbf{d}\left(\mathbf{w}^{i+}_{ij},\mathbf{w}^{j-}_{ij},\boldsymbol{n}_{ij}\right)\right]+\boldsymbol{n}_i\cdot\mathbf{f}^{\mathrm{BC}}_i=\mathbf{0} \qquad \forall i\in\mathcal{V}(\partial\Omega),
\end{aligned} \tag{7.18}$$

where we use the notation $\mathbf{f}^{i+}_{ij}=\mathbf{f}(\mathbf{w}^{i+}_{ij})$, $\mathbf{f}^{j-}_{ij}=\mathbf{f}(\mathbf{w}^{j-}_{ij})$ for the flux function evaluated at the *extrapolated states* $\mathbf{w}^{i+}_{ij}$ and $\mathbf{w}^{j-}_{ij}$ (explained below). Explicit expressions for the Roe-dissipation term $\mathbf{d}\left(\mathbf{w}^{i+}_{ij},\mathbf{w}^{j-}_{ij},\boldsymbol{n}_{ij}\right)$ is given in the Appendix, and the boundary flux is

$$\boldsymbol{n}_i\cdot\mathbf{f}^{\mathrm{BC}}_i=\begin{cases}\boldsymbol{n}_i\cdot\mathbf{f}^{\mathrm{w}}_i & i\in\mathcal{V}(\Gamma_w),\\ \frac{1}{2}\left(\mathbf{f}_i\cdot\boldsymbol{n}_i\right)+\frac{1}{2}\left(\mathbf{f}_\infty\cdot\boldsymbol{n}_i\right)+\mathbf{d}\left(\mathbf{w}_i,\mathbf{w}_\infty,\boldsymbol{n}_i\right) & i\in\mathcal{V}(\Gamma_\infty),\end{cases}$$

where $\mathbf{f}^{\mathrm{w}}_i=\mathbf{f}^{\mathrm{w}}(\mathbf{w}_i)$, with $\mathbf{f}^{\mathrm{w}}$ from expression (7.14), and $\mathbf{f}_\infty=\mathbf{f}(\mathbf{w}_\infty)$, where $\mathbf{w}_\infty$ is the fluid state at the free stream.

Node-based objects are given a single subscript such as $\mathbf{w}_i$ and $\mathbf{w}_j$ while edge-based objects are given a double subscript such as n_{ij} and n_{ij}. Additional classifiers are added such as math accents and superscripts (e.g. $\mathbf{w}_{ij}^{i+}$) for further classification.

If zeroth-order (i.e. no) extrapolation is used, that is, $\mathbf{w}_{ij}^{i+} = \mathbf{w}_i$ and $\mathbf{w}_{ij}^{j-} = \mathbf{w}_j$, the scheme (7.18) reduces to a first-order upwind scheme. However, in the MUSCL spirit, we will use a *limited linear reconstruction* to obtain a higher-resolution scheme, that is, we define

$$\begin{aligned} \mu_{i+} &= \mu_i + \phi\left(\Delta^+\mu_i, \Delta^-\mu_i\right) \nabla_h\mu_i \cdot \frac{\boldsymbol{t}_{ij}}{2}, \\ \mu_{j-} &= \mu_j - \phi\left(\Delta^+\mu_j, \Delta^-\mu_j\right) \nabla_h\mu_j \cdot \frac{\boldsymbol{t}_{ij}}{2}, \end{aligned} \tag{7.19}$$

where $\boldsymbol{t}_{ij} = \boldsymbol{x}_j - \boldsymbol{x}_i$ is the tangent vector along the edge pointing from node i to node j. The reconstruction is done component by component μ of either the conservative or primitive variables. Note that these two reconstructions are *not* equivalent. The discrete gradient operator in (7.19) is

$$\nabla_h\mu_i = \begin{cases} \dfrac{1}{|V_i|}\displaystyle\sum_{j\in\mathcal{N}_i} \frac{\mu_i+\mu_j}{2}\boldsymbol{n}_{ij} & i \in \mathcal{V}(\Omega), \\ \dfrac{1}{|V_i|}\displaystyle\sum_{j\in\mathcal{N}_i} \frac{\mu_i+\mu_j}{2}\boldsymbol{n}_{ij} + \frac{1}{|V_i|}\mu_i\boldsymbol{n}_i, & i \in \mathcal{V}(\partial\Omega) \end{cases} \tag{7.20}$$

where $|V_i|$ is the cell volume of the dual control volume V_i centered at node i. Moreover, the *slope limiter* is a real-valued function of two variables $\phi = \phi(a, b)$. Two possible choices of limiters are

$$\phi(a,b) = \begin{cases} \dfrac{ab + |ab|}{a^2 + b^2 + \epsilon} & \text{(van Albada)}, \\ 2\dfrac{ab + |ab|}{(a+b)^2 + \epsilon} & \text{(van Leer)}. \end{cases} \tag{7.21}$$

The arguments of the limiter in expression (7.19) are

$$\begin{aligned} \Delta^+\mu_i &= \Delta^-\mu_j = \mu_j - \mu_i, \\ \Delta^-\mu_i &= 2\nabla_h\mu_i \cdot \boldsymbol{t}_{ij} - (\mu_j - \mu_i), \\ \Delta^+\mu_j &= 2\nabla_h\mu_j \cdot \boldsymbol{t}_{ij} - (\mu_j - \mu_i). \end{aligned} \tag{7.22}$$

The reconstruction of the conservative variables can be summarized in vector form. First, the gradient computation $\mathbf{g}_i = \nabla_h\mathbf{w}_i \in \mathbf{R}^{(N_d+2)\times(N_d)}$ is

$$\mathbf{g}_i \;=\; \begin{cases} \dfrac{1}{2|V_i|}\displaystyle\sum_{j\in\mathcal{N}_i} (\mathbf{w}_i + \mathbf{w}_j) \otimes \boldsymbol{n}_{ij} & i \in \mathcal{V}(\Omega), \\ \dfrac{1}{2|V_i|}\displaystyle\sum_{j\in\mathcal{N}_i} (\mathbf{w}_i + \mathbf{w}_j) \otimes \boldsymbol{n}_{ij} + \frac{1}{|V_i|}\mathbf{w}_i \otimes \boldsymbol{n}_i, & i \in \mathcal{V}(\partial\Omega) \end{cases}. \tag{7.23}$$

Later, it will be convenient to define the functions

$$\mathbf{w}_{ij}^{i+} \;=\; \mathbf{w}_{ij}^{i+}\left(\mathbf{w}_i, \mathbf{w}_j, \mathbf{p}_{ij}^i\right) \tag{7.24}$$

$$\mathbf{w}_{ij}^{j-} \;=\; \mathbf{w}_{ij}^{j-}\left(\mathbf{w}_i, \mathbf{w}_j, \mathbf{p}_{ij}^j\right) \tag{7.25}$$

with $\mathbf{p}_{ij}^i = \mathbf{g}_i \cdot \boldsymbol{t}_{ij}$ and $\mathbf{p}_{ij}^j = \mathbf{g}_j \cdot \boldsymbol{t}_{ij}$. These functions are defined as

$$\begin{aligned}
\Delta_{ij}^+ \mathbf{w}_i &= \mathbf{w}_j - \mathbf{w}_i && (7.26)\\
\Delta_{ij}^- \mathbf{w}_i &= 2\mathbf{p}_{ij}^i - (\mathbf{w}_j - \mathbf{w}_i) && (7.27)\\
\Delta_{ij}^+ \mathbf{w}_j &= 2\mathbf{p}_{ij}^j - (\mathbf{w}_j - \mathbf{w}_i) && (7.28)\\
\Delta_{ij}^- \mathbf{w}_j &= \mathbf{w}_j - \mathbf{w}_i && (7.29)\\
\mathbf{w}_{ij}^{i+} &= \mathbf{w}_i + \tfrac{1}{2}\boldsymbol{\Phi}(\Delta_{ij}^+ \mathbf{w}_i, \Delta_{ij}^- \mathbf{w}_i)\mathbf{p}_{ij}^i && (7.30)\\
\mathbf{w}_{ij}^{j-} &= \mathbf{w}_j - \tfrac{1}{2}\boldsymbol{\Phi}(\Delta_{ij}^+ \mathbf{w}_j, \Delta_{ij}^- \mathbf{w}_j)\mathbf{p}_{ij}^j && (7.31)
\end{aligned}$$

where $\boldsymbol{\Phi}(\Delta_{ij}^+ \mathbf{w}_i, \Delta_{ij}^- \mathbf{w}_i) = \text{diag}\{\phi(\Delta_{ij}^+ \boldsymbol{\mu}_i, \Delta_{ij}^- \boldsymbol{\mu}_i)\}$ is a diagonal matrix containing the component-wise evaluations of the limiter.

Lemma 1. *The following symmetry properties hold.*

$$\mathbf{w}_{ij}^{i+} = \mathbf{w}_{ji}^{i-} \qquad \mathbf{w}_{ij}^{j-} = \mathbf{w}_{ji}^{j+}$$

7.2.3 State computations

7.2.3.1 Gradient and residual

The left sides of equation (7.18) is the *residual* $\mathbf{r}_i$ at node i, which can be written

$$\mathbf{r}_i = \begin{cases} \displaystyle\sum_{j \in \mathcal{N}_i} \mathbf{r}_{ij} & i \in \mathcal{V}(\Omega), \\ \displaystyle\sum_{j \in \mathcal{N}_i} \mathbf{r}_{ij} + \boldsymbol{n}_i \cdot \mathbf{f}_i^{\text{BC}} & i \in \mathcal{V}(\partial\Omega), \end{cases} \tag{7.32}$$

where

$$\mathbf{r}_{ij} = \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ij}^{i+}) \cdot \boldsymbol{n}_{ij}\big) + \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ij}^{j-}) \cdot \boldsymbol{n}_{ij}\big) + \mathbf{d}\big(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \boldsymbol{n}_{ij}\big) \tag{7.33}$$

is the contribution to residual $\mathbf{r}_i$ from the edge $\overset{ij}{\longrightarrow}$. By interchanging indices i and j, we obtain the contribution to residual $\mathbf{r}_j$ from the edge $\overset{ji}{\longrightarrow}$,

$$\begin{aligned}
\mathbf{r}_{ji} &= \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ji}^{j+}) \cdot \boldsymbol{n}_{ji}\big) + \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ji}^{i-}) \cdot \boldsymbol{n}_{ji}\big) + \mathbf{d}\big(\mathbf{w}_{ji}^{j+}, \mathbf{w}_{ji}^{i-}, \boldsymbol{n}_{ji}\big) \\
&= -\frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ij}^{j-}) \cdot \boldsymbol{n}_{ij}\big) - \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ij}^{i+}) \cdot \boldsymbol{n}_{ij}\big) - \mathbf{d}\big(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \boldsymbol{n}_{ij}\big) = -\mathbf{r}_{ij},
\end{aligned} \tag{7.34}$$

where we have utilized Lemma 1.

The computation of the residual can be accomplished in two steps. The first step computes the gradient of each component of the conservative (or primitive) variables, which will be needed in the reconstruction (7.24)–(7.25),

Algorithm 8. GRADIENT COMPUTATION.

1. Set $\mathbf{g}_i \leftarrow \mathbf{0}$ for each $i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\overset{i\,j}{\longrightarrow}$:
   (a) $\mathbf{g}_i \leftarrow \mathbf{g}_i + \dfrac{1}{2|V_i|}(\mathbf{w}_i + \mathbf{w}_j) \otimes \boldsymbol{n}_{ij}$,
   (b) $\mathbf{g}_j \leftarrow \mathbf{g}_j - \dfrac{1}{2|V_j|}(\mathbf{w}_i + \mathbf{w}_j) \otimes \boldsymbol{n}_{ij}$.
3. For each boundary node k:
   (a) $\mathbf{g}_k \leftarrow \mathbf{g}_k + \dfrac{1}{|V_k|}\mathbf{w}_i \otimes \boldsymbol{n}_k$.

In the second loop over edges and boundary nodes, the variables are extrapolated and the residual is computed making use of the skew symmetry property (7.34):

Algorithm 9. COMPUTATION OF THE RESIDUAL.

---

1. Set $\mathbf{r}_i \leftarrow \mathbf{0}$ for each $i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\overset{i\,j}{\longrightarrow}$:
   (a) $\mathbf{r}_i \leftarrow \mathbf{r}_i + \mathbf{r}_{ij}$
   (b) $\mathbf{r}_j \leftarrow \mathbf{r}_j - \mathbf{r}_{ij}$, where
   $$\mathbf{r}_{ij} = \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ij}^{i+}) \cdot \boldsymbol{n}_{ij}\big) + \frac{1}{2}\big(\mathbf{f}(\mathbf{w}_{ij}^{j-}) \cdot \boldsymbol{n}_{ij}\big) + \mathbf{d}\big(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \boldsymbol{n}_{ij}\big)$$
   and where $\mathbf{w}_{ij}^{i+}$ and $\mathbf{w}_{ij}^{j-}$ are computed according (7.24)–(7.25).
3. For each boundary node k:
   (a) $\mathbf{r}_k \leftarrow \mathbf{r}_k + \boldsymbol{n}_k \cdot \mathbf{f}_k^{\text{BC}}$

7.2.3.2 Forward state Jacobian-vector product

Newton-based solvers for equation (7.18) and SAND optimization algorithms require solves with the linear operator $\partial\mathbf{r}/\partial\mathbf{w}$. Typically, linear systems such as this are solved with an iterative method (such as GMRES). An essential computation in an iterative linear solver is the repeated application of the linear operator

$$\delta\mathbf{r} = \frac{\partial\mathbf{r}}{\partial\mathbf{w}}\delta\mathbf{w} \tag{7.35}$$

at some point $(\mathbf{w}, \boldsymbol{\alpha})$ for arbitirary vectors $\delta\mathbf{w}$. Such a linear operator can be implemented by first forming a sparse matrix for $\partial\mathbf{r}/\partial\mathbf{w}$ up front before using it in repeated applications or by using a matrix-free

method. A popular matrix-free method is to use directional-finite differences but it is well know that these methods suffer from serious numerical problems. Instead, here we consider machine-accurate matrix-free methods for assembling (7.35).

One approach for assemblying (7.35) is to apply forward AD to the edge-level C++ functions used in Algorithm 8 and Algorithm 9. While this is relatively efficient and easy to program, more invasive and specialized hybrid symbolic/AD approaches can yield significant improvements in storage and CPU time; see Chapter 5. Below, the symbolic linearized operator is derived.

Let an arbitrary set of states $\mathbf{w}_i$, $i \in \mathcal{V}(\overline{\Omega})$, be given for a set mesh. Linearizing residual expression (7.32) about this fluid state yields

$$\begin{aligned}\delta\mathbf{r}_i = \delta\mathbf{r}_i^{\text{int}} = \sum_{j\in\mathcal{N}_i}\Bigg[&\frac{1}{2}\frac{\partial\left(\mathbf{f}_{ij}^{i+}\cdot\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{i+}}\delta\mathbf{w}_{ij}^{i+} + \frac{1}{2}\frac{\partial\left(\mathbf{f}_{ij}^{j-}\cdot\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{j-}}\delta\mathbf{w}_{ij}^{j-}\\ &+\frac{\partial\mathbf{d}\left(\mathbf{w}_{ij}^{i+},\mathbf{w}_{ij}^{j-},\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{i+}}\delta\mathbf{w}_{ij}^{i+} + \frac{\partial\mathbf{d}\left(\mathbf{w}_{ij}^{i+},\mathbf{w}_{ij}^{j-},\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{j-}}\delta\mathbf{w}_{ij}^{j-}\Bigg] \qquad i\in\mathcal{V}(\Omega), \quad (7.36a)\end{aligned}$$

$$\delta\mathbf{r}_i = \delta\mathbf{r}_i^{\text{int}} + \frac{\partial\left(\mathbf{f}_i^{\text{BC}}\cdot\boldsymbol{n}_i\right)}{\partial\mathbf{w}_i}\delta\mathbf{w}_i \qquad i\in\mathcal{V}(\partial\Omega). \quad (7.36b)$$

Defining *local Jacobians* associated with edge $\xrightarrow{i\,j}$ and boundary node i,

$$\begin{aligned}\mathbf{J}_{ij}^{i+} &= \frac{1}{2}\frac{\partial\left(\mathbf{f}_{ij}^{i+}\cdot\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{i+}} + \frac{\partial\mathbf{d}\left(\mathbf{w}_{ij}^{i+},\mathbf{w}_{ij}^{j-},\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{i+}},\\ \mathbf{J}_{ij}^{j-} &= \frac{1}{2}\frac{\partial\left(\mathbf{f}_{ij}^{j-}\cdot\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{j-}} + \frac{\partial\mathbf{d}\left(\mathbf{w}_{ij}^{j-},\mathbf{w}_{ij}^{j-},\boldsymbol{n}_{ij}\right)}{\partial\mathbf{w}_{ij}^{j-}},\\ \mathbf{J}_i^{\text{BC}} &= \frac{\partial\left(\mathbf{f}_i^{\text{BC}}\cdot\boldsymbol{n}_i\right)}{\partial\mathbf{w}_i}\end{aligned} \quad (7.37)$$

allows expression (7.36) to be written in the shorter form

$$\delta\mathbf{r}_i = \begin{cases}\displaystyle\sum_{j\in\mathcal{N}_i}\left[\mathbf{J}_{ij}^{i+}\delta\mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-}\delta\mathbf{w}_{ij}^{j-}\right] & i\in\mathcal{V}(\Omega),\\ \displaystyle\sum_{j\in\mathcal{N}_i}\left[\mathbf{J}_{ij}^{i+}\delta\mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-}\delta\mathbf{w}_{ij}^{j-}\right] + \mathbf{J}_i^{\text{BC}}\delta\mathbf{w}_i & i\in\mathcal{V}(\partial\Omega).\end{cases} \quad (7.38)$$

Differentiating the reconstruction (7.24)–(7.25) yields the relation between $\delta\mathbf{w}_{ij}^{i+}$, $\delta\mathbf{w}_{ij}^{j-}$ and corresponding quantities at the nodes

$$\begin{aligned}\delta\mathbf{w}_{ij}^{i+} &= \frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{w}_i}\delta\mathbf{w}_i + \frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{w}_j}\delta\mathbf{w}_j + \frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{p}_{ij}^{i}}\delta\mathbf{p}_{ij}^{i},\\ \delta\mathbf{w}_{ij}^{j-} &= \frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{w}_i}\delta\mathbf{w}_i + \frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{w}_j}\delta\mathbf{w}_j + \frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{p}_{ij}^{j}}\delta\mathbf{p}_{ij}^{j},\end{aligned} \quad (7.39)$$

where $\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i}$, $\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j}$, $\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^{i}}$, $\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i}$, $\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j}$ and $\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^{j}}$ are diagonal matrices containing the partial derivatives of the reconstruction functions in (7.24)–(7.25). Expressions for these diagonal Jacobians are given in Appendix 12.4

Lemma 2. *The following symmetry properties hold.*

$$\mathbf{J}_{ji}^{j+} = -\mathbf{J}_{ij}^{j-} \qquad \mathbf{J}_{ji}^{i-} = -\mathbf{J}_{ij}^{i+} \qquad \delta\mathbf{w}_{j+}^{ji} = \delta\mathbf{w}_{ij}^{j-}, \qquad \delta\mathbf{w}_{i-}^{ji} = \delta\mathbf{w}_{ij}^{i+}.$$

If the reconstruction (7.24)–(7.25) is done on the components of the *primitive* variables $\mathbf{v}$, expression (7.39) should be replaced by

$$\begin{aligned}
\delta\mathbf{w}_{ij}^{i+} &= \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{v}_{i+}} \left[\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i} \frac{\partial \mathbf{v}_i}{\partial \mathbf{w}_i} \delta\mathbf{w}_i + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j} \frac{\partial \mathbf{v}_j}{\partial \mathbf{w}_j} \delta\mathbf{w}_j + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^{i}} \boldsymbol{t}_{ij} \cdot \nabla_h \frac{\partial \mathbf{v}_i}{\partial \mathbf{w}_i} \delta\mathbf{w}_i \right], \\
\delta\mathbf{w}_{ij}^{j-} &= \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{v}_{j-}} \left[\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i} \frac{\partial \mathbf{v}_i}{\partial \mathbf{w}_i} \delta\mathbf{w}_i + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j} \frac{\partial \mathbf{v}_j}{\partial \mathbf{w}_j} \delta\mathbf{w}_j + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^{j}} \boldsymbol{t}_{ij} \cdot \nabla_h \frac{\partial \mathbf{v}_j}{\partial \mathbf{w}_j} \delta\mathbf{w}_j \right].
\end{aligned} \tag{7.40}$$

As a consequence of the symmetries of Lemma 2, the linearized residual in expression (7.38) may be computed as follows.

Algorithm 10. COMPUTATION OF THE LINEARIZED RESIDUAL.

The first edge loop computes the discrete gradient $\delta\mathbf{g}_i = \nabla_h \delta\mu_i \; \forall i \in \mathcal{V}(\overline{\Omega})$ (cf. (7.20)). Each assignment should be repeated for each component $\delta\mu_i$ of $\delta\mathbf{w}_i$.

1. Set $\delta\mathbf{g}_i \leftarrow \mathbf{0} \qquad \forall i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{ij}$:
   (a) $\delta\mathbf{g}_i \leftarrow \delta\mathbf{g}_i + \dfrac{1}{2|V_i|}(\delta\mathbf{w}_i + \delta\mathbf{w}_j) \otimes \boldsymbol{n}_{ij}$
   (b) $\delta\mathbf{g}_j \leftarrow \delta\mathbf{g}_j - \dfrac{1}{2|V_j|}(\delta\mathbf{w}_i + \delta\mathbf{w}_j) \otimes \boldsymbol{n}_{ij}$
3. For each $i \in \mathcal{V}(\partial\Omega)$:
   (a) $\delta\mathbf{g}_i \leftarrow \delta\mathbf{g}_i + \dfrac{1}{|V_i|}\mathbf{w}_i \times \boldsymbol{n}_i$

The second edge loop assembles the linearized residual.

1. Set $\delta\mathbf{r}_i = \mathbf{0} \qquad \forall i \in \mathcal{V}(\overline{\Omega})$
2. For each edge $\xrightarrow{ij}$:
   (a) $\delta\mathbf{r}_{ij} \leftarrow \mathbf{J}_{ij}^{i+}\delta\mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-}\delta\mathbf{w}_{ij}^{j-}$,

where $\mathbf{J}_{ij}^{i+}$, $\mathbf{J}_{ij}^{j-}$, $\delta\mathbf{w}_{ij}^{i+}$, and $\delta\mathbf{w}_{ij}^{j-}$ are computed by expressions (7.37) and (7.39).

(b) $\delta\mathbf{r}_i \leftarrow \delta\mathbf{r}_i + \delta\mathbf{r}_{ij}$,

(c) $\delta\mathbf{r}_j \leftarrow \delta\mathbf{r}_j - \delta\mathbf{r}_{ij}$,

3. For each $i \in \mathcal{V}(\partial\Omega)$:

(a) $\delta\mathbf{r}_i \leftarrow \delta\mathbf{r}_i + \mathbf{J}_i^{\mathrm{BC}}\delta\mathbf{w}_i$

Note that $\delta\mathbf{w}$ and $\delta\mathbf{r}$ can be any node-based vectors and therefore the above algorithm defines the implementation of the forward linear operator $\delta\mathbf{r} = \frac{\partial\mathbf{r}}{\partial\mathbf{w}}\delta\mathbf{w}$. The edge calculations involve the objects $\mathbf{J}_{ij}^{i+}$ and $\mathbf{J}_{ij}^{j-}$ (Jacobians); and $\frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{w}_i}$, $\frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{w}_j}$, $\frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{p}_{ij}^{i}}$, $\frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{w}_i}$, $\frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{w}_j}$, $\frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{p}_{ij}^{j}}$ (diagonal matrices containing derivatives of the reconstruction). For minimum computational work in a Newton–Krylov solver, where the linearized residual needs to be recalculated numerous times, these objects can be precomputed in one loop over the edges and stored in an edge-based data structure. Similarly, the boundary Jacobian $\mathbf{J}_i^{\mathrm{BC}}$ is computed and stored by one loop over the boundary nodes. It is also possible to device corresponding algorithm when the reconstruction is made on the primitive variables. Corresponding formulas will invoke the transformation matrix $\partial\mathbf{v}/\partial\mathbf{w}$ and its inverse.

7.2.3.3 Adjoint state Jacobian-vector product

Here we consider the implementation of the adjoint state Jacobian-vector product

$$\mathbf{r}^* = \frac{\partial\mathbf{r}}{\partial\mathbf{w}}^T\mathbf{z} \tag{7.41}$$

where $\mathbf{z}$, in addition to being an estimate of the adjoint state, may be any node-based vector of the proper dimension. This operator is applied repeatedly in an iterative linear solver, such as GMRES, for the solution of the adjoint system (7.6).

The linearized operator in (7.41) can be assembled using the reverse mode of AD at the edge level. However, since this operator is applied repeatedly inside of an iterative solver, more effort is justified in developing a hybrid symbolic/AD approach and this is the focus of the rest of this section.

This adjoint product is

$$\mathbf{r}_i^* = \begin{cases} \displaystyle\sum_{j\in\mathcal{N}_i}\mathbf{r}_{ij}^* - \nabla_h\cdot\mathbf{g}_i^* & \text{for } i\in\mathcal{V}(\Omega), \\ \displaystyle\sum_{j\in\mathcal{N}_i}\mathbf{r}_{ij}^* - \nabla_h\cdot\mathbf{g}_i^* + \left(\mathbf{J}_i^{\mathrm{BC}}\right)^T\mathbf{z}_i & \text{for } i\in\mathcal{V}(\partial\Omega), \end{cases} \tag{7.42}$$

in which

$$\mathbf{r}_{ij}^* = \left[\frac{\partial\mathbf{w}_{ij}^{i+}}{\partial\mathbf{w}_i}(\mathbf{J}_{ij}^{i+})^T + \frac{\partial\mathbf{w}_{ij}^{j-}}{\partial\mathbf{w}_i}(\mathbf{J}_{ij}^{j-})^T\right](\mathbf{z}_i - \mathbf{z}_j) \tag{7.43}$$

is the contribution from edge $\xrightarrow{ij}$ to the adjoint residual at node i, and

$$\mathbf{g}_i^* = \sum_{j \in \mathcal{N}_i} \left[\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^i} (\mathbf{J}_{ij}^{i+})^T (\mathbf{z}_i - \mathbf{z}_j) \right] \otimes \boldsymbol{t}_{ij} \tag{7.44}$$

is an object of the same dimension as the flux function (7.14). The discrete divergence $\nabla_h \cdot$ in expression (7.42) is defined by

$$\nabla_h \cdot \mathbf{f}_i = \begin{cases} \displaystyle\sum_{j \in \mathcal{N}_i} \frac{1}{2} \left(\frac{1}{|V_j|} \mathbf{f}_j \cdot \boldsymbol{n}_{ij} - \frac{1}{|V_i|} \mathbf{f}_i \cdot \boldsymbol{n}_{ij} \right) & \text{for } i \in \mathcal{V}(\Omega), \\ \displaystyle\sum_{j \in \mathcal{N}_i} \frac{1}{2} \left(\frac{1}{|V_j|} \mathbf{f}_j \cdot \boldsymbol{n}_{ij} - \frac{1}{|V_i|} \mathbf{f}_i \cdot \boldsymbol{n}_{ij} \right) - \frac{1}{|V_i|} \mathbf{f}_i \cdot \boldsymbol{n}_i & \text{for } i \in \mathcal{V}(\partial\Omega), \end{cases} \tag{7.45}$$

where $\mathbf{f}_i$ is any object of the same dimension as the flux function (7.14). This adjoint Jacobian-vector product is needed for the implementation of an iterative solver for the adjoint linear system in (7.6). The edge-based algorithm for assembling the adjoint product is given below.

Algorithm 11. ADJOINT RESIDUAL COMPUTATION.

The first edge loop computes the residual contributions coming from (7.43) *and the* $\mathbf{g}_i^*$*'s of expression* (7.44)*.*

1. Set $\mathbf{g}_i^* \leftarrow \mathbf{0} \qquad \forall i \in \mathcal{V}(\overline{\Omega})$.
2. Set $\mathbf{r}_i^* \leftarrow \mathbf{0} \qquad \forall i \in \mathcal{V}(\overline{\Omega})$.
3. For each edge $\xrightarrow{ij}$:
   (a) $\mathbf{q}_{ij}^i \leftarrow \left(\mathbf{J}_{ij}^{i+}\right)^T (\mathbf{z}_i - \mathbf{z}_j)$
   (b) $\mathbf{q}_{ij}^j \leftarrow \left(\mathbf{J}_{ij}^{j-}\right)^T (\mathbf{z}_i - \mathbf{z}_j)$
   (c) $\mathbf{r}_i^* \leftarrow \mathbf{r}_i^* + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i} \mathbf{q}_{ij}^i + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i} \mathbf{q}_{ij}^j$
   (d) $\mathbf{r}_j^* \leftarrow \mathbf{r}_j^* + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j} \mathbf{q}_{ij}^i + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j} \mathbf{q}_{ij}^j$
   (e) $\mathbf{g}_i^* \leftarrow \mathbf{g}_i^* + \left[\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^i} \mathbf{q}_{ij}^i \right] \otimes \boldsymbol{t}_{ij}$
   (f) $\mathbf{g}_j^* \leftarrow \mathbf{g}_j^* + \left[\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^j} \mathbf{q}_{ij}^j \right] \otimes \boldsymbol{t}_{ij}$

The second edge loop adds the contributions from the discrete divergence and the boundary terms in expression (7.42)*.*

1. For each edge $\xrightarrow{ij}$:
   (a) $\mathbf{r}_i^* \leftarrow \mathbf{r}_i^* - \frac{1}{2} \left(\frac{1}{|V_j|} \mathbf{g}_j^* \cdot \boldsymbol{n}_{ij} - \frac{1}{|V_i|} \mathbf{g}_i^* \cdot \boldsymbol{n}_{ij} \right)$

(b) $\mathbf{r}_j^* \leftarrow \mathbf{r}_j^* - \frac{1}{2}\left(\frac{1}{|V_j|}\mathbf{g}_j^* \cdot \boldsymbol{n}_{ij} - \frac{1}{|V_i|}\mathbf{g}_i^* \cdot \boldsymbol{n}_{ij}\right)$

2. For each $k \in \mathcal{V}(\partial\Omega)$:

(a) $\mathbf{r}_k^* \leftarrow \mathbf{r}_k^* + \left(\mathbf{J}_k^{\mathrm{BC}}\right)^T \mathbf{z}_k - \frac{1}{|V_k|}\mathbf{g}_k^* \cdot \boldsymbol{n}_k$

7.2.4 Adjoint geometric Jacobian-vector product

We now consider the simultaneous computation of the following three adjoint Jacobian-vector products

$$\boldsymbol{q}^x = \frac{\partial \mathbf{r}}{\partial \boldsymbol{x}}^T \mathbf{z}, \tag{7.46}$$

$$\boldsymbol{q}^n = \frac{\partial \mathbf{r}}{\partial \boldsymbol{n}}^T \mathbf{z}, \tag{7.47}$$

$$q^V = \frac{\partial \mathbf{r}}{\partial V}^T \mathbf{z}, \tag{7.48}$$

as required in (7.10), where $\mathbf{z}$, in addition to being the adjoint state vector, may be any node-based vector of the proper dimension.

The quantities $\boldsymbol{q}^x$, $\boldsymbol{q}^n$ and $\boldsymbol{q}^V$ can be computed together in a single edge loop. A simple way to compute these quantities is to use the reverse mode of AD with the edge-level C++ functions that assemble the residual in Algorithm 9.

This adjoint computation is only performed once in the computation of the reduced gradient (7.4). Therefore, the iterative solution of the adjoint system (7.6) dominates and this adjoint computation does not consume a significant percentage of the overall CPU time. As a result, this adjoint computation does not warrent the effort to develop a more invasive hybrid symbolic/AD implementation.

7.2.5 Finite-volume dual-mesh algorithm

The computation of a the finite-volume dual mesh (i.e. control volumes V_i and edge norms $\boldsymbol{n}_{ij}$) from the positions of the nodes in the finite-element mesh (i.e. $\boldsymbol{x}_i$) is an important part of the residual computation. First, the dual-mesh algorithm is described followed by the require adjoint computation.

Denote by $K \in \mathcal{T}_h$ a finite-element mesh object K in the "triangulation" $\mathcal{T}_h$ of the domain Ω. The mesh object K may be a hexahedron, tetrahedron, pyramid, or prism. The boundary ∂K of mesh object K is composed of subsurfaces $\partial_m K$ that are planar (tetrahedrons) or bilinear (hexahedrons). Similarly, the boundary $\partial(\partial_m K)$ of each subsurface $\partial_m K$ is composed of a number of straight edges $\partial_n \partial_m K$ (figure 7.3). The dual mesh algorithm loops each subsurface of each finite element. For each edge in current subsurface, the algorithm computes normal and cell volume contributions as depicted in Figure 7.3.

Denote by $\mathcal{V}(\overline{K})$, $\mathcal{V}(\partial_m K)$, and $\mathcal{V}(\partial_n \partial_m K)$ lists of mesh indices in the corners of K, $\partial_m K$, and $\partial_n \partial K$, respectively. The number of mesh nodes in these lists are denoted $|\mathcal{V}(\overline{K}|)$ and $|\mathcal{V}(\partial_m K)|)$. For a hexahedron, we have $|\mathcal{V}(\overline{K})| = 8$ and $|\mathcal{V}(\partial_m K)| = 4$. Each edge list $\mathcal{V}(\partial_n \partial_m K)$ contains two mesh indices i_n and j_n (or simply i and j if there is no risk of confusion). The formulas in Algorithm 12 produce normals $\boldsymbol{n}_{ij}$ pointing from node i to node j and positive control volumes $|V_i|$, $|V_j|$ if node i and j in the edge list $\mathcal{V}(\partial_n \partial_m K)$ are given a positive orientation with respect to surface $\partial_m K$ (Figure 7.3), that is, traveling from node $\boldsymbol{x}_i$ to node $\boldsymbol{x}_j$ corresponds to a counterclockwise direction along the edge of surface $\partial_m K$ viewed from outside of element K.

Algorithm 12. DUAL MESH ALGORITHM

First loop all elements

1. For each $K \in \mathcal{T}_h$:

   (a) Set $\overline{\boldsymbol{x}}_K \leftarrow \dfrac{1}{|\mathcal{V}(\overline{K})|} \displaystyle\sum_{i \in \mathcal{V}(\overline{K})} \boldsymbol{x}_i$

   Loop all surfaces to current element

   (b) For each $\partial_m K \subset \partial K$:

      i. Set $\overline{\boldsymbol{x}}_{\partial_m K} \leftarrow \dfrac{1}{|\partial_m K|} \displaystyle\sum_{i \in \mathcal{V}(\partial_m K)} \boldsymbol{x}_i$

      Loop all edges of current surface

      ii. For each $\partial_n \partial_m K \subset \partial(\partial_m K)$:

         A. Set $\overline{\boldsymbol{x}}_{\partial_n \partial_m K} \leftarrow \frac{1}{2}(\boldsymbol{x}_{i_n} + \boldsymbol{x}_{j_n})$

         B. Set $\boldsymbol{n}_{ij}^m \leftarrow \frac{1}{2}(\overline{\boldsymbol{x}}_K - \overline{\boldsymbol{x}}_{\partial_n \partial_m K}) \times (\overline{\boldsymbol{x}}_{\partial_m K} - \overline{\boldsymbol{x}}_{\partial_n \partial_m K})$

         C. Set $|V_{ij}^K| \leftarrow \frac{1}{2}\boldsymbol{n}_{ij}^m \cdot (\boldsymbol{x}_{j_n} - \boldsymbol{x}_{i_n})$

         Now update edge normals and cell volumes

         D. $\boldsymbol{n}_{ij} \leftarrow \boldsymbol{n}_{ij} + \boldsymbol{n}_{ij}^m$

         E. $|V_i| \leftarrow |V_i| + |V_{ij}^m|$

         F. $|V_j| \leftarrow |V_j| + |V_{ij}^m|$

Algorithm 12 repeatedly computes the normal contributions $\boldsymbol{n}_{ij}^m$.

7.2.5.1 Adjoint dual-mesh Jacobian-vector product

The composite adjoint operator in (7.10) requires the adjoint of the dual mesh algorithm. The required operation is

$$y = \frac{\partial D_n}{\partial \boldsymbol{x}}^T \boldsymbol{q}^n + \frac{\partial D_V}{\partial \boldsymbol{x}}^T \boldsymbol{q}^V, \tag{7.49}$$

where $\boldsymbol{q}^n$ and $\boldsymbol{q}^V$ are given in (7.47) and (7.48).

The adjoint operation in (7.49) can be performed by appling the reverse mode of AD to the element-level C++ function that performs the dual-mesh assembly in Algorithm 12.

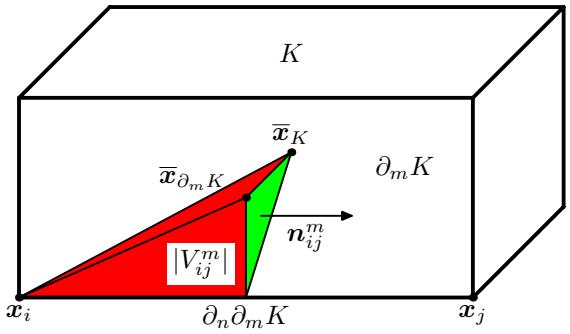


Figure 7.3. The boundary ∂K of a mesh object K is composed of faces $\partial_m K$. There are 6 faces for a hexahedral K. The boundary of each subsurface $\partial_m K$ is composed of straight edges $\partial_n \partial_m K$. For a quadrilateral face there are 4 edges. The dual mesh algorithm computes contributions $\boldsymbol{n}_{ij}^m = \frac{1}{2}(\overline{\boldsymbol{x}}_K - \overline{\boldsymbol{x}}_{\partial_n \partial_m K}) \times (\overline{\boldsymbol{x}}_{\partial_m K} - \overline{\boldsymbol{x}}_{\partial_n \partial_m K})$ and $|V_{ij}^m| = \frac{1}{2}\boldsymbol{n}_{ij}^m \cdot (\boldsymbol{x}_j - \boldsymbol{x}_i)$ to the normal $\boldsymbol{n}_{ij}$ and the cell volumes $|V_i|$, $|V_j|$ that are associated with the surface $\partial_m K$, in the picture illustrated by the front vertical face.

Just as is the case for the geometric residual Jacobian-vector product described in Section 7.2.4, the above dual-mesh adjoint computation is only performed once in the computation of the reduced gradient and therefore does not consume enough CPU time to justify a more invasive hybrid symbolic/AD implementation.

7.3 Summary

Hybrid symbolic/AD[3] methods were used to develop exact forward and adjoint linear operators for the second-order accurate Euler equations. Shape optimization is the ultimate goal and we outline the mathematical formulation to calculate the final reduced gradient of the objective function. This calculation is dependent on a number of derivatives as a result of several dependencies, such as shape parameterization, mesh movement, dual mesh, and the residual calculation. We derive this gradient calculation and break it down into a series of fundamental steps. We use a finite-volume discretization which requires a dual-mesh algorithm to translate the standard finite-element formatted mesh datasets to a finite-volume format. We address the upwinding with a Monotonic Upwinding Centered Scheme for conservation law (MUSCLE) and use a Roe dissipation sheme. The dissipation term is derived and included in the appendix.

[3] automatic differentiation

Chapter 8

The Development of Abstract Numerical Algorithms and Interfacing to Linear Algebra Libraries and Applications.

8.1 Introduction

One area of steady improvement in large-scale engineering and scientific applications is the increased modularity of application design and development. Specification of publicly-defined interfaces, combined with the use of third-party software to satisfy critical technology needs in areas such as mesh generation, data partitioning, and solution methods have been generally positive developments in application design. While the use of third party software introduces dependencies from the application developer's perspective, it also gives the application access to the latest technology in these areas, amortizes library and tool development across multiple applications and, if properly designed, gives the application easy access to more than one option for each critical technology area, e.g., access to multiple linear solver packages.

One category of modules that is becoming increasingly important is abstract numerical algorithms (ANAs). ANAs such as linear and nonlinear equation solvers, methods for stability and bifurcation analysis, transient solvers, uncertainty quantification methods and nonlinear programming solvers for optimization are typically mathematically sophisticated but have surprisingly little essential dependence on the details of what computer system is being used or how matrices and vectors are stored and computed. Thus, by using abstract interface capabilities in languages such as C++, we can implement ANA software such that it will work, unchanged, with a variety of applications and linear algebra libraries. Such an approach is often referred to as *generic programming* [3].

In this chapter we describe a set of basic interfaces for the Trilinos Solver Framework (TSF) called TSFCore as the common interface for (i) ANA development, (ii) the integration of an ANA into an application (APP) and (iii) providing services to the ANA from a linear algebra library (LAL). By agreeing on a simple minimal common interface layer such as TSFCore, we eliminate the many-to-many dependency problem of ANA/APP interfaces. TSFCore is not primarily designed to be the most convenient interface for the direct development of ANAs but it can be used in direct ANA development. Instead, TSFCore is designed to make it easier for developers to provide the basic functionality from APPs and

LALs required for the implementation of ANAs. As an integral subcomponent of optimization, sensitivity analysis stands to benefit from such a standard interface which can significantly reduce the overhead of developing multiple interfaces for each home grown linear algebra system.

It is difficult to describe a set of linear algebra interfaces outside of the context of some class of numerical problems. For this purpose, we will consider numerical algorithms where it is possible to implement all of the required operations exclusively through well defined interfaces to vectors, vector spaces and linear operators. The interfaces described here are the common denominator of all abstract numerical algorithms.

We assume that the reader has a basic understanding of sobject-orientation [86] and C++, and knows how to read basic Unified Modeling Language (UML) [84] class diagrams. We also assume that the reader has some background in large-scale numerics and will therefore be able to appreciate the challenges that are addressed by TSFCore.

To motivate TSFCore, we discuss the context for TSFCore in large-scale (both in lines of code and in problem dimensionality) numerical software. The major requirements for TSFCore are spelled out and this is followed by an overview of the TSFCore linear algebra interfaces.

Bartlett et al. [25, 24] discuss the TSFCore interfaces in more detail.

8.2 Classification of linear algebra interfaces

Although we will discuss APPs, ANAs and LALs in detail later in this section, we want to briefly introduce these terms here to make them clear. While there are certainly other types of modules in a large-scale application, we only focus on these three:

- Application (APP): The modules of an application that are not ANA or LAL modules. Typically this includes the code that is unique to the application itself such as the code that formulates and generates the discrete problem. In general it would also include other third-party software that is not an ANA or LAL module.
- Abstract Numerical Algorithm (ANA): Software that drives a solution process, e.g., an iterative linear or nonlinear solver. This type of package provides solutions to and requires services from the APP, and utilizes services from one or more LALs. It can usually be written so that it does not depend on the details of the computer platform, or the details of how the APP and LALs are implemented, so that an ANA can be used across many APPs and with many LALs.
- Linear Algebra Library (LAL): Software that provides the ability to construct concrete linear algebra objects such as matrices and vectors. A LAL can also be provide a specific linear solver or a preconditioner.

An important focus of this chapter is to clearly identify the interfaces between APPs, ANAs and LALs for the purposes of defining the TSFCore interface.

The requirements for the linear algebra objects as imposed by an ANA are very different from the requirements imposed by an APP code. In order to differentiate the various types of interfaces and the requirements associated with each, consider Figure 8.1. This figure shows the three major categories of

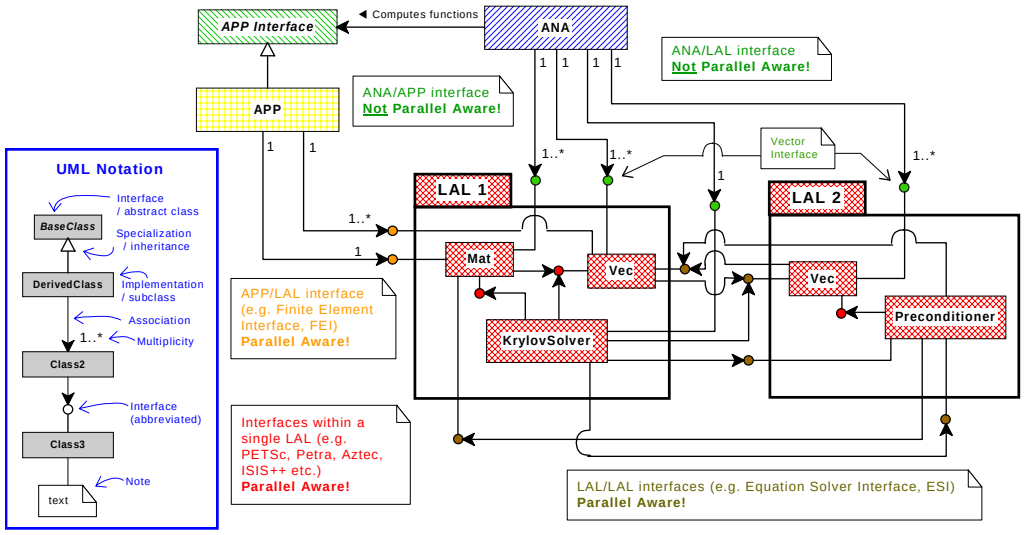


Figure 8.1. UML [42] class diagram : Interfaces between abstract numerical algorithm (ANA), linear algebra library (LAL), and application (APP) software.

software modules that make up a complete numerical application. The first category is application (APP) software in which the underlying data is defined for the problem. This could be something as simple as the right-hand-side and matrix coefficients of a single linear system or as complex as a finite-element method for a 3-D nonlinear PDE-constrained optimization problem. The second category is linear algebra library (LAL) software that implements basic linear algebra operations [71, 9, 39, 204, 16, 164]. These types of software include primarily matrix-vector multiplication, the creation of a preconditioner (e.g. ILU), and may even include several different types of direct linear solvers. The third category is ANA software that drives the main solution process and includes such algorithms as iterative methods for linear and nonlinear systems; explicit and implicit methods for ODEs and DAEs; and nonlinear programming (NLP) solvers [175]. There are many example software packages [16, 204, 164, 47, 31] that contain ANA software.

The types of ANAs described here only require operations like matrix-vector multiplication, linear solves and certain types of vector reduction and transformation operations. All of these operations can be performed with only a very abstract view of vectors, vector spaces and linear operators.

An application code, however, has the responsibility of populating vector and matrix objects and requires the passing of explicit function and gradient value entries, sometimes in a distributed-memory parallel environment; and this is the purpose of a APP/LAL interface. This interface involves a very different set of requirements than those described above for the ANA/APP and ANA/LAL interfaces. An Examples of APP/LAL interfaces can be found in the FEI [60].

Figure 8.1 also shows a set of LAL/LAL interfaces that allows linear algebra objects from one LAL to collaborate with the objects from another LAL. Theses interfaces are very similar to the APP/LAL interfaces and the requirements for this type of interface are also not addressed by TSFCore. The ESI [193] contains examples of LAL/LAL interfaces.

TSFCore, as described in this chapter, specifies only the ANA/LAL interface. TSFCore-based ANA/APP interfaces are described elsewhere (e.g. [24]).

8.3 Vectors in Numerical Software and Challenges in Developing Abstract Interfaces

Vectors provide the primary foundation for the ANA-LAL and ANA-APP interfaces. Beyond transporting vector objects back and forth to the APP and LAL interfaces, ANAs also need to perform various vector reduction (e.g. norms, dot products) and transformation (e.g. vector addition, scaling) operations. In addition, many specialized "nonstandard" vector operations must be performed. Examples of nonstandard operations are presented below to help motivate our design, followed by a discussion of how current and established approaches would attempt to handle these non-standard operations.

8.3.1 Variety of vector operations needed

Perhaps the primary distinction between vectors and other linear algebra objects is the large number of non-standard operations that a complex ANA requires. In addition to the 15 BLAS [126] operations, many other types of operations need to be performed. For example, some of the nonstandard operations an interior-point optimization algorithm (e.g. OOQP [87]) may perform are

$$y_i = \begin{cases} y^{\min} - y_i & \text{if } y_i < y^{\min} \\ y^{\max} - y_i & \text{if } y_i > y^{\max} \\ 0 & \text{if } y^{\min} \le y_i \le y^{\max} \end{cases} \qquad \text{for } i = 1...n, \tag{8.1}$$

$$\alpha \leftarrow \{\max \alpha \ : \ x + \alpha d \ge \beta\}, \tag{8.2}$$

$$\gamma \leftarrow (x + \alpha p)^T (y + \alpha q). \tag{8.3}$$

The interior-point NLP algorithm described in [72] performs several more unusual vector operations, such as

$$d_i \leftarrow \begin{cases} (b-u)_i^{1/2} & \text{if } w_i < 0 \text{ and } b_i < +\infty \\ 1 & \text{if } w_i < 0 \text{ and } b_i = +\infty \\ (u-a)_i^{1/2} & \text{if } w_i \ge 0 \text{ and } a_i > -\infty \\ 1 & \text{if } w_i \ge 0 \text{ and } a_i = -\infty \end{cases} \qquad \text{for } i = 1...n. \tag{8.4}$$

8.3.2 Current approaches to developing interfaces for vectors and vector operations

Currently there are three established approaches to abstracting vectors from ANA software that may be used to address special and unusual vector operations. Below we describe each of these approaches and discuss their limitations.

The first approach (I) is to allow an ANA to access the vector data in some controlled way which enables the ANA to perform required operations. This is by far the most common approach and in the case of parallel numerical codes using SPMD[1], this is currently the preferred method [164, 16, 61]. This approach however assumes that vector data is readily available in every process where the ANA runs. Otherwise, moving large segments of vector data to processes where the ANA is running can cause considerable inefficiencies. For instance, in the case of a client-server architecture, copies of vector data would have to be communicated from the client to the server causing considerable inefficiencies. Approach I potentially provides for an efficient development environment, provided data movement is not an issue and assuming the ANA algorithms do not need to be re-used in other computing environments. Even in an SPMD environment, this approach is not without difficulties (e.g. ghost elements and reduction operations).

More recently, a second approach (II) has been used where each specific ANA defines its own customized abstract LAL interface and then leaves it up to the end user to provide the implementations [150, 87, 49]. ITL [150] uses the C++ template mechanism and requires compile-time polymorphism while OOQP [87] and DiffPack [49] use C++ classes and virtual functions and allow for runtime polymorphism. While this approach is more flexible than approach I and abstracts away the data mapping issues, it simply passes the interfacing problem on to the end user, who is forced to implement the required operations given an existing LAL. For example, OOQP includes more than 30 vector operations, many of which must be implemented from scratch for a new LAL or computing configuration.

Finally, a third approach (III) constructs a general linear algebra interface that tries to anticipate what fundamental or "primitive" operations will be needed. An ANA is expected to implement more specialized operations by stringing together a set of primitives. In theory, this approach allows ANA and LAL software to be developed and maintained independently and be used together with very little extra work. Such an approach was taken by the designers of the Hilbert Class Library (HCL) version 1.0 [95] and was originally motivated by out-of-core data sets but is applicable to any computing environment. Even though approach III is more promising than approaches I and II, it still has three primary shortcomings. First, stringing together a set of primitive operations requires temporary copies and can create an inefficient implementation. Second, this approach requires the standardization of the primitive operations that are part of the interface. Consequently, the developer of an ANA can not add a new operation to an existing LAL interface and expect it to be automatically supported by LAL implementations. Third, nonstandard operations are often difficult to develop through the use of a finite number of primitives.

To demonstrate how a series of primitive vector operations can be used to implement a more specialized operation, consider the vector reduction operation (8.2). This operation could be performed with six temporary vectors $u, v, w, y, z \in \mathbf{R}^n$ and the six primitive vector operations

$$\begin{aligned} &-x_i \to u_i, \quad u_i + \beta \to v_i, \quad v_i/d_i \to w_i, \quad 0 \to y_i, \quad max\left\{w_i, y_i\right\} \to z_i, \\ &min\left\{z_i, i = 1...n\right\} \to \alpha. \end{aligned} \tag{8.5}$$

[1] Single Program Multiple Data (SPMD): A single program running in a distributed-memory environment on multiple parallel processors

Many other vector operations can be performed using primitives. However, it is difficult to see how operations like (8.1) and (8.4) could be implemented with general purpose primitive vector operations. A large number of primitive operations need to be included in a generic vector interface in order to implement most of the required vector operations. For example, the vector interface in HCL 1.0 contains more than 50 operations and still can not accommodate some of the above xsexample vector/array operations.

Another problem is that, in a parallel program, stringing primitives together can result in a serial bottleneck. For instance, in ISIS++ [61], the combined reduction operation

$$\{\alpha, \gamma, \xi, \rho, \varepsilon\} \leftarrow \{(x^T x)^{1/2}, (v^T v)^{1/2}, (w^T w)^{1/2}, w^T v, v^T t\} \tag{8.6}$$

was implemented for the quasi-minimum-residual (QMR) iterative solver.

All five of the reduction operations in (8.6) can be performed in one pass through the vector data (four vector reads) and one global reduction. In contrast, if one must rely on primitive LAL methods for computing each reduction separately, one would need at least seven vector reads and five global reductions. It is certainly possible to add an operation like (8.6) to a LAL, but such an ANA-specialized operation can not be added to a generic LAL interface without the direct support of the developers of all the LALs used with the ANA.

8.4 Vector Reduction/Transformation Operators

8.4.1 Introduction to vector reduction/transformation operations

Our design addresses all the above described limitations associated with approaches I, II, and III. The key design strategy consists of passing user-defined operations to vector objects and having the vector implementations apply the operations to the vector data. ANA developers are therefore not limited to the use of primitives, can freely develop their vector operator implementations, and do not have to depend on temporary copies. In addition to an efficient implementation, this approach is also independent of the underlying data mapping of the vectors. The design allows ANA developers to create any vector reduction/transformation operator (RTOp) that is equivalent to the following element-wise operators:

$$\text{element-wise transformation :} \quad op_T(i, v_i^0 \dots v_i^{p-1}, z_i^0 \dots z_i^{q-1}) \rightarrow z_i^0 \dots z_i^{q-1}, \tag{8.7}$$

$$\text{element-wise reduction :} \quad op_R(i, v_i^0 \dots v_i^{p-1}, z_i^0 \dots z_i^{q-1}) \rightarrow \beta, \tag{8.8}$$

$$\text{reduction of intermediates :} \quad op_{RR}(\hat{\beta}, \tilde{\beta}) \rightarrow \tilde{\beta}, \tag{8.9}$$

where $v^0 \dots v^{p-1} \in \mathbf{R}^n$ are p non-mutable input vectors; $z^0 \dots z^{q-1} \in \mathbf{R}^n$ are q mutable input/output vectors; and β is a reduction target object which may be a simple scalar, a more complex non-scalar (e.g. $\{\alpha, \gamma, \xi, \rho, \epsilon\}$) or NULL. In the most general case, the ANA can define an operator that will simultaneously

perform multiple reduction and transformation operations involving a set of vectors. Simpler operations can be formed by setting $p = 0$, $q = 0$ or $\beta = \text{NULL}$. For example, reduction operations over one vector argument, such as vector norms ($||v||$), are defined with $p = 1$, $q = 0$ and $\beta = \{\text{scalar}\}$. With this design, all of the standard BLAS operations, the example vector operations in (8.1)–(8.6) and many more vector operators can be expressed. The key to optimal performance is that the vector implementation applies (8.7) and (8.8) together on an entire set of sub-vectors (for elements $i = a \dots b$) at once as

$$op(a, b, v^0_{a:b} \dots v^{p-1}_{a:b}, z^0_{a:b} \dots z^{q-1}_{a:b}, \beta) \rightarrow z^0_{a:b} \dots z^{q-1}_{a:b}, \beta. \tag{8.10}$$

In this way, as long as the size of the sub-vectors is sufficiently large, the cost of performing a function call to invoke the operator will be insignificant compared to the cost of performing the computations within the operator. In a parallel distributed vector, $op(\dots)$ is applied to the local sub-vectors on each processor. The only communication between processors is to reduce the intermediate reduction objects $op(\hat{\beta}, \tilde{\beta}) \rightarrow \tilde{\beta}$ (unless $\beta = \text{NULL}$, then no communication is required). It is important to understand that it is the vector implementation that decides how to best segment the vector data into chunks that are passed to the user-defined operator and this results in the most efficient implementation possible.

On most machines, the dominant cost of performing a vector operation is the movement of data to and from main memory [71]. This is especially true for out-of-core vectors. Therefore, performing multiple operations on a vector at the same time such as in (8.6) will be faster than performing them separately in most computing environments.

It is important to note that the type of element-wise operators described in (8.7)–(8.9) can not be used to implement general linear and nonlinear vector operators. For example, a general linear operator A

$$z = op(v) = Av$$

computes the ith element of the output vector z as a linear combination of perhaps all of the right-hand-side vector elements in v. In a distributed-memory environment, this requires careful handling of vector data which results in the use of ghost elements and the targeted communication of potentially large amounts of vector data. For this reason, the element-wise operators defined in (8.7) explicitly state this element-wise requirement. As is clearly seen in (8.7), the ith element in the input/output vectors z can only be computed using information from the ith elements in the vectors v and z and from no other vector elements. It is impossible to design an efficient operator interface that allows non-element-wise transformations that does not also require a detailed knowledge of the computing environment, the layout of vector data and functionality for communicating vector data.

8.4.2 An object-oriented design for reduction and transformation operators

Here an object-oriented [42, 86] design for vector reduction/transformation operators is presented which is based on the "Visitor" pattern [86]. Figure 8.2 shows the general structure of the design. At the core of the design is an interface for vector operators called `RTOp` for which different ANA-specific operator types can be implemented. This operator interface includes methods for the operations (8.9) and (8.10). A vector interface `AbstractVector` includes a method that accepts user-defined `RTOp` operator objects. A vector implementation applies operators in an appropriate manner and returns reduction objects (if

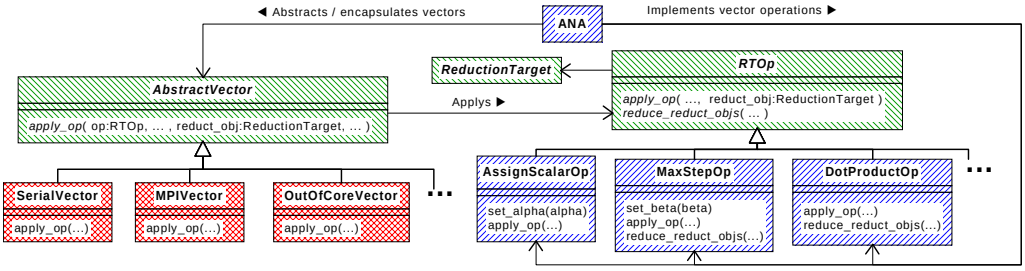


Figure 8.2. UML class diagram : vector reduction/transformation operators

non-NULL). Examples of a few different concrete vector subclasses are shown in Figure 8.2. The reason that this design pattern is called “Visitor” is that an “ObjectStructure” takes a client’s user-defined visitor object and then decides how this visitor object will visit all of the “Elements” containing the data. The key is that the client’s user-defined operations are taken to the data in a transparent way, the data is not presented to the client (as in the “Iterator” design pattern).

The mechanism by which a vector implementation applies a user-defined operator depends on the computing environment. The following three scenarios show how an ANA code can be used with the same operator implementations in three different computing configurations: (i) out-of-core, (ii) SPMD, and (iii) client-server. The ANA code does not need to be recompiled for any scenario and the LAL implementations can be changed at run time.

(i) For out-of-core data sets (Figure 8.3), the vector implementation reads the data from disk one chunk at a time. The operator is called to transform the chunks and/or compute a reduction object. The transformed chunks are then written back to disk and the computed reduction object is returned. The vector implementation applies the operator in the same manner regardless of the operator’s implementation.

(ii) For a SPMD environment (Figure 8.4), the ANA runs in parallel in each process. Once the ANA in each process gives the operator object to the vector object, the vector implementation in each process applies the user-defined operator to only the local elements owned by the process. The intermediate reduction objects in each process are then globally reduced (i.e. using a single call to `MPI_Allreduce(...)`) and the final reduction object is returned. If the operator has a NULL reduction object, then no global reduction is performed and no communication is required.

(iii) In the client-server environment, the ANA runs only in the client process while the APP and LAL run in separate processes on the server. In this scenario, the operator object must be transported from the client to the server, where it is applied to the local vector data in each process. Then the reduction object is returned to the ANA on the client. The client-server configuration demonstrates the fundamental difference of this design from current approaches; the operator is taken to the data, the data is not moved to the operator. The application of an operator in a client-server configuration is involved and the reader is referred to [26] for additional details.

There are several advantages to the RTOp approach. Specifically:

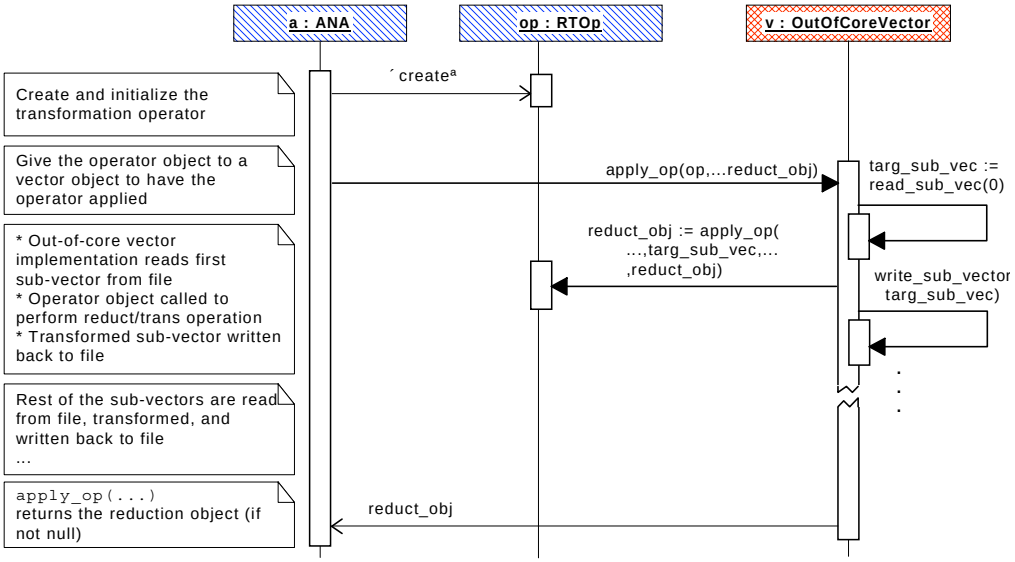


Figure 8.3. UML interaction diagram : Applying a RTOp operator for an out-of-core vector

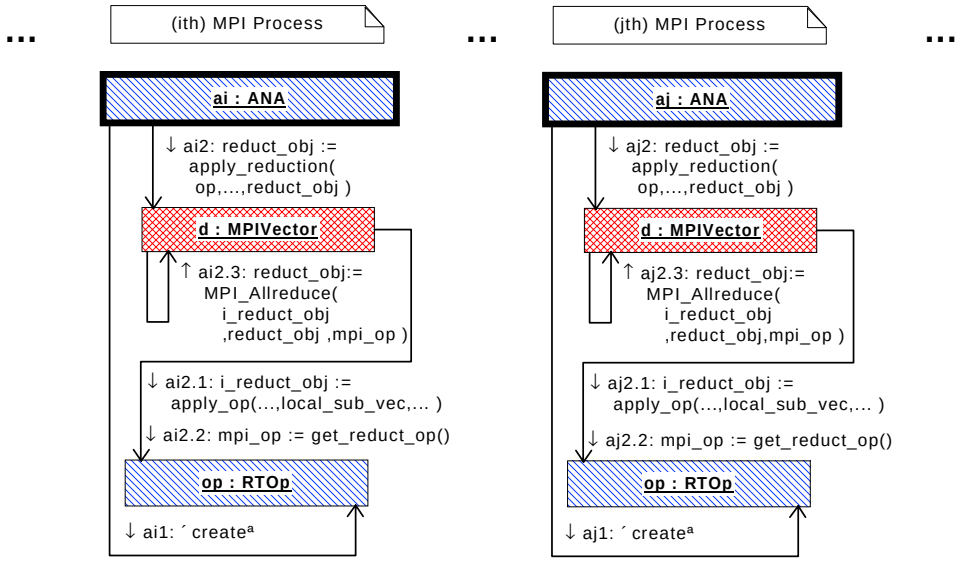


Figure 8.4. UML collaboration diagram : Applying a RTOp operator for a distributed parallel vector

1. LAL developers need only implement one operation — *apply_op(...)* — and not a large collection of primitive vector operations.

2. ANA developers can implement *specialized* vector operations without needing any support from LAL maintainers. Note that *common* vector operators can be shared by the numerical community and need not be implemented from scratch by each set of developers of an ANA code.

3. ANA developers can optimize time consuming vector operations on their own for the platforms they work with.

4. Reduction/transformation operators are more efficient than using primitive operations and temporary vectors (see Section 8.4.3).

5. ANA-appropriate vector interfaces that require built-in standard vector operations (i.e. axpy and norms) can use RTOp operators for the default implementations of these operations. In this way, some ANA developers may not ever need to work with RTOp operators directly in order to apply standard vector operations given a well written vector interface. In other words, the RTOp approach need not inconvenience ANA developers in any way.

8.4.3 Computational results for reduction/transformation operators

Conducting numerical experiments exclusively with low level linear algebra and some communication from reduction operations is somewhat predictable. However, we verify our design with two basic numerical experiments to validate an efficient implementation, and show better performance than any primitive stringing process.

In the first example, a test program based on a mock QMR algorithm is used to investigate the impact of using five primitive operations versus the all-at-once operator in (8.6). A total of 128 processors on CPlant [182] was used in SPMD mode. The ratio of computation to communication was varied by manipulating the number of local vector elements per process (`local_dim`) and the number of axpys per reduction (`num_axpys`). When (`local_dim`)(`num_axpys`) is sufficiently large, computation dominates and there is very little difference between the two implementations. However, when (`local_dim`)(`num_axpys`) is smaller, the all-at-once operator (with a single global reduction verses 5 global reductions) was noticeably more efficient. Figure 8.5 shows the ratio in runtimes for the two approaches. These results indicate that for nontrivial problem sizes the savings in runtime can be significant.

In the second example, the impact of multiple access of the same vector data and the creation of temporaries are investigated. The operation in (8.2) is used for the comparison for which the implementation using primitives is shown in (8.5). Figure 8.6 shows the ratio of CPU times for the all-at-once RTOp operator implementation versus separate primitive vector implementations. The C++ code is written using explicit loops and therefore removes any function call overhead that would otherwise have a dramatic impact for small vectors. There are two variants of the string of primitive vector operations implemented: one that uses preallocated temporary vectors (cached temporaries) and one that uses newly allocated temporary vectors for every evaluation (dynamic temporaries). Vector data is allocated using `std::valarray<double> v(n)` where `n` is the size of the vectors. Note that `std::valarray<>` is not supposed to call constructors on the vector data upon construction so in principle the construction of the vector object could be an $O(1)$ operation (this is not true for `std::vector<>`). This is strictly a serial program so there is no communication overhead to consider. The vector operators are performed several times in a loop and the ratios of runtimes are computed. The test program was compiled using GCC

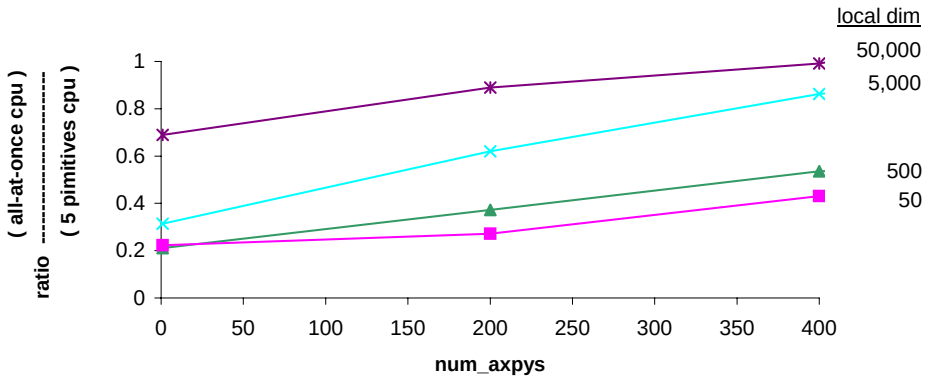


Figure 8.5. Ratio of total process CPU times for using six primitive operations versus the all-at-once operator for the operation in (8.6) (number of processes = 128, number elements per process = 50, 500, 5000, 50000 and `num_axpys = 1 ... 400`)

3.1 under Redhat Linux 7.2 and run on a 1.7 GHz Pentium IV processor. As shown in Figure 8.6, even without the impact of multiple dynamic memory allocations, the implementation using the six primitive vector operations only achieved about 35% of the speed of the all-at-once operator. When naive dynamic allocations where used, the ratio of runtime dropped to about 20%. This example clearly shows the deterioration in runtime performance that the vector primitives approach can have over the all-at-once RTOp approach.

8.5 TSFCore

TSFCore is built on the foundation of vector reduction/transformation operators (RTOp). Here we present the basic requirements for the TSFCore interfaces followed by a brief overview and then provide some examples and expand on some more interesting details.

8.5.1 Basic requirements for TSFCore

Before describing the C++ interfaces for TSFCore, some basic requirements are stated.

1. TSFCore interfaces should be portable to all the ASC [177] platforms where SIERRA [76] and other ASC applications might run. However, a platform where C++ templates are fundamentally broken will not be a supported platform for TSFCore.

2. TSFCore interfaces should provide for stable and accurate numerical computations at a fundamental level.

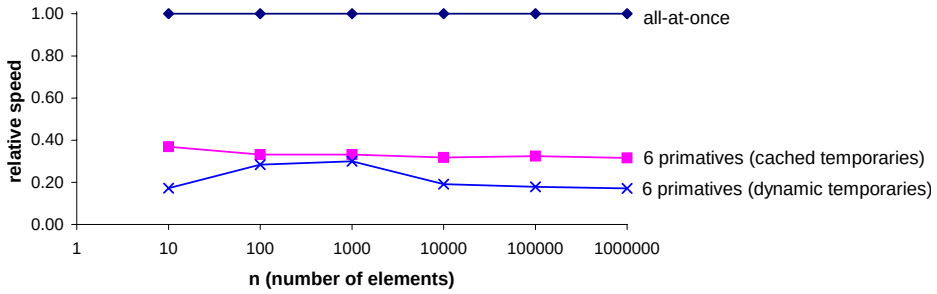


Figure 8.6. Ratio of total process CPU times for using five primitive reductions versus the all-at-once operator for the operation in (8.2) and (8.5). The times for the primitive operation approaches with cached temporary vectors and dynamically allocated temporary vectors are both given.

3. TSFCore should provide a minimal, but complete, interface that addresses all the basic efficiency needs (in both speed and storage) which will result in near-optimal implementations of all of the linear algebra objects and all of the above mentioned ANA algorithms that use these objects.

4. ANAs developed with TSFCore should be able to transparently utilize different types of computing environments such as SPMD, client/server[2] and out-of-core[3] implementations.

5. The work required to implement adapter subclasses (see the "Adapter" pattern in [86]) for and with TSFCore should be minimal and straightforward for all of the existing related linear algebra and ANA interfaces (e.g. the linear algebra interfaces in MOOCHO [23] and NOX [136]). This requirement is facilitated by the fact that the TSFCore interfaces are minimal.

A hand-coded program (e.g. using Fortran 77 and MPI) should not provide any significant gains in performance in any of the above categories in any computing environment. If a hand-coded algorithm in Fortran 77 with MPI should not result in significant improvements in storage requirements, computational speed, or numerical stability. There are many numerical algorithms can can not be considered to be "abstract" and therefore TSFCore and like abstract interfaces should not be used for such algorithms.

8.5.2 Overview of TSFCore

The basic linear algebra abstractions that make up TSFCore are shown in Figure 8.7. The key abstractions include vectors, vector spaces and linear operators. All of the interfaces are templated on the `Scalar` type (the UML notation for templated classes is not used in the figure for the sake of improving readability).

[2]Client/Server: The ANA runs in a process on a client computer and the APP and LAL run in processors on a server
[3]Out-of-core: The data for the problem is stored on disk and is read from and written to back disk as needed

Vector space is the foundation for all other linear algebra abstractions. Vector spaces are abstracted through the *VectorSpace* interface. A *VectorSpace* object acts primarily as an "Abstract Factory" [86] that creates vector objects (which are the "products" in the "Abstract Factory" design pattern).

Vectors are abstracted through the *Vector* interface. The *Vector* interface is very minimal and really only defines one nontrivial function *applyOp(...)*. The *applyOp(...)* function accepts user-defined (i.e. ANA-defined) reduction/transformation operator (RTOp) objects through the templated RTOp C++ interface *RTOpPack::RTOpT*. A set of standard vector operations is provided as nonmember functions using standard RTOp subclasses. The set of operations is also easily extensible. Every *Vector* object provides access to its *VectorSpace* (that was used to create the *Vector* object) through the function `space()` (shown in Figure 8.7 as the role name `space` on the association connecting the *Vector* and *VectorSpace* classes).

The *VectorSpace* interface also provides the ability to create *MultiVector* objects through the *createMembers(numMembers)* function. A *MultiVector* is a tall thin dense matrix where each column in the matrix is a *Vector* object which is accessible through the *col(...)* function. *MultiVector*s are needed for near-optimal processor cache performance (in serial and parallel programs) and to minimize the number of global communications in a distributed parallel environment. The *MultiVector* interface is useful in many different types ANAs as described later. The interface class *Vector* is derived from *MultiVector* so that every *Vector* is a *MultiVector*. This simplifies the development of ANAs in that any ANA that can handle *MultiVector* objects should automatically be able to handle *Vector* objects as well.

VectorSpace also declares a virtual function called *scalarProd(x,y)* which computes the scalar product $< x, y >$ for the vector space. This function has a default implementation based on the dot product $x^T y$. Subclasses can override the *scalarProd(x,y)* function for other, more specialized, application-specific definitions of the scalar product. There is also a *MultiVector* version *VectorSpace::scalarProds(...)* (not shown in the figure). Finally, *VectorSpace* also includes the ability to determine the compatibility of vectors from different vector spaces through the function *isCompatible(vecSpc)*.

Another important type of linear algebra abstraction is a linear operator which is represented by the interface class *LinearOp*. The *LinearOp* interface is used to represent quantities such as a Jacobian matrix. A *LinearOp* object defines a linear mapping from vectors in one vector space (called the `domain`) to vectors in another vector space (called the `range`). Every *LinearOp* object provides access to these vector spaces through the functions `domain()` and `range()` (shown as the role names `domain` and `range` on the associations linking the *OpBase* and *VectorSpace* classes). The exact form of this mapping, as implemented by the function *apply(...)*, is

$$y = \alpha \, op(M) \, x + \beta y \tag{8.11}$$

where M is a *LinearOp* object; x and y are *Vector* objects; and α and β are `Scalar` objects. Note that the linear operator in (8.11) is shown as $op(M)$ where $op(M) = M$, M^T (transpose) or M^* (adjoint) (which is selected by the argument `M_trans`). This implies that the non-transposed, the transposed, and the adjoint linear mappings can be performed. However, support for transposed and adjoint operations by a *LinearOp* object are only optional. If an operation is not supported then the function *opSupported(M_trans)* will return `false`. Note that when $op(M) = M^T$ or M^*, then x and y must lie in the `range` and `domain` spaces respectively which is the opposite for the case where $op(M) = M$.

In addition to implementing linear mappings for single *Vector* objects, the *LinearOp* interface also

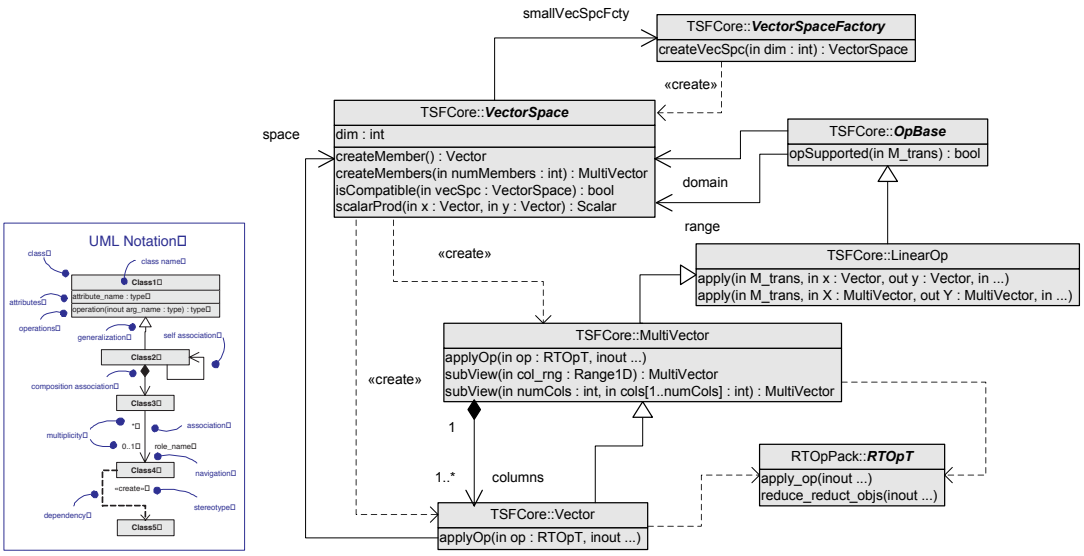


Figure 8.7. UML class diagram : Major components of the TSF interface to linear algebra

provides linear mappings of `MultiVector` objects through an overloaded function `apply(...)` which performs

$$Y = \alpha\, op(M)\, X + \beta Y \tag{8.12}$$

where X and Y are `MultiVector` objects. The `MultiVector` version of the `apply(...)` function has a default implementation based on the `Vector` version. The `Vector` version `apply(...)` is a pure virtual function and therefore must be overridden by subclasses.

The next section goes into some more detail behind the design philosophy for the core interfaces and the use of these interfaces by both clients and subclass developers.

8.5.3 Some TSFCore details and examples

A basic overview of the interface classes shown in Figure 8.7 was provided in Section 8.5.2. In the following sections, we go into more detail about the design of these interfaces and give examples of the use of these classes. Note that in all the below code examples it is assumed that the code is in a source file which include the appropriate header files. Although we could have provided an example for adjoint sensitivities, these interfaces are not very interesting or complex. The majority of the linear algebra interface will already exist in the forward simulation to handle direct or adjoint sensitivities, because the forward solution mechanism are just being reused with different right hand sides. The more difficult interfaces are associated with other objects as part of the optimization algorithm, such as an approximation to the Hessian. In the next section we describe a compact rank-2 update procedure that is easily designed using the TSFCore syntax.

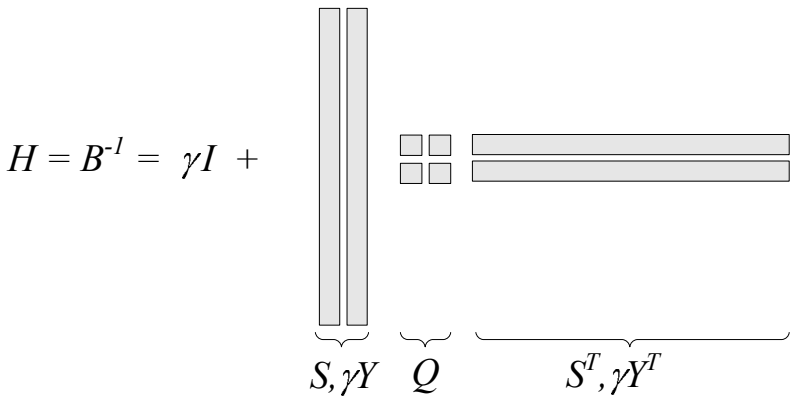


Figure 8.8. A compact limited-memory representation of the inverse of a BFGS matrix.

8.5.3.1 A motivating example sub-ANA : Compact limited-memory BFGS

To motivate the following discussion and to provide examples, we consider the issues involved in using TSFCore to implement an ANA for the compact limited-memory BFGS (LBFGS) method described in [46]. BFGS and other variable-metric quasi-Newton methods are used to approximate a Hessian matrix $B \in \mathbf{R}^{n \times n}$ of second derivatives. This approximation is then used to generate search directions for various types of optimization algorithms. The Hessian matrix B and/or its inverse $H = B^{-1}$ is approximated using only changes in the gradient $y = \nabla f(x_{k+1}) - \nabla f(x_k) \in \mathbf{R}^n$ of some multi-variable scalar function $f(x)$ for changes in the variables $s = x_{k+1} - x_k \in \mathbf{R}^n$. A set of matrix approximations B_k are formed using rank-2 updates where each update takes the form

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}. \tag{8.13}$$

In a limited-memory BFGS method, only a fixed maximum number m_{max} of updates

$$S = \left[\begin{array}{cccc} s_1 & s_2 & \dots & s_m \end{array} \right] \in \mathbf{R}^{n \times m} \tag{8.14}$$

$$Y = \left[\begin{array}{cccc} y_1 & y_2 & \dots & y_m \end{array} \right] \in \mathbf{R}^{n \times m} \tag{8.15}$$

are stored where $m \leq m_{\text{max}}$ is the current number of stored updates; and S and Y are multi-vectors (note that the subscripts in (8.14)–(8.15) correspond to column indexes in the multi-vector objects, not iteration counters k). When an optimization algorithm begins, $m = 0$. Each iteration m is incremented until $m = m_{\text{max}}$, after which the method starts dropping older update pairs (s, y) to make room for newer ones. In a compact LBFGS method, the inverse H (shown in Figure 8.8) of the quasi-Newton matrix B (where when the index k is dropped, it implicitly refers to the current iteration B_k) is approximated using the tall thin multi-vectors S and Y along with a small (serial) coordinating matrix Q (which is computed from S and Y and updated each iteration). The scalar γ is chosen for scaling reasons and $H_0 = B_0^{-1} = \gamma I$ represents the initial matrix approximation from which the updates are performed. A similar compact formula also exists for B which involves the same matrices (and requires solves with Q). In an SPMD

configuration, the multi-vectors S and Y may contain vector elements spread over many processors. However, the number of columns m in S and Y is usually less than 40. Because of the small number of columns in S and Y, all of the linear algebra performed with the matrix Q is performed serially using dense methods (i.e. BLAS and LAPACK). A parallel version of the compact LBFGS method is implemented, for example, as an option in MOOCHO [???]. TSFCore supports efficient versions of all of the operations needed for a near-optimal parallel implementation of this LBFGS method.

The *MultiVector* class provides the critical functionality needed for the efficient implementation of the compact LBFGS sub-ANA and therefore *MultiVector*s are the focus in the next section.

8.5.3.2 *MultiVector*

While the concepts of a *VectorSpace* and *Vector* are well established, the concept of a multi-vector is fairly new. The idea of a multi-vector was motivated by the library Epetra [113] which contains mostly concrete implementations of distributed-memory linear algebra classes using MPI [166]. A key issue is how multi-vectors and vectors relate to each other. In Epetra, the vector class is a specialization of the multi-vector class and each multi-vector object can be seen as a collection of vector objects. This is the approach that TSFCore takes as well.

Note that a multi-vector is not the same thing as a blocked or product vector. In fact, multi-vectors and product vectors are orthogonal concepts and it is possible to have product multi-vectors.

All of the below examples will involve the compact LBFGS implementation described above in Section 8.5.3.1. For these examples we will consider interactions with the two principle *MultiVector* objects `Y_store` and `S_store` which each have $m_{\max}$ columns.

The following example function copies the most recent update vectors `s` and `y` into the multi-vectors `S_store` and `Y_store` and increments the counter `m` for a compact LBFGS implementation.

```
template<class Scalar>
void TSFCore::update_S_Y( const Vector<Scalar>& s, const Vector<Scalar>& y
                          ,MultiVector<Scalar>* S_store, MultiVector<Scalar>* Y_store, int* m )
{
  const int m_max = S_store->domain()->dim(); // Get the maximum number of updates allowed
  if(*m < m_max) {
    ++(*m);                                   // Increment the number of updates
    assign(S_store->col(*m).get(),s);         // Copy in s into S(:,m)
    assign(Y_store->col(*m).get(),y);         // Copy in y into Y(:,m)
  }
  else {
    // We must drop the oldest pair (s,y) and copy in the newest pair
    ...
  }
}
```

The most important issues to discuss with regard to *MultiVector* relate to where it fits in the class hierarchy. The decision adopted for TSFCore was to make *MultiVector* specialize *LinearOp*. In other words, a *MultiVector* object can also act as a *LinearOp* object. Therefore, a *MultiVector* can apply itself to *Vector* or other *MultiVector* objects.

As an example where this is needed, consider using the LBFGS inverse matrix H shown in Figure 8.8 as a

linear operator which acts on multi-vector arguments $U \in \mathbf{R}^{n \times p}$ and $V \in \mathbf{R}^{n \times p}$ in an operation of the form

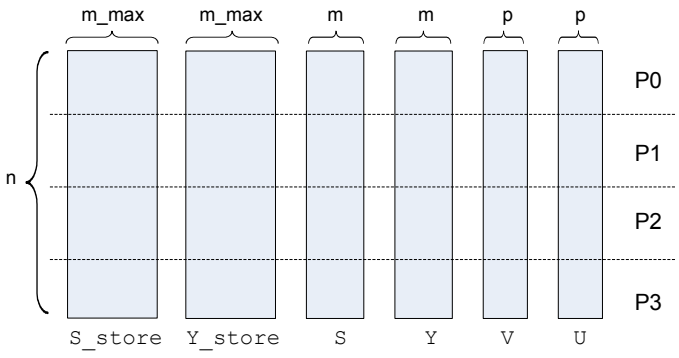
$$\begin{aligned} U &= \alpha B^{-1} V \\ &= \alpha H V \\ &= \alpha g V + \alpha \begin{bmatrix} S & \gamma Y \end{bmatrix} \begin{bmatrix} Q_{ss} & Q_{sy} \\ Q_{sy}^T & Q_{yy} \end{bmatrix} \begin{bmatrix} S^T \\ \gamma Y^T \end{bmatrix} V \end{aligned}$$

where the matrices Q_{ss}, Q_{ys} and Q_{yy} are stored as small `MultiVector` objects. A multi-vector solve using the inverse $H = B^{-1}$ might be used, for instance, in an active-set optimization algorithm where $V \in \mathbf{R}^{n \times p}$ represents the p gradient vectors of the active constraints. This is an important operation, for instance, in the formation of a Schur complement of the KKT system in the QP subproblem of an reduced-space SQP method [22]. This multi-vector operation using H can be performed with the following atomic operations

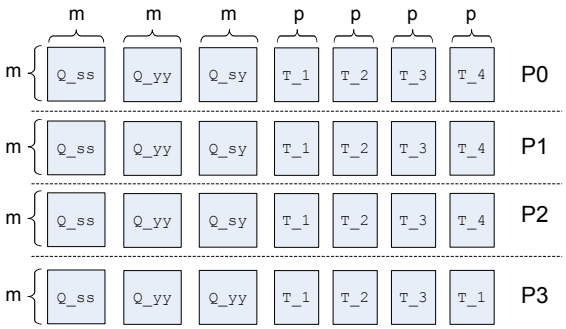
$$\begin{aligned} T_1 &= S^T V \\ T_2 &= Y^T V \\ T_3 &= Q_{ss} T_1 + \gamma Q_{sy} T_2 \\ T_4 &= Q_{sy}^T T_1 + \gamma Q_{yy} T_2 \\ U &= \alpha \gamma V + \alpha S T_3 + \alpha \gamma Y T_4 \end{aligned}$$

where T_1, T_2, T_3 and T_4 are all temporary `MultiVector` objects of dimension $m \times p$. The following function shows how the above operations are performed in order to implement the overall multi-vector operation.

```
template<class Scalar>
void TSFCore::LBFGS_solve(
  int m, Scalar g, const MultiVector<Scalar>& S_store, const MultiVector<Scalar>& Y_store
  ,const MultiVector<Scalar>& Q_ss, const MultiVector<Scalar>& Q_sy, const MultiVector<Scalar>& Q_yy
  ,const MultiVector<Scalar>& V, MultiVector<Scalar>* U, Scalar alpha = 1.0, Scalar = beta = 0.0
  )
{
  // validate input
  ...
  const int p = V.domain()->dim();               // Get number of columns in V and U
  Teuchos::RefCountPtr<const MultiVector<Scalar> >
    S = get_updated(S_store,m),                  // Get view of only stored columns in S_store
    Y = get_updated(Y_store,m);                  // Get view of only stored columns in Y_store
  Teuchos::RefCountPtr<MultiVector<Scalar> >
    T_1 = S->domain()->createMembers(p),         // Create the tempoarary multi-vectors
    T_2 = Y->domain()->createMembers(p),         // ...
    T_3 = S->domain()->createMembers(p),         // ...
    T_4 = Y->domain()->createMembers(p);         // ...
  S->apply(TRANS,V,T_1->get());                  // T_1  =  S'*V
  Y->apply(TRANS,V,T_2->get());                  // T_2  =  Y'*V
  Q_ss.apply(NOTRANS,*T_1,T_3->get());           // T_3  =  Q_ss*T_1
  Q_sy.apply(NOTRANS,*T_2,T_3->get(),gamma,1.0); // T_3 +=  gamma*Q_sy*T_2
  Q_sy.apply(TRANS,  *T_1,T_4->get());           // T_4  =  Q_sy'*T_1
  Q_yy.apply(NOTRANS,*T_2,T_4->get(),gamma,1.0); // T_4 +=  gamma*Q_yy*T_2
  S->apply(NOTRANS,*T_3,U,alpha);                // U    =  alpha*S*T_3
  Y->apply(NOTRANS,*T_4,U,alpha*gamma,1.0);      // U   +=  alpha*gamma*Y*T_4
  axpy(alpha*g,V,U);                             // U   +=  alpha*g*V
}
```



a) Disturbed-memory multi-vectors



b) Locally replicated multi-vectors

Figure 8.9. Carton of disturbed and locally replicated multi-vectors which are used in the example compact LBFGS sub-ANA when run in SPMD mode on four processors. The process boundaries are shown as dotted lines. The numbers for rows and columns of each multi-vector are also shown.

Consider the use of the above function in an SPMD environment where the ANA runs in duplicate and in parallel on each processor. Here, the elements for the multi-vector objects `S_store` (and view `S`) , `Y_store` (and view `Y`), `V` and `U` are distributed across many different processors as Figure 8.9 shows. The case shown[4] in Figure 8.9 is for the situation where $p < m$. In SPMD mode, all of the elements in the multi-vector objects `Q_ss`, `Q_sy`, `Q_yy`, `T_1`, `T_2`, `T_3` and `T_4` are stored locally and in duplicate (i.e. locally replicated) on each processor as shown[5] in Figure 8.9.b. Now let us consider the performance of this set of operations in this context. Note that there are principally three different types of operations with multi-vectors that are performed through the *`MultiVector::apply(...)`* function.

The first type of operation performed by *`MultiVector::apply(...)`* is the parallel/parallel matrix-matrix products performed in the lines

[4]The aspect ratio of the number of rows to number of columns in Figure 8.9 is exaggerated in that in a realistic case the number of rows usually numbers in the tens to hundreds of thousands while the number of columns usually number in only the tens. This was done for illustrative purposes. If the true aspect ratio where shown in Figure 8.9 then all of these multi-vectors would appear to be just vertical lines and would not show a distinction between different multi-vectors.

[5]The same unrealistic aspect ratio shown in Figure 8.9 is the same as shown in Figure 8.9.a, again for illustrative purposes.

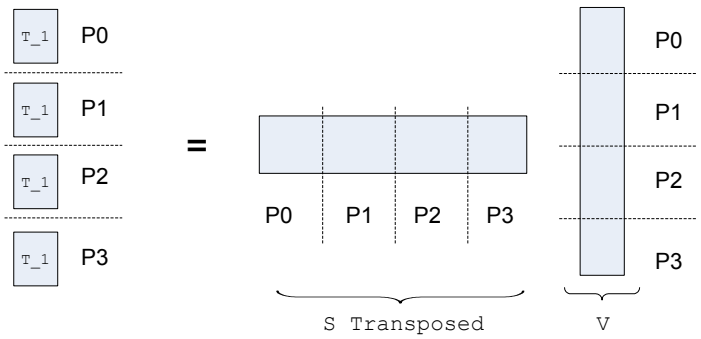


Figure 8.10. Carton of distributed-memory matrix-matrix product (i.e. block dot product) $T_1 = S^T V$ run in SPMD mode on four processors. This operation first performs local matrix-matrix multiplication with the entries of S^T and V on each processor using level-3 BLAS and then a global reduction summation operation is performed (using MPI) to produce T_1 which is returned to all of the processors.

```
S->apply(TRANS,V,T_1->get());
Y->apply(TRANS,V,T_2->get());
```

where the results are stored in the local multi-vectors `T_1` and `T_2`. The operation $T_1 = S^T V$ is shown in Figure 8.10 for the SPMD mode on four processors. This type of operation is also known as a block dot products. These two operations only require a single global reduction for each, independent of the number of updates m represented in S and Y or columns p in V. Note that if there was no concept of a multi-vector and these matrix-matrix products had to be performed one set of vectors at a time, then these two parallel matrix-matrix products would require a staggering $2mp$ global reductions. For $m = 40$ and $p = 20$ this would result in $2mp = 2(40)(20) = 1600$ global reductions! Clearly this many global reductions would destroy the parallel scalability of the overall ANA any many cases. It is in this type of operation that the concept of a *`MultiVector`* is most critical for near-optimal performance in parallel programs. In addition to minimizing communication overhead, the *`MultiVector`* implementation can utilize level-3 BLAS to perform the local processor matrix-matrix multiplications yielding near-optimal cache performance on most systems.

The second type of operation performed by *`MultiVector::apply(...)`* is the local/local matrix-matrix products of small local *`MultiVector`* objects in the lines

```
Q_ss.apply(NOTRANS,*T_1,T_3->get());
Q_sy.apply(NOTRANS,*T_2,T_3->get(),g,1.0);
Q_sy.apply(TRANS,  *T_1,T_4->get());
Q_yy.apply(NOTRANS,*T_2,T_4->get(),g,1.0);
```

where the operation $T_3 = Q_{ss} T_1$ is shown, for example, in Figure 8.11. Note that these types of local computations classify as serial overhead and therefore it is critical that the cost of these operations be kept to a minimum or they could cripple the parallel scalability of the overall ANA. Each of these four

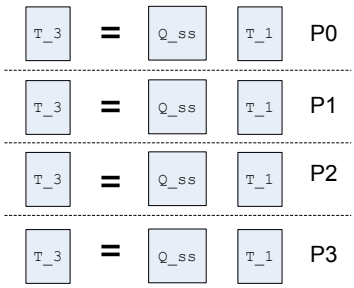


Figure 8.11. Carton of local matrix-matrix product $T_3 = Q_{ss}T_1$ involving locally replicated multi-vectors run in SPMD mode on four processors. In SPMD mode this operation involves no processor-to-processor communication at all.

matrix-matrix multiplications involve only one virtual function call and the matrix-matrix multiplication itself can be performed with level-3 BLAS, achieving the fastest possible flop rate attainable on most processors [71].

The third type of operation performed by *MultiVector::apply(...)* is local/parallel matrix-matrix multiplications performed in the lines

```
S->apply(NOTRANS,*T_3,U,alpha);
Y->apply(NOTRANS,*T_4,U,alpha*g,1.0);
```

where the operation $U = ST_3$ is shown, for example, in Figure 8.12. This type of operation involves fully scalable work with no communication or synchronization required. Here, a vector-by-vector implementation will not be a bottleneck from a standpoint of global communication. However, this operation will utilize level-3 BLAS and yield near-optimal local cache performance where a vector-by-vector implementation would not.

The last type of operation performed in the above **LBFGS_solve(...)** function does not involve *MultiVector::apply(...)* and is shown in the line

```
axpy(alpha*g,V,U);
```

The implementation of this function uses an RTOp transformation operator with the *MultiVector::applyOp(...)* function. Note that this function only involves transformation operations (i.e. no communication) which are fully scalable.

8.6 Conclusions

TSFCore provides the intersection of all of the functionality required by a variety of abstract numerical algorithms ranging from iterative linear solvers all the way up to optimizers. The foundation of TSFCore

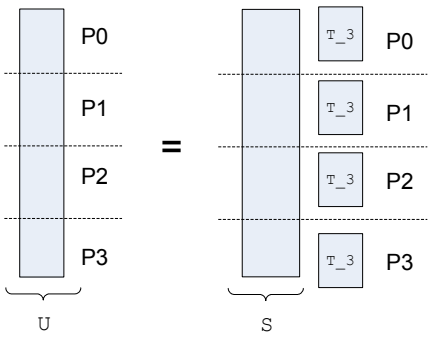


Figure 8.12. Carton of distributed-local matrix-matrix product $U = ST_3$ involving both distributed and locally multi-vectors run in SPMD mode on four processors. To perform this operation, the local elements in the distributed multi-vector S are multiplied with the locally replicated multi-vector T_3 and the result of the matrix-matrix product are set to the local elements of the distributed multi-vector U. In SPMD mode this operation requires no processor-to-processor communication at all.

described here only covers vector spaces, vectors, multi-vectors and linear operators. While this is sufficient for most linear ANA algorithms (i.e. linear equation solvers and eigen solvers) it is not sufficient for higher-level nonlinear algorithms. An extension of the basic TSFCore interfaces for nonlinear problems is described in [24].

By adopting TSFCore as a standard interface layer, interoperability between applications, linear algebra libraries and abstract numerical algorithms in advanced scientific computing environments becomes automatic to a large extent.

Growing complexities associated with computational environments, application domains, and data mapping place difficult demands on the development of numerical algorithms. The vector operator interface proposed here addresses these issues and allows the development of many types of complex abstract numerical algorithms that are highly flexible and reusable.

Advanced object-oriented design patterns were used to develop the RTOp interface and somewhat predictable numerical experiments demonstrate high efficiency in comparison to using a combination of primitives. Also, simple scalability tests confirm minimal serial overhead for large number of processors. Though the numerical efficiencies are noteworthy, development efficiencies and functionality provide the main advantages of this approach.

In summary, there are the five primary advantages to this approach:

1. LAL developers need only implement one operation — `apply_op(...)` — and not a large collection of primitive vector operations.

2. ANA developers can implement *specialized* vector operations without needing any support from LAL maintainers.

3. ANA developers can optimize time consuming vector operations on their own for the platforms they work with.

4. Reduction/transformation operators are more efficient than using primitive operations and temporary vectors (see Section 8.4.3).

5. ANA-appropriate vector interfaces that require built-in standard vector operations (i.e. axpy and norms) can use RTOp operators for the default implementations of these operations.

A large set of vector operators are already available, but more significant is the flexibility to extend functionality. In addition, the extensions are independent of computer architecture and data mapping. By allowing the user to define reduction/transformation operators, all of the vector operations previously mentioned and many more can be implemented efficiently without requiring any temporary vectors.

The success of this approach relies on the adoption of a relatively small interface for reduction/transformation operators that is contained in `RTOp.h`. Future work will include the use of the RTOp approach in a client-server environment, which will specifically address development issues and bottlenecks related to distributed computing, heterogeneous networks and grid computing.

This design for vector reduction and transformation operators has been used to design very powerful ANA-LAL and ANA-APP (for nonlinear programming) interfaces in C++ called `AbstractLinAlgPack` and `NLPInterfacePack` respectively. These interfaces in turn have been used to upgrade a successive quadratic programming optimization package MOOCHO (a.k.a. rSQP++ [26]) to allow fully transparent parallel linear algebra and arbitrary implementations of linear solvers (direct and iterative).

Chapter 9

Instrusive Optimization of a PDE Device Model using Partial Nonlinear Elimination Newton Based Solver

9.1 Introduction

A mathematical formulation for mixed-level simulation algorithms has been developed where different physics interact at potentially different spatial resolutions in a single domain. Some examples can be found in electrical simulation in which PDE-based device modeling provides input into a circuit simulation [123, 124] and device simulation [69] and also in aeroelasticity where Euler equations interact with structural dynamics. Most commonly, solution mechanisms consist of explicit methods where a very uncoupled procedure is used to converge to a final solution. From an implementation standpoint, explicit methods are by far the most efficient. However, these methods are computationally very inefficient in addition to being limiting from an analysis standpoint. If a mixed-level simulation requires sensitivities of all states with respect to certain design parameters, an explicit solution mechanism precludes such an analysis. SAND optimization formulations, however, offer a possible option for not only implicitly formulating the solution of the forward problem, but also for providing the necessary derivative components for analysis capabilities, such as sensitivity analysis and optimization.

We investigate the use of a partial nonlinear elimination solver technique to solve these mixed-level problems. In addition, we show how these formulations are closely coupled to SAND optimization approaches and sensitivity analysis. The next section discusses, in some detail, the mathematical formulation, followed by a section that discusses a potential application in the electrical simulation area. The implementation of these techniques are future work and, consequently, this chapter is for mathematical exposition purposes only.

9.2 Partial Nonlinear Elmination in a Newton-based Nonlinear Equation Solver

Here we consider specialized solution techniques for sets of nonlinear equations where the two sets of equations are treated differently from a globalization perspective. The full set of square nonlinear equations takes the form

$$c(y) = 0, \tag{9.1}$$

where

$y \in \mathbf{R}^{n_y}$ is the vector of state variables, and
$c(y) : \mathbf{R}^{n_y} \to \mathbf{R}^{n_y}$ is the state constraint vector function.

Now let assume that y and $c(y)$ can be partitioned as $y^T = \left[\begin{array}{cc} y_1^T & y_2^T \end{array} \right]$ and

$$c_1(y_1, y_2) = 0, \tag{9.2}$$
$$c_2(y_1, y_2) = 0, \tag{9.3}$$

where

$y_1 \in \mathbf{R}^{n_{y,1}}$,
$y_2 \in \mathbf{R}^{n_{y,2}}$,
$c_1(y_1, y_2) : \mathbf{R}^{n_y} \to \mathbf{R}^{n_{y,1}}$,
$c_2(y_1, y_2) : \mathbf{R}^{n_y} \to \mathbf{R}^{n_{y,2}}$,

and (9.2) defines an implicit function for y_1 as

$$c_1(y_1, y_2) = 0 \quad \Longrightarrow \quad y_1 = \breve{y}_1(y_2). \tag{9.4}$$

The implicit function $y_1 = \breve{y}_1(y_2)$ defined in (9.4) is implemented using a nonlinear equation solver. This type of implementation can be found in a variety of nonlinear solvers such as NOX [136] and MOOCHO [23]. The sensitivity of y_1 with respect to y_2 in (9.4) is given by the direct sensitivity matrix (see [206, Chater 2])

$$\frac{\partial y_1}{\partial y_2} = -\frac{\partial c_1}{\partial y_1}^{-1} \frac{\partial c_1}{\partial y_2} \tag{9.5}$$

which assumes that the Jacobian matrix $\frac{\partial c_1}{\partial y_1}$ is nonsingular at solutions of (9.2) and is therefore a sufficient condition for the Implicit Function Theorem. It then follows that the implicit function in (9.4) exists and is well defined in the neighborhood of an initial guess.

Using (9.4) we obtain the reduced set of equations

$$c_2(\breve{y}_1(y_2), y_2) = 0 \quad \Longrightarrow \quad \breve{c}(\breve{y}), \tag{9.6}$$

where $\breve{y} = y_2$ is introduced for notational convenience.

9.2.1 Two-level Newton methods

Here we consider a straightforward method for the solution of the reduced set of equations in (9.6) using Newton's method which, at iterate $\breve{y}_k$, has linear subproblems of the form

$$\frac{\partial \breve{c}}{\partial \breve{y}} \delta \breve{y} = -\breve{c}(\breve{y}_k) \tag{9.7}$$

with the Jacobian $\frac{\partial \breve{c}}{\partial \breve{y}}$ (evaluated at $\breve{y}_k$) given as

$$\frac{\partial \breve{c}}{\partial \breve{y}} = \frac{\partial c_2}{\partial y_2} + \frac{\partial c_2}{\partial y_2}\frac{\partial y_1}{\partial y_2} = \frac{\partial c_2}{\partial y_2} - \frac{\partial c_2}{\partial y_1}\frac{\partial c_1}{\partial y_1}^{-1}\frac{\partial c_1}{\partial y_2} \tag{9.8}$$

which is a result of (9.5) and (9.6). This straightforward application of Newton's method to the reduced set of equations is known as a two-level Newton method [133, 159, 160]. The drawback of this method is clear from (9.8) due to the presence of the inverse $\frac{\partial c_2}{\partial y_2}^{-1}$; the problems with the strightforward implementations of the two-level Newton method are discussed below. There are two approaches to the solution of the linear system in (9.7). The first is to solve the outer Newton system iteratively which requires matrix-vector products of the form

$$\frac{\partial \breve{c}}{\partial \breve{y}} v = \frac{\partial c_2}{\partial y_2} v - \frac{\partial c_2}{\partial y_1}\frac{\partial c_1}{\partial y_1}^{-1}\left(\frac{\partial c_1}{\partial y_2} v\right) \tag{9.9}$$

which requires a full (exact) linear solve with $\frac{\partial c_1}{\partial y_1}$ for each matrix-vector product. If linear systems with $\frac{\partial c_1}{\partial y_1}$ are also solved using an iterative solver, then this approach will be prohibitively expensive. For example, if each linear solve with $\frac{\partial c_1}{\partial y_1}$ requires 100 inner iterations on average and the outer linear solver for $\frac{\partial \breve{c}}{\partial \breve{y}}$ requires 100 iterations on average, then a total of $100 \times 100 = 10,000$ inner linear solver iterations for solves with $\frac{\partial c_1}{\partial y_1}$ would be required to solve for just a single Newton step $\delta \breve{y}$. Also, due to the form of $\frac{\partial \breve{c}}{\partial \breve{y}}$, it is difficult to form a preconditioner for this system which may otherwise be able to reduce the number of outer linear solver iterations.

The second strightforward approach to solve (9.7) is to explicitly form the matrix $\frac{\partial y_1}{\partial y_2} = -\frac{\partial c_1}{\partial y_1}^{-1}\frac{\partial c_1}{\partial y_2}$ up front which requires, in the worst case, $n_{y,2}$ full linear solves with $\frac{\partial c_1}{\partial y_1}$. In problems with special structure (i.e. the PDE-device circuit problem), the number of required linear solves can be much less. If an iterative solver is used for $\frac{\partial c_1}{\partial y_1}$, each of these $n_{y,2}$ linear solves may require tens or hundreds of linear solver iterations. Therefore, the formation of the direct sensitivity matrix $\frac{\partial y_1}{\partial y_2}$ may require hundreds or thousands of inner linear solve iterations. Once the matrix $\frac{\partial y_1}{\partial y_2}$ is formed, the outer Newton system (9.7) can be solved using a direct or iterative method.

Once a Newton step $\delta \breve{y}$ has been computed from (9.7), some globalization needs to be used such as a standard line-search or trust-region globalization method. For example, consider a Armijo-based line search method [174] which requires that

$$\phi(\breve{y}_k + \alpha \delta \breve{y}) \leq \phi(\breve{y}_k) + \alpha \eta \left.\frac{d(\phi(\breve{y}_k + \alpha \delta \breve{y}))}{d(\alpha)}\right|_{\alpha=0} \tag{9.10}$$

where η is a small constant less than one (usually $\eta = 1 \times 10^{-4}$), α is the step length along the Newton direction and $\phi(\breve{y})$ is some norm on $\breve{c}(\breve{y})$. One particual choice for $\phi(\breve{y})$ is

$$\phi(\breve{y}) = \tfrac{1}{2}||\breve{c}(\breve{y})||_2^2. \tag{9.11}$$

If the reduced Newton system in (9.7) is solved exactly, then it is easy to show that

$$\left.\frac{d(\phi(\breve{y}_k + \alpha\delta\breve{y}))}{d(\alpha)}\right|_{\alpha=0} = -||\breve{c}(\breve{y})||_2^2. \tag{9.12}$$

Note that the evaluation of $\breve{c}(\breve{y}) = c_2(\breve{y}_1(y_2), y_2)$ in (9.11) requires a complete nonlinear solve with $c_1(y_1, y_2) = 0$ in (9.4). In a standard backtracking linear search procedure, an initial step length of $\alpha = 1$ is usually tried first since this is the optimal step length near the solution. Then α is reduced until a condition such as in (9.10) is statisfied and the step is accepted. However, this stategy generally does not work well when using a two-level Newton method because $\breve{c}(\breve{y})$ is not just a simple residual evaluation but involves an inner nonlinear solve to evaluate the implicit function $y_1 = \breve{y}_1(y_2)$ which may not converge if α is too large and a good starting guess for y_1 is not available. Therefore, in these cases, it is usually better to start out with $\alpha = \alpha_{\text{small}}$ and then increase α a little at a time which insures that the inner solve will converge since good starting guesses are available.

The main advantage of the above two-level Newton method is that it may be far more robust than a straightforward full Newton method applied to (9.1) which uses standard globalization techniques.

9.2.2 Full Newton step computation with two-level Newton globalization

In this section we show how a full Newton method for the solution of (9.1) can be modified in order to apply the same type of globalization used in the two-level Newton method described in Section 9.2.1. The resulting method is mathematically equivalent to the strightforward two-level Newton method but can be significantly less expensive in many cases.

Consider a full Newton method for the solution of (9.1) which at the current iterate $y_k = (y_{1,k}, y_{2,k})$ has the linear Newton system

$$\underbrace{\begin{bmatrix} \frac{\partial c_1}{\partial y_1} & \frac{\partial c_1}{\partial y_2} \\ \frac{\partial c_2}{\partial y_1} & \frac{\partial c_2}{\partial y_2} \end{bmatrix}}_{\frac{\partial c}{\partial y}} \underbrace{\begin{bmatrix} \delta y_1 \\ \delta y_2 \end{bmatrix}}_{\delta y} = -\underbrace{\begin{bmatrix} c_1(y_{1,k}, y_{2,k}) \\ c_2(y_{1,k}, y_{2,k}) \end{bmatrix}}_{c(y_k)} \tag{9.13}$$

If at the current iterate $c_1(y_{1,k}, y_{2,k}) = 0$, then the following theorem says that the solution δy_2 to the full Newton system (9.13) is exactly the same as the solution $\delta\breve{y}$ to the reduced Newton system (9.7).

Theorem 1. Equavalence of full and reduced Newton steps

If the Jacobian matrices $\frac{\partial c_1}{\partial y_1}$ in (9.8) and $\frac{\partial c}{\partial y}$ in (9.13) are nonsingular and $c_1(y_{1,k}, y_{2,k}) = 0$, then the solution δy_2 from (9.13) is equal to the solution $\delta\breve{y}$ from (9.8).

Proof

From (9.7) and (9.8) we have

$$\frac{\partial\breve{c}}{\partial\breve{y}}\delta\breve{y} + \breve{c} = \frac{\partial c_2}{\partial y_2}\delta\breve{y} - \frac{\partial c_2}{\partial y_1}\frac{\partial c_1}{\partial y_1}^{-1}\frac{\partial c_1}{\partial y_2}\delta\breve{y} + \breve{c} = 0 \tag{9.14}$$

where $\breve{c} = \breve{c}(\breve{y}_k)$. By substituting $\frac{\partial c_1}{\partial y_2}\delta\breve{y}$ and $\frac{\partial c_2}{\partial y_2}\delta\breve{y}$ (with $\delta y_2 = \delta\breve{y}$) from (9.13) into the above equation we have

$$\begin{aligned} & \left(-\frac{\partial c_2}{\partial y_1}\delta y_1 - c_2\right) - \frac{\partial c_2}{\partial y_1}\frac{\partial c_1}{\partial y_1}^{-1}\left(-\frac{\partial c_1}{\partial y_1}\delta y_1 - \underbrace{c_1}_{0}\right) + c_2 \\ = \quad & -\frac{\partial c_2}{\partial y_1}\delta y_1 - c_2 + \frac{\partial c_2}{\partial y_1}\delta y_1 + c_2 \\ = \quad & 0 \quad \square \end{aligned}$$

The implication of the above theorem is that one has a choice in how one computes the Newton step for the outer Newton iteration in (9.7) for the two-level Newton method. In many cases, the computational effort to solve the full Newton system in (9.13) will be far less than for the solution of reduced Newton system in (9.7). In other cases, it may be more computationaly effective to solve the reduced Newton system as described above. It is interesting to note that the reduced Jacabian matrix $\frac{\partial\breve{c}}{\partial\breve{y}}$ in (9.7) is actually the Schur complement of $\frac{\partial c_1}{\partial y_1}$ in $\frac{\partial c}{\partial y}$. This means that a linear solve with $\frac{\partial\breve{c}}{\partial\breve{y}}$ followed by a linear solve with $\frac{\partial c_1}{\partial y_1}$ can be used to solve any linear system with $\frac{\partial c}{\partial y}$. For example, general linear systems of the form

$$\begin{bmatrix} \frac{\partial c_1}{\partial y_1} & \frac{\partial c_1}{\partial y_2} \\ \frac{\partial c_2}{\partial y_1} & \frac{\partial c_2}{\partial y_2} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{9.15}$$

can be solved as

$$x_2 = \frac{\partial\breve{c}}{\partial\breve{y}}^{-1}\left(b_2 - \frac{\partial c_2}{\partial y_1}\frac{\partial c_1}{\partial y_1}^{-1}b_1\right) \tag{9.16}$$

$$x_1 = \frac{\partial c_1}{\partial y_1}^{-1}\left(b_1 - \frac{\partial c_1}{\partial y_2}x_2\right). \tag{9.17}$$

It is easy to verify the above Schur complement equations. Note that if transposed solves are supported for $\frac{\partial\breve{c}}{\partial\breve{y}}$ and $\frac{\partial c_1}{\partial y_1}$ then this Schur complement method can also be used to solve for transposed systems with $\frac{\partial c}{\partial y}$ which is very useful for SAND optimization algorithms [206]. For example, general linear systems involving the transpose of the form

$$\begin{bmatrix} \frac{\partial c_1}{\partial y_1}^T & \frac{\partial c_2}{\partial y_1}^T \\ \frac{\partial c_1}{\partial y_2}^T & \frac{\partial c_2}{\partial y_2}^T \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{9.18}$$

can be solved as

$$x_2 = \frac{\partial\breve{c}}{\partial\breve{y}}^{-T}\left(b_2 - \frac{\partial c_1}{\partial y_2}^T\frac{\partial c_1}{\partial y_1}^{-T}b_1\right) \tag{9.19}$$

$$x_1 = \frac{\partial c_1}{\partial y_1}^{-T}\left(b_1 - \frac{\partial c_2}{\partial y_1}^T x_2\right). \tag{9.20}$$

The identification of $\frac{\partial\breve{c}}{\partial\breve{y}}$ as a Schur complement of $\frac{\partial c}{\partial y}$ means that the linear algebra computations used in the two-level Newton method described in Section 9.2.1 can be used to implement other types of algorithms including direct and perhaps adjoint (if transposed solves are supported) SAND optimization methods.

Once the full Newton system in (9.13) is solved, the exact same type of line-search or trust-region globalization that is applied in the two-level Newton method can be applied here also. Note that step δy computed from (9.13) need not be discarded but instead should be used as part of an initial guess for the solution of the inner implicit solve $c_1(y_1, y_2) = 0$ with the initial guess $y_{1,k+1} = y_{1,k} + \alpha\delta y_1$.

9.3 Electrical Simulation Example

Analog circuit simulation has been used extensively in the electronic design community and is based upon describing an electrical circuit as a system of network-coupled differential algebraic equations (DAE's). Device simulation is a higher fidelity simulation, which describes a single semiconductor device with a set of coupled partial differential equations (PDE's), discretized on a spatial mesh. These different types of simulations generally present different solver (time integration, nonlinear and linear) requirements, so coupling them together is potentially problematic. A two-level Newton algorithm address these issues by isolating the circuit and PDE device phases of the calculation. A standard single-level Newton method is also utilized and tested against the two-level Newton method.

Coupling algorithms between PDE device simulation and analog circuit simulation have been studied over the past 20 years [75, 165, 184, 161, 214]. Most of these studies employed a full-Newton approach, which is still the most commonly applied method for commercial simulators.

In the early 1990's, a new approach to mixed-level simulation was published by Mayaram [159, 160]. Mayaram proposed the two-level Newton algorithm, and successfully applied it to mixed-level simulation. Later, Rotella [185] also used the two-level approach, building on the work of Mayaram. Unlike earlier work, which was applied to just DC operating point (steady state) and transient calculations, both Mayaram and Rotella also applied their approach to small-signal AC analysis. Both Mayaram and Rotella reported that the two-level Newton algorithm was much more robust than the full-Newton for mixed-level simulation, although it tended to be slower.

9.4 Problem Definition

This section defines the general circuit problem, as well as the general device problem, in terms of the set of equations to be solved. First is a brief description of the circuit problem, then a detailed description of the device problem. Finally coupling equations for mixed-level simulation are presented.

Methods of analog circuit simulation (e.g., SPICE [178]) are well described by Vlach and Singal [212], among others [11, 59, 220]. The circuit to be simulated is represented as a system of coupled DAE's, which are obtained from Kirchoff's current and voltage laws (KCL and KVL, respectively). In general, there are several mathematical formulations that may be used but in practice, nearly all circuit simulators use the "modified KCL" formulation [188]. This is the formulation used by Xyce, and has been described in detail in the Xyce math document [134].

9.4.1 Device Simulation

Methods of device simulation are described by many references including Kramer [139] and Selberherr [195]. As with circuit simulation, there are several different approaches that may be used. However, the most common one, and the one that is used for this work, is the drift-diffusion (DD) formulation. This formulation consists of three coupled PDE's: a single Poisson equation for electrostatic potential and two continuity equations; one each for electrons and holes.

9.4.1.1 Poisson equation

The electrostatic potential ϕ satisfies Poisson's equation

$$-\nabla \cdot (\epsilon \nabla \phi(x)) = \rho(x), \tag{9.21}$$

where ρ is the charge density and ϵ is the permittivity of the material. For semiconductor devices, the charge density is determined by the local carrier densities and the local doping,

$$\rho(x) = q(p(x) - n(x) + C(x)) \tag{9.22}$$

Here $p(x)$ is the spatially-dependent concentration of holes, $n(x)$ the concentration of electrons, and q the magnitude of the charge on an electron. $C(x)$ is the total doping concentration, which can also be represented as $C(x) = N_D^+(x) - N_A^-(x)$, where N_D^+ the concentration of positively ionized donors, N_A^- the concentration of negatively ionized acceptors.

9.4.1.2 Species continuity equations

The continuity equations relate the convective derivative of the species concentrations to the creation and destruction of particles ("recombination/generation").

$$\frac{\partial n(x)}{\partial t} + \nabla \cdot \Gamma_n = -R(x) \tag{9.23}$$

$$\frac{\partial p(x)}{\partial t} + \nabla \cdot \Gamma_p = -R(x) \tag{9.24}$$

Here n is the electron concentration and p is the hole concentration. R is the recombination rate for both species. Γ_n and Γ_p are particle fluxes for electrons and holes, respectively. The sign of R is chosen because R is usually expressed as a recombination rate, and is positive if particles are annihilating. The right hand sides are equal since creation and destruction of carriers occurs in pairs.

One way in which the drift-diffusion model differs from other common formulations is the manner in which the quantities Γ_n and Γ_p are determined. The expressions used are

$$\Gamma_n = n(x)\mu_n E(x) + D_n \nabla n(x), \tag{9.25}$$

$$\Gamma_p = p(x)\mu_p E(x) + D_p \nabla p(x). \tag{9.26}$$

Here μ_n, μ_p are mobilities for electrons and holes, and D_n, D_p are diffusion constants. $E(x)$ is the electric field, which is given by the gradient of the potential, or $-\partial\phi/\partial x$.

9.4.1.3 Initial Condition

Equilibrium Approximation
Even when the SG method is employed, PDE semiconductor simulations will often fail to converge if not given a good initial guess. A simple initial guess, based on the equilibrium electron and hole concentrations can be calculated using information about the doping profile.

$$\text{if } C(x) >= 0 \quad n(x) = \frac{1}{2}\left(\sqrt{C(x)^2 + 4Ni^2} + |C(x)|\right) \tag{9.27}$$

$$\text{if } C(x) < 0 \quad n(x) = \frac{2Ni^2}{\left(\sqrt{C(x)^2 + 4Ni^2} + |C(x)|\right)} \tag{9.28}$$

$$\tag{9.29}$$

$$\text{if } C(x) <= 0 \quad p(x) = \frac{1}{2}\left(\sqrt{C(x)^2 + 4Ni^2} + |C(x)|\right) \tag{9.30}$$

$$\text{if } C(x) > 0 \quad p(x) = \frac{2Ni^2}{\left(\sqrt{C(x)^2 + 4Ni^2} + |C(x)|\right)} \tag{9.31}$$

$$\tag{9.32}$$

where $C(x)$ is the spatially dependent doping concentration, and Ni is the intrinsic carrier concentration. Once the electron and hole concentrations have been calculated, the equilibrium potential can be approximated by

$$\phi(x) = Vt \ln\left(N_D/p(x)\right). \tag{9.33}$$

Vt is the thermal voltage, or $k_b T/q$. N_D in this case is a scalar quantity, representing the maximum donor concentration across the spatial domain.

Nonlinear Poisson
For harder problems, a better initial guess than can be obtained from the above expressions is often necessary. Semiconductor PDE problems tend to be more difficult to solve for high doping levels and also for complex geometries. One can approximate the solution to the drift diffusion semiconductor problem by solving a nonlinear Poisson equation, given by

$$-\nabla \cdot (\epsilon \nabla \phi(x)) = q\left(N_A e^{((\phi_{min} - \phi(x))/Vt)} - N_D e^{((\phi(x) - \phi_{max})/Vt)} + C(x)\right). \tag{9.34}$$

This equation is the same as (9.21), except that now the electron and hole densities are functions of the electrostatic potential, so the equation is nonlinear. The variables N_A and N_D are the maximum acceptor and donor concentrations. In equilibrium, they will approximate the maximum hole and electron concentrations. This equation can easily be solved on the same mesh as the full drift-diffusion formulation using a damped Newton method. Upon solution, the electron and hole densities are obtained by simply evaluating the density terms in (9.34)

$$p(x) = N_A e^{((\phi_{min} - \phi(x))/Vt)} \tag{9.35}$$

$$n(x) = N_D e^{((\phi(x)-\phi_{max})/Vt)} \tag{9.36}$$

For the case of no voltage being applied across the device, solving the nonlinear Poisson gives a result that is very close to the drift-diffusion result, but is much easier to obtain. While (9.34) is nonlinear, it tends to be very well behaved from a nonlinear solver point of view. Most of the difference between the nonlinear Poisson solution and the DD solution for the unbiased case comes from the fact that (9.34) does not account for generation-recombination effects.

In running the simulation, the result of the initial condition calculation is applied before performing the initial steady-state calculation (often referred to as the DC operating point calculation). After the main calculation is underway (transient, DC sweep, etc.), the results of the nonlinear Poisson solution are no longer a valid initial guess, because (9.34) assumes equilibrium conditions.

9.4.1.4 Boundary Conditions

Boundary conditions at the electrodes for all three variables are Dirichlet, while boundary conditions everywhere else are Neumann. At non-electrode boundaries, no current should escape, so the Neumann condition is for zero flux. In the box integration scheme, this boundary condition is automatically imposed by default. The values imposed on electron and hole densities at the electrode boundaries are determined from the doping levels at those boundaries. In this work, we have applied the expressions

$$n_{bc} = \frac{1}{2}\left(\sqrt{C_{bc}^2 + 4Ni^2} + C_{bc}\right), \tag{9.37}$$

$$p_{bc} = \frac{1}{2}\left(\sqrt{C_{bc}^2 + 4Ni^2} - C_{bc}\right), \tag{9.38}$$

where n_{bc} is the imposed value of electron density at a boundary, while p_{bc} is the imposed value for hole density. Also, C_{bc} is the doping level at the boundary. These expressions were taken from Selberherr [195], and enforce both thermal equilibrium and a vanishing space charge at ohmic contacts.

The boundary conditions on the electrostatic potential are external inputs to the problem, which may come from the user or from an external circuit simulator. Their application, however, is not as straightforward as may first appear. Semiconductor devices will establish their own internal potential variation due to charge separation caused by the doping. This can be observed by considering (9.33). Doping levels will usually be different at each electrode boundary, so (9.33) will evaluate to a different value for each one.

Therefore, when an external potential is applied to a PDE device electrode, the boundary condition applied at that electrode has to include an offset due to the built in potential. This offset is determined by evaluating (9.33) at each boundary. The potential boundary condition which is applied is then:

$$\phi_{bc} = \phi_{ckt} + \phi_{offset} \tag{9.39}$$

It should be noted that for some PDE devices, the doping will not be constant along an electrode boundary. For such cases, the imposed boundary conditions on all three variables need to vary accordingly along the electrode.

9.4.2 Coupling Equations

A mixed-level problem will include the full set of circuit equations as well as the full set of device equations. In addition, such a simulation will also require extra equations associated with the coupling to complete the system. In order to couple a PDE-level device simulation to a circuit simulation, it is necessary for the PDE device simulator to fulfill the role of a SPICE-style analog device. While the Xyce device-package interface makes this relatively easy from a programming standpoint, in practice, the PDE-device must provide currents and conductances as a function of voltage.

The interface between a PDE device simulation and a circuit simulation occurs at the device electrodes, each of which are physically connected to a node of the circuit. The current flowing between the device and the circuit must satisfy the equation

$$\sum_{i=1}^{N} A_i \cdot (J_i^n + J_i^p) = i_c, \tag{9.40}$$

where N is the number of mesh nodes along the electrode, A_i is the surface area associated with mesh node i. J_i^n and J_i^p are the electron and hole current densities, respectively. The term on the right hand side of (9.40), i_c, is the circuit current which flows into circuit node 1.

Potentially, a large number of nodes on the PDE mesh lie on an electrode boundary, so the summation necessary to obtain the total current may involve a relatively large number of terms. For a three dimensional simulation, N could potentially be in the thousands.

Equation (9.40) is essentially a KCL equation, so while it involves terms from the PDE device, it is natural to think of it as belonging to the circuit part of the problem. The coupling between the device and the circuit is also enforced on the PDE side of the problem through boundary conditions on the electrode potentials. Semiconductor devices, even when unbiased, possess a certain amount of internal potential variation, due to variations in the doping. A semiconductor device which is not connected to a circuit, and has no external potentials applied to it, will still have this internal potential variation. As a result, any potentials applied to a PDE device that have been obtained from a connected circuit must be adjusted, to account for the internal variation. This results in the equation

$$\phi_{bc} = \phi_{circuit} + \phi_{offset}. \tag{9.41}$$

Here ϕ_{bc} is the voltage boundary condition to be applied to the PDE device on the electrode, $\phi_{circuit}$ is the voltage at the circuit node which is connected to the electrode, and ϕ_{offset} is different for each electrode and depends upon the local doping. It is common for the variables of the PDE simulation to be scaled differently than the variables of the circuit equation. Thus, both (9.40) and (9.41) need to take this into account.

9.5 Solution Methods

This section focuses on solver methods employed by circuit, device and mixed-level simulation. Circuit and device simulation have a number of similarities, without which a two-level Newton approach would not be applicable. First, the similarities between circuit and device simulation from a solver point of view are discussed, followed by a discussion of some of their differences. A discussion of coupling algorithms is the last part of this section.

9.5.1 Similarities

Circuit simulation and PDE device simulation are similar in several regards. Both deal with the same basic physical quantities - electrostatic potential and current. Both problems are nonlinear and implicit in time.

The solver structure consists of a nested set of solver loops. The outer loop is a control loop, which may be a time integration loop or a DC sweep. The middle level is a nonlinear solver, and the inner-most entity is the linear solver.

The system of equations at each time step is represented as

$$\mathbf{f}(\mathbf{w}) = 0 \tag{9.42}$$

in which $\mathbf{w}$ is the solution vector and $\mathbf{f}$ is the system of equations. Equation (9.42) is solved using a Newton method, in which the solution $\mathbf{w}$ is obtained from repeated solves of the equations

$$\mathbf{J}\Delta\mathbf{w} = -\mathbf{f}, \tag{9.43}$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta\mathbf{w}, \tag{9.44}$$

where k is the Newton iteration index. This set of equations is repeatedly evaluated until (9.42) is satisfied.

9.5.2 Differences

Circuit simulation and device simulation are different enough to have somewhat different solver needs, for the time integrator, nonlinear solver, and linear solver. For the linear solver phase, PDE device simulation is usually successful with iterative methods. For circuit simulation direct methods are generally much more reliable than iterative methods, although research in this area is ongoing [27].

For the nonlinear solve, device simulation can easily be handled using a conventional damped Newton, as long as the simulator is provided with a good initial guess. Equation (9.44) is modified as

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\Delta\mathbf{w}, \tag{9.45}$$

in which α is a scalar that is adjusted such that $\mathbf{w}_{k+1}$ results in a reduced value for $|\mathbf{f}(\mathbf{w}_{k+1})|$.

Circuit simulation is somewhat more problematic, in that applying a single scalar α to the entire update is often not adequate to converge the operating point calculation. Many of the nonlinear circuits in the Xyce test suite will fail if such an approach is taken. A more effective approach for circuits is voltage limiting [134]. The junction voltages of nonlinear devices such as BJT's, MOSFET's and diodes are restricted from varying too much from one Newton iteration to the next. This is similar to damping, except that it is applied on a device-by-device basis, and the limiting applied in one device may not be consistent with the limiting applied in another. The implementation in Xyce is similar to that of legacy circuit simulators, in that a correction due to voltage limiting has to be applied on the right hand side of equation 9.43. This takes the form

$$\mathbf{J}(\mathbf{w}')\Delta\mathbf{w}_{total} = -\mathbf{f}(\mathbf{w}') + \mathbf{J}(\mathbf{w}')\Delta\mathbf{w}_{limit}, \tag{9.46}$$

where $\Delta\mathbf{w}_{limit}$ is the change in $\mathbf{w}$ at the current Newton step due to voltage limiting, and $\Delta\mathbf{w}_{total} = \Delta\mathbf{w}_{Newton} + \Delta\mathbf{w}_{limit}$. The quantities $\mathbf{w}_{k+1}$, $\mathbf{w}_k$ and $\mathbf{w}'_k$ are solution vectors related by

$$\mathbf{w}'_k = \mathbf{w}_k + \Delta\mathbf{w}_{limit}, \tag{9.47}$$

$$\mathbf{w}_{k+1} = \mathbf{w}'_k + \Delta\mathbf{w}_{Newton}. \tag{9.48}$$

The limits are generally evaluated and applied in terms of voltage drops within individual devices, so the correction term in (9.46) is applied during the load phase of the individual devices.

Although voltage limiting has been around for a long time [169], and has been surprisingly successful, it has the drawback of being incompatible with other nonlinear solver methods. It is a source of non-smooth behavior in part of the solver, which can be problematic. More recently, homotopy methods have been applied successfully to circuit simulation [162, 163], and they have become the method of choice for modern circuit codes.

9.5.3 Coupling Algorithms

As described in the introduction, techniques for circuit-device coupling have been described by Mayaram [160], and later by Rotella [185]. In these works, two general approaches have been presented, the "full Newton" method and the "two-level Newton" method.

9.5.3.1 Tight Coupling: Full Newton

The "full Newton" method is one in which the entire problem is solved simultaneously as part of a single nonlinear system of equations. Each PDE device is represented by large sub-blocks of the Newton matrix, and the circuit portion of the problem is also represented by a sub-block. In general, this will not be the initial ordering chosen by Xyce. In its current capability, Xyce will treat each PDE device the same as it would any other traditional **SPICE**-style device in the circuit from a topological perspective. As a result, the "circuit" part of the matrix will often not be represented by a single continuous sub-block, but rather will have many components in and around the PDE device sub-blocks.

9.5.3.2 Loose Coupling: Two-Level Newton

The "Two-Level Newton" was first proposed by Mayaram [160]. In this approach, the PDE device and circuit problems are considered separately, with the required information being passed back and forth between the simulators. The adjective "Two-Level" refers to the fact that the nonlinear solution of the circuit problem is considered as an outer level control loop, while for PDE device problem(s), the nonlinear solution is considered to be the inner level. In the most common implementation, the inner level PDE device problem is completely solved at each circuit Newton step. This results in a much slower simulation than the "full Newton" approach, but it has been shown to be much more robust [160, 185].

The PDE sub-problems are solved in the same manner as a stand-alone PDE simulator, with voltages from the external circuit applied as boundary conditions. The circuit problem which comprises the outer loop, is solved similarly to a stand-alone circuit code as well, with the PDE devices represented by extracted conductances and currents. The procedure for extracting device conductances has been described in detail by Mayaram [160]. The sub-block in the circuit matrix associated with each PDE device is an N x N conductance matrix, where N is the number of electrodes. Each entry in this conductance matrix is given by

$$G_{mn} = \frac{\partial i_m}{\partial V_n} = \frac{\partial I_m}{\partial V_n} + \frac{\partial I_m}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial V_n}, \tag{9.49}$$

where m is the row index and n is the column index. From the perspective of the circuit Jacobian, m corresponds to the KCL equation associated with the circuit node attached to the mth electrode of the PDE device, and n corresponds to the voltage variable for the circuit node attached to the nth electrode. Also, i_m is the current at the circuit node attached to electrode m, I_m is also a current, but represents the current at electrode m, from the perspective of the PDE simulation. Of course

$$i_m(V_n) = I_m(\mathbf{w}, V_n). \tag{9.50}$$

Most of the terms in (9.49) correspond to Jacobian terms that need to be calculated for the full Newton approach. The first term of (9.49), $\partial I_m/\partial V_n$, corresponds to the PDE device contribution to the circuit node KCL on the Jacobian matrix diagonal, and is zero if $m \neq n$. The vector $\partial \mathbf{w}/\partial V_n$ is determined by first rewriting (9.42) as

$$\mathbf{f}_{PDE}(\mathbf{w}_{PDE}(V), V) = 0. \tag{9.51}$$

This is the equivalent of (9.42), except that it is only with respect to one PDE sub-problem. The residual vector of the PDE device $\mathbf{f}_{PDE}$ is the solution vector for the PDE device, and V refers to externally applied voltages, which will be coming from the connected circuit. Taking the derivative of this system with respect to V, and re-arranging, the linear system

$$\mathbf{J}_{PDE}\frac{\partial \mathbf{w}_{PDE}}{\partial V_n} = -\frac{\partial \mathbf{f}_{PDE}}{\partial V_n} \tag{9.52}$$

is obtained, where $\mathbf{J}_{PDE}$, or $\partial \mathbf{f}_{PDE}/\partial \mathbf{w}_{PDE}$, is the Jacobian matrix from the PDE problem. This linear system has to be solved for each electrode. The object $\partial \mathbf{f}_{PDE}/\partial V_n$ is a sparse vector with nonzero terms only for mesh elements connected directly to the electrode n; It corresponds to a single column of the full-Newton Jacobian.

Xyce has a slightly different implementation than the one described by Mayaram [160]. This design was chosen in part for expedient implementation, as the same matrix structure can be used at both levels of the two-Level Newton method. Equations can be removed from the problem by placing 1.0 on the diagonal and 0.0 at every other location in the problem's respective matrix row, and 0.0 for the corresponding element in the residual vector. By using this shortcut, different parts of the problem can be "turned off" and "turned on", depending upon the current two-level Newton phase.

This implementation has some advantages. An optimal approach to a transient mixed-level simulation is to have the simulator apply a two-level Newton algorithm for the DC operating point, and a full Newton algorithm during the transient phase, and switching between the two modes requires very little work. During the inner loop of the two-level solve, all the PDE devices are solved simultaneously even though they are not coupled together. Several instances of (9.52) can also solved simultaneously as well, with the nth electrode of each PDE device addressed in one matrix solve.

9.5.3.3 Loose Coupling: Modified Two-Level Newton

One of the major difficulties of solving a mixed-level problem is that PDE simulation is very sensitive to the initial guess. The voltages of connected circuit nodes provide voltage boundary conditions to PDE devices, and, over the course of a two-level-Newton solve, these voltages may change a great deal at each outer loop step. A PDE simulator can often handle abrupt voltage boundary condition changes, if they are on the order the built-in potential of the device. Changes much greater than that (hundreds of volts, or thousands of volts, etc.) will almost always prevent the PDE problem from converging on the inner Newton loop, if the result of the previous inner loop solve is used as the initial guess.

Mayaram [160] has proposed a modified two-level-Newton scheme, in which a prediction of the PDE device solution can be obtained using the derivative information of the two-level Newton. This predicted solution can then be used as an initial guess for the PDE problem's Newton loop. Mayaram uses the expression

$$\mathbf{w}_{PDE}^{k+1} = \mathbf{w}_{PDE}^{k} + \sum_{n=0}^{n=N} \left[\frac{\partial \mathbf{w}_{PDE}}{\partial V_n} \right] \Delta V_n \tag{9.53}$$

to get the initial guess for each problem, where k is the iteration index of the outer Newton loop and ΔV_n is the change in applied voltage at the electrode n between outer loop iterations. A potential drawback of this approach is that if ΔV_n is large enough, and negative, the resulting $\mathbf{w}^{k+1}$ may contain nonphysical negative species densities. Equation (9.53) has been implemented in Xyce, but so far has not been very successful because of this issue. However, using this technique to obtain the initial guess has been shown to be very effective [160], so more study is needed. Possibly, if voltage limiting is employed (discussed later in this document), the change in nodal voltage can be kept small enough for (9.53) to give a good guess.

9.5.3.4 Loose Coupling: Two-Level Newton with Continuation

A very robust approach to obtaining the solution to PDE sub-problems is to apply a simple continuation algorithm to the PDE. At the beginning of each inner loop PDE solve, an assessment is made of how much the connected circuit node voltages have changed. If this change is large, then instead of a single Newton solve, the inner PDE problem is addressed with a series of Newton solves is undertaken. The electrode voltage boundary conditions are changed gradually prior to each solve, and each solve is converged completely before moving on to the next one.

Initially, we implemented the continuation algorithm to vary the electrode potentials by a fixed step-size ΔV at each continuation step. The maximum voltage change to which a PDE device can be subjected and still achieve convergence is problem dependent, so it is difficult to predict exactly how much change can be tolerated by each PDE. Using a constant step-size approach can result in taking a prohibitive number of continuation steps, to account for the worst case scenario.

To overcome this we implemented a simple variable step-size continuation algorithm, in which the step-size ΔV changes depending upon the success or failure of the previous attempted step. Each time a continuation step is successful the step-size is increased by a factor α. For each failure, the step-size is reduced by another factor β, and the step is re-taken. For this work, we chose $\alpha = 1.5$ and $\beta = 0.125$. To prevent the scheme from being too aggressive, the step-size is only increased when the number of successive successful steps is equal to or greater than the most recent number of successive failed steps. By using this variable step-size algorithm, the number of required continuation steps was often reduced by over an order of magnitude compared with the constant step-size approach.

This approach to the two-level Newton has proven to be very robust. As of this writing, the inner-loop solve has never failed using this approach, but it has the disadvantage of being relatively slow. If the circuit voltages are changing a great deal, the continuation algorithm may take hundreds of steps, even using the variable step-size algorithm. If the number of continuation steps required for the inner solve is steadily increasing, that is usually a sign that the two-level algorithm is diverging. Fortunately, for most circuits, large changes in nodal voltages only occur during the first few outer-loop Newton steps.

9.5.3.5 Voltage Limiting with Two-Level Newton

Voltage limiting can be applied to mixed-level simulation, and is often necessary for cases where the circuit portion of the problem is difficult to solve. Applying it in the case of the full-Newton approach is logistically problematic, but applying voltage limiting within the context of a two-level algorithm is straightforward. The two-level Newton is easier because the correction term in (9.46) can be applied in terms of the extracted conductances, instead of in terms of the equations of the PDE device.

Additionally, device simulation usually requires conventional Newton damping, but applying damping to the PDE device part of the problem, while applying voltage limiting to the circuit part of the problem, is very difficult. With a two-level approach, this issue is avoided. Also, by applying voltage limiting on the circuit level, the change in nodal voltages seen by the PDE device is relatively small, and the number of required continuation steps is correspondingly small.

9.6 Conclusions

A mathematical formulation for mixed-level simulation algorithms has been developed where different physics interact at potentially different spatial resolutions in a single domain. To minimize the implementation effort, explicit solution methods can be considered, however, implicit methods are preferred if computational efficiency is of high priority. We present the use of a partial elimination nonlinear solver technique to solve these mixed-level problems and show how these formulation are closely coupled to SAND optimization approaches and sensitivity analyses. We discuss the details of a potential application in the electrical simulation area. The implementation of these techniques are future work.

Chapter 10

Time-Domain Decomposition Iterative Methods for the Solution of PDE Constrained Optimization Problems

10.1 Introduction

Optimization problems governed by time dependent partial differential equations (PDEs) arise in the context of optimal control, optimal design, and parameter identification problems in applications involving fluid flow [2, 63, 101, 103, 104, 125, 140, 117, 118], convective-diffusive transport [7], phase transition [112, 121], and superconductivity [102], among many others. One of the challenges that arises in the numerical solution of such problems is the strong coupling in space and time of the state (solution of the governing PDE), the so-called adjoint, and the control. This chapter discusses time-domain decomposition to deal with this problem for a class of optimal control problems governed by parabolic PDEs

To make the discussion more concrete, we consider problems of the form

$$\text{minimize} \quad \int_0^T l(t, y(t), u(t))dt + \psi(y(T)), \tag{10.1a}$$

subject to

$$\frac{\partial}{\partial t}y(t) + F(t, y(t), u(t)) = 0, \quad t \in (0, T) \tag{10.1b}$$

$$y(0) = y_0. \tag{10.1c}$$

In (10.1) the function u is the control to be determined and y denotes the solution of the partial differential equation (10.1b) with initial condition (10.1c). All applications cited above, fit into this context with a suitable problem formulation in a Banach or Hilbert space setting. The precise formulation is typically problem dependent and for the various applications discussed in the references above. Additionally, abstract infinite dimensional optimization problems governed by partial differential equations (PDEs) and ordinary differential equations (ODEs) are studied in [81] and [127], respectively. To present the issues and discuss the ideas for the solution of (10.1) it is sufficient to proceed formally, at this time.

As mentioned earlier, one challenge for the practical solution of optimization problems of the type (10.1) is their strong coupling in space and time, which can be seen from the necessary optimality conditions for (10.1). Formally, the necessary optimality conditions for (10.1) are given as follows. If y, u solve (10.1), then there exists a function λ such that y, u, λ solve the state equations (10.1b),(10.1c), the adjoint equations

$$-\frac{\partial}{\partial t}\lambda(t) + F_y(t, y(t), u(t))^*\lambda(t) = -\nabla_y l(t, y(t), u(t)), \quad t \in (0, T), \tag{10.2a}$$

$$\lambda(T) = -\nabla\psi(y(T)), \tag{10.2b}$$

and the equation

$$\nabla_u l(t, y(t), u(t)) + F_u(t, y(t), u(t))^*\lambda(t) = 0, \quad t \in (0, T). \tag{10.3}$$

Here subscripts u and y denote partial derivatives and given an operator M, its adjoint is denoted by M^*. If M is a real matrix, $M^* = M^T$.

While the state equation (10.1b),(10.1c) is solved forward in time, the adjoint equation (10.2) is solved backward in time. Both equations are coupled via (10.3). In most applications of interest, the amount of storage required to hold state, adjoint and control information easily exceeds the amount of memory available.

To cope with the huge storage requirements, several techniques have been proposed. Among them are storage management techniques instantaneous control, receeding horizon or moving horizon methods and model reduction techniques. See [111] for a brief overview in the context of time domain decomposition methods. We note that instantaneous control, receeding horizon or moving horizon methods as well as model reduction techniques also address aspects other than the storage requirements arising in the practical solution of problems of the type (10.1). Instantaneous control, receeding horizon, moving horizon methods, and model reduction techniques alter the optimization problem that is ultimately been solved. The focus of this paper is on a class of iterative methods for the solution of (10.1).

Our time–domain decomposition is a multiple shooting approach that equivalently reformulates (10.1) as a so-called discrete-time optimal control (DTOC) problem. For our target problems the operators/matrices arising in the multiple shooting reformulation of the problem cannot be computed explicitly; only operator/matrix–vector products can be formed. Therefore existing multiple shooting implementations for systems of ODEs or 'small' PDEs (see, e.g., [13, 34, 41, 45, 73, 142, 143, 176, 202]), which rely on matrix factorizations are not applicable. Our approach is matrix free. It requires the solution of discretized PDEs (10.1b), (10.2), or linearizations of (10.1b) on smaller time domains and, depending on the iterative scheme build on the time-domain decomposition, the approximate iterative solution of optimal control problems which are essentially smaller copies of (10.1) restricted to smaller time domains. For these subproblems existing solver methods can be used.

The time-domain decomposition of (10.1) offers two potential benefits. First, depending on the solution algorithm used to solve the time-decomposed problem, parallelism is introduced in the time-domain. Additional parallelism may be introduced into the solution of subtasks within the individual time domains. Secondly, the time sub-interval problems are smaller than (10.1). Even if solutions to state and adjoint equations restricted to time subdomains cannot be stored in memory, storage management techniques [99] are more efficient the larger the number of checkpoints (i.e., the larger the amount of storage) per number of time steps (see [120]).

10.2 Temporal decomposition of the problem

We use a multiple shooting approach to reformulate the optimization problem (10.1). We select a partition

$$0 = T_0 < T_1 \ldots < T_N = T$$

of $[0, T]$ and we introduce the auxiliary variables

$$\overline{y}_i, \quad n = 0, \ldots, N-1,$$

where

$$\overline{y}_0 \stackrel{\text{def}}{=} y_0.$$

Moreover, we set

$$\overline{u}_i = u|_{(T_i, T_{i+1})}, \quad i = 0, \ldots, N-1.$$

Assume that for every $\overline{y}_i \in H$, $\overline{u}_i \in U_i$ the time-subdomain state equation

$$\begin{aligned} \frac{\partial}{\partial t} y_i(t) + F(t, y_i(t), \overline{u}_i(t)) &= 0, \quad t \in (T_i, T_{i+1}) && \text{(10.4a)} \\ y_i(T_i) &= \overline{y}_i, && \text{(10.4b)} \end{aligned}$$

$i = 0, \ldots, N-1$, has a unique solution

$$y_i(\cdot; \overline{y}_i, \overline{u}_i).$$

If we enforce the continuity conditions

$$\overline{y}_{i+1} = y_i(T_{i+1}; \overline{y}_i, \overline{u}_i), \qquad i = 0, \ldots, N-2, \tag{10.5}$$

then the function y, which is defined by 'glueing' the y_i's together, solves the original state equation (10.1b), (10.1c). If y solves (10.1b), (10.1c), then $\overline{y}_i = y(T_i)$, $y_i = y|_{\Omega \times (T_i, T_{i+1})}$, $i = 0, \ldots, N-1$, solves (10.4),(10.5) and vice versa.

Thus, we can rewrite the optimal control problem (10.1) equivalently as follows:

$$\begin{aligned} &\text{Minimize} && \sum_{i=0}^{N-1} \int_{T_i}^{T_{i+1}} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t)) dt + \psi(y_{N-1}(T_N; \overline{y}_{N-1}, \overline{u}_{N-1})) && \text{(10.6)} \\ &\text{subject to} && \overline{y}_{i+1} = y_i(T_{i+1}; \overline{y}_i, \overline{u}_i), \qquad i = 0, \ldots, N-2. \end{aligned}$$

Here $y_i(\cdot; \overline{y}_i, \overline{u}_i)$ is the (by assumption) unique solution of (10.4) for given data $\overline{y}_i$ and $\overline{u}_i$. Problems of the form (10.6) are also known as discrete time optimal control problems (DTOCPs).

Note, the optimal control problem in the original formulation (10.1) is an optimization problem in the variables y, u. Its reformulation (10.6) is an optimization problem in the variables $\overline{y}_i$, $i = 1, \ldots, N-1$, and $\overline{u}_i$, $i = 0, \ldots, N-1$. The problem (10.6) is a *multiple shooting* reformulation of (10.1). This reformulation is well-known in the context of control problems governed by ODEs or DAEs [13, 34, 45, 73, 142, 143, 202]. The difference between the problems we consider and the problems

considered in the traditional multiple shooting context is their size. Techniques discussed e.g., in the references above for the solution of multiple shooting formulations of optimal control problems governed by ODEs, DAEs, or even 'small' PDEs are not feasible in our context. The goal of this paper is the discussion of matrix-free methods for the solution of (10.6).

The Lagrangian associated with (10.6) is given by

$$\begin{aligned}\bar{L}(\overline{y},\overline{u},\overline{\lambda}) &= \sum_{i=0}^{N-1}\int_{T_i}^{T_{i+1}} l(t,y_i(t;\overline{y}_i,\overline{u}_i),\overline{u}_i(t))dt + \psi(y_{N-1}(T_N;\overline{y}_{N-1},\overline{u}_{N-1})) \\ &\quad + \sum_{i=0}^{N-2}\langle\overline{\lambda}_{i+1},\overline{y}_{i+1} - y_i(T_{i+1};\overline{y}_i,\overline{u}_i)\rangle. \end{aligned} \tag{10.7}$$

For brevity we set $y_i = y_i(\cdot;\overline{y}_i,\overline{u}_i)$ and for $i = 1,\ldots,N-1$ we consider

$$-\frac{\partial}{\partial t}\lambda_i(t) + F_y(t,y_i(t),\overline{u}_i(t))^*\lambda_i(t) = -\nabla_y l(t,y_i(t),\overline{u}_i(t)), \quad t\in(T_i,T_{i+1}), \tag{10.8a}$$

$$\lambda_i(T_{i+1}) = \begin{cases} -\nabla\psi(y_i(T)), & \text{if } i = N-1, \\ \overline{\lambda}_{i+1}, & \text{else.}\end{cases} \tag{10.8b}$$

Let $\lambda_i(\cdot;\overline{y}_i,\overline{u}_i,\overline{\lambda}_{i+1})$, $i = 1,\ldots,N-2$, $\lambda_{N-1}(\cdot;\overline{y}_{N-1},\overline{u}_{N-1})$, be the solution of (10.8). To avoid distinction between $i = 1,\ldots,N-2$, and $i = N-1$, we use the notation $\lambda_i(\cdot;\overline{y}_i,\overline{u}_i,\overline{\lambda}_{i+1})$ for all $i = 1,\ldots,N-1$. Since there is no $\overline{\lambda}_N$, this should be understood as $\lambda_i(\cdot;\overline{y}_i,\overline{u}_i)$ if $i = N-1$.

Under suitable conditions, the necessary optimality conditions for the problem (10.6) are given by the DTOCP adjoint equations

$$\overline{\lambda}_i = \lambda_i(T_i;\overline{y}_i,\overline{u}_i,\overline{\lambda}_{i+1}), \qquad i = 1,\ldots,N-1, \tag{10.9a}$$

the DTOCP gradient equation

$$\begin{aligned}\nabla_u l(t,y_i(t;\overline{y}_i,\overline{u}_i),\overline{u}_i(t)) + F_u(t,y_i(t;\overline{y}_i,\overline{u}_i),\overline{u}_i(t))^*\lambda_i(t;\overline{y}_i,\overline{u}_i,\overline{\lambda}_{i+1}) &= 0, \\ t\in(T_i,T_{i+1}),\quad i = 0,\ldots,N-1,&\end{aligned} \tag{10.9b}$$

and the DTOCP state equation

$$\overline{y}_{i+1} = y_i(T_{i+1};\overline{y}_i,\overline{u}_i), \qquad i = 0,\ldots,N-2. \tag{10.9c}$$

10.3 The Lagrange-Newton-SQP Method

We view (10.9) as a system

$$\mathbf{G}(\mathbf{x}) = \mathbf{0} \tag{10.10}$$

and solve it using Newton's method. This approach is locally (i.e., if $\mathbf{x}$ is near a point that satisfies the second order sufficient optimality conditions) equivalent to the Lagrange-Newton-SQP method.

The variables $\mathbf{x}$ and the operator $\mathbf{G}$ in (10.10) are specified in Figure 10.1. The evaluation of $\mathbf{G}(\mathbf{x})$ is easily parallelized across time subdomains.

Algorithm 13. [Evaluation of $\mathbf{G}(\mathbf{x})$] For $i = 0, \ldots, N-1$ do (in parallel)

1. Compute solution $y_i(t; \overline{y}_i, \overline{u}_i)$ of the time subinterval state equation (10.4).
2. Compute solution $\lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ of the time subinterval adjoint equation (10.8).
3. Evaluate

$$\nabla_u l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t)) + F_u(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}) \quad t \in (T_i, T_{i+1}),$$
$$\overline{y}_{i+1} - y_i(T_{i+1}; \overline{y}_i, \overline{u}_i),$$
$$\overline{\lambda}_i - \lambda_i(T_i; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}).$$

Figure 10.1. The variables $\mathbf{x}$ and the operator $\mathbf{G}$

$$\mathbf{x} = \left(\begin{array}{c} \overline{y}_1 \\ \overline{u}_0 \\ \hline \overline{y}_2 \\ \overline{u}_1 \\ \overline{\lambda}_1 \\ \hline \overline{y}_3 \\ \overline{u}_2 \\ \overline{\lambda}_2 \\ \hline \vdots \\ \vdots \\ \hline \overline{y}_{N-1} \\ \overline{u}_{N-2} \\ \overline{\lambda}_{N-2} \\ \hline \overline{u}_{N-1} \\ \overline{\lambda}_{N-1} \end{array}\right), \qquad \mathbf{G}(\mathbf{x}) = \left(\begin{array}{c} \nabla_u l(\cdot, y_0(\cdot;\overline{y}_0,\overline{u}_0),\overline{u}_0) + F_u(\cdot, y_0(\cdot;\overline{y}_0,\overline{u}_0),\overline{u}_0)^* \lambda_0(\cdot;\overline{y}_0,\overline{u}_0,\overline{\lambda}_1) \\ \overline{y}_1 - y_0(T_1;\overline{y}_0,\overline{u}_0), \\ \hline \overline{\lambda}_1 - \lambda_1(T_1;\overline{y}_1,\overline{u}_1,\overline{\lambda}_2) \\ \nabla_u l(\cdot, y_1(\cdot;\overline{y}_1,\overline{u}_1),\overline{u}_1) + F_u(\cdot, y_1(\cdot;\overline{y}_1,\overline{u}_1),\overline{u}_1)^* \lambda_1(\cdot;\overline{y}_1,\overline{u}_1,\overline{\lambda}_2) \\ \overline{y}_2 - y_1(T_2;\overline{y}_1,\overline{u}_1) \\ \hline \overline{\lambda}_2 - \lambda_2(T_2;\overline{y}_2,\overline{u}_2,\overline{\lambda}_3) \\ \nabla_u l(\cdot, y_2(\cdot;\overline{y}_2,\overline{u}_2),\overline{u}_2) + F_u(\cdot, y_2(\cdot;\overline{y}_2,\overline{u}_2),\overline{u}_2)^* \lambda_2(\cdot;\overline{y}_2,\overline{u}_2,\overline{\lambda}_3) \\ \overline{y}_3 - y_2(T_3;\overline{y}_2,\overline{u}_2) \\ \hline \vdots \\ \vdots \\ \hline \overline{\lambda}_{N-2} - \lambda_{N-2}(T_{N-2};\overline{y}_{N-2},\overline{u}_{N-2},\overline{\lambda}_{N-1}) \\ \nabla_u l(\cdot, y_{N-2}(\cdot;\overline{y}_{N-2},\overline{u}_{N-2}),\overline{u}_{N-2}) + F_u(\cdot, y_{N-2}(\cdot;\overline{y}_{N-2},\overline{u}_{N-2}),\overline{u}_{N-2})^* \lambda_{N-2}(\cdot;\overline{y}_{N-2},\overline{u}_{N-2},\overline{\lambda}_{N-1}) \\ \overline{y}_{N-1} - y_{N-2}(T_{N-1};\overline{y}_{N-2},\overline{u}_{N-1}) \\ \hline \overline{\lambda}_{N-1} - \lambda_{N-1}(T_{N-1};\overline{y}_{N-1},\overline{u}_{N-1}) \\ \nabla_u l(\cdot, y_{N-1}(\cdot;\overline{y}_{N-1},\overline{u}_{N-1}),\overline{u}_{N-1}) + F_u(\cdot, y_{N-1}(\cdot;\overline{y}_{N-1},\overline{u}_{N-1}),\overline{u}_{N-1})^* \lambda_{N-1}(\cdot;\overline{y}_{N-1},\overline{u}_{N-1}) \end{array}\right).$$

Formally (this can be made rigorous for the applications cited in the introdution) the Fréchet derivative of $\mathbf{G}$ at $\mathbf{x}$ in the direction

$$\boldsymbol{\delta}\mathbf{x} = \left(\begin{array}{c} \overline{\delta y}_1 \\ \overline{\delta u}_0 \\ \hline \overline{\delta y}_2 \\ \overline{\delta u}_1 \\ \overline{\delta \lambda}_1 \\ \hline \vdots \\ \vdots \\ \hline \overline{\delta y}_{N-1} \\ \overline{\delta u}_{N-2} \\ \overline{\delta \lambda}_{N-2} \\ \hline \overline{\delta u}_{N-1} \\ \overline{\delta \lambda}_{N-1} \end{array} \right)$$

involves the solution $z_i = z_i(\cdot; \overline{y}_i, \overline{u}_i; \overline{\delta y}_i, \overline{\delta u}_i)$ of the linearized time subinterval state equation

$$\begin{aligned} &\frac{\partial}{\partial t} z_i(t) + F_y(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ z_i(t) \\ &\qquad + F_u(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ \overline{\delta u}_i(t) \quad &= \quad 0, \quad t \in (T_i, T_{i+1}) \qquad (10.11a) \\ &\qquad\qquad z_i(T_i) \quad &= \quad \begin{cases} 0, & \text{if } i = 0, \\ \overline{\delta y}_i, & \text{else}, \end{cases} \qquad (10.11b) \end{aligned}$$

$i = 0, \ldots, N-1$, where $y_i(\cdot; \overline{y}_i, \overline{u}_i)$ is the solution of (10.4), and the solution $\eta_i = \eta_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta \lambda}_{i+1})$ of the linearized time subinterval adjoint equation

$$\begin{aligned} &-\frac{\partial}{\partial t}\eta_i(t) + F_y(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))^* \eta_i(t) \\ &\quad = \quad -\Big(F_{yy}(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ z_i(t; \overline{y}_i, \overline{u}_i; \overline{\delta y}_i, \overline{\delta u}_i) \Big)^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}) \\ &\qquad -\Big(F_{yu}(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ \overline{\delta u}_i(t) \Big)^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}) \\ &\qquad -\nabla_{yu} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t)) \overline{\delta u}_i(t) \\ &\qquad -\nabla_{yy} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t)) z_i(t; \overline{y}_i, \overline{u}_i; \overline{\delta y}_i, \overline{\delta u}_i), \qquad t \in (T_i, T_{i+1}), \qquad (10.12a) \end{aligned}$$

$$\eta_i(T_{i+1}) = \begin{cases} -\nabla_{yy} \psi(y_i(T; \overline{y}_i, \overline{u}_i)) z_i(T; \overline{y}_i, \overline{u}_i; \overline{\delta y}_i, \overline{\delta u}_i), & \text{if } i = N-1, \\ \overline{\delta \lambda}_{i+1}, & \text{else}, \end{cases} \qquad (10.12b)$$

$i = 1, \ldots, N-1$, where $y_i(\cdot; \overline{y}_i, \overline{u}_i)$ is the solution of (10.4), $\lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ is the solution of (10.8), and $z_i(\cdot; \overline{y}_i, \overline{u}_i; \overline{\delta y}_i, \overline{\delta u}_i)$ is the solution of the linearized state equation (10.11).

The Fréchet derivative of $\mathbf{G}$ at $\mathbf{x}$ in the direction $\boldsymbol{\delta}\mathbf{x}$ is specified in Figure 10.2.

Figure 10.2. The operator vector product $\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x}$. (Here, $y_i = y_i(\cdot;\overline{y}_i,\overline{u}_i)$ and $\lambda_i = \lambda_i(\cdot;\overline{y}_i,\overline{u}_i,\overline{\lambda}_{i+1})$ are the solution of the time subinterval state equation (10.4) and the time subinterval adjoint equation (10.8), respectively, and $z_i = z_i(\cdot;\overline{y}_i,\overline{u}_i,\overline{\delta y}_i,\overline{\delta u}_i)$ and $\eta_i = \eta_i(\cdot;\overline{y}_i,\overline{u}_i,\overline{\lambda}_{i+1},\overline{\delta y}_i,\overline{\delta u}_i,\overline{\delta\lambda}_{i+1})$ are the solution of the time subinterval linearized state equation (10.11) and the time subinterval linearized adjoint equation (10.12), respectively).

$$
\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x} = \left(\begin{array}{c}
\nabla_{uu}l(\cdot,y_0,\overline{u}_0)\overline{\delta u}_0 + \nabla_{uy}l(\cdot,y_0,\overline{u}_0)z_0 + \left(F_{uu}(\cdot,y_0,\overline{u}_0)\,\overline{\delta u}_0\right)^*\lambda_0 + \left(F_{uy}(\cdot,y_0,\overline{u}_0)\,z_0\right)^*\lambda_0 + F_u(\cdot,y_0,\overline{u}_0)^*\eta_0 \\
\overline{\delta y}_1 - z_0(T_1), \\
\hline
\overline{\delta\lambda}_1 - \eta_1(T_1) \\
\nabla_{uu}l(\cdot,y_1,\overline{u}_1)\overline{\delta u}_1 + \nabla_{yu}l(\cdot,y_1,\overline{u}_1)z_1 + \left(F_{uu}(\cdot,y_1,\overline{u}_1)\,\overline{\delta u}_1\right)^*\lambda_1 + \left(F_{uy}(\cdot,y_1,\overline{u}_1)\,z_1\right)^*\lambda_1 + F_u(\cdot,y_1,\overline{u}_1)^*\eta_1 \\
\overline{\delta y}_2 - z_1(T_2) \\
\hline
\overline{\delta\lambda}_2 - \eta_2(T_2) \\
\nabla_{uu}l(\cdot,y_2,\overline{u}_2)\overline{\delta u}_2 + \nabla_{uy}l(\cdot,y_2,\overline{u}_2)z_2 + \left(F_{uu}(\cdot,y_2,\overline{u}_2)\,\overline{\delta u}_2\right)^*\lambda_2 + \left(F_{uy}(\cdot,y_2,\overline{u}_2)\,z_2\right)^*\lambda_2 + F_u(\cdot,y_2,\overline{u}_2)^*\eta_2 \\
\overline{\delta y}_3 - z_2(T_3) \\
\hline
\vdots \\
\vdots \\
\hline
\overline{\lambda}_{N-2} - \lambda_{N-2}(T_{N-2};\overline{y}_{N-2},\overline{u}_{N-2},\overline{\lambda}_{N-1}) \\
\nabla_{uu}l(\cdot,y_{N-2},\overline{u}_{N-2})\overline{\delta u}_{N-2} + \nabla_{uy}l(\cdot,y_{N-2},\overline{u}_{N-2})z_{N-2}\cdots \\
\cdots + \left(F_{uu}(\cdot,y_{N-2},\overline{u}_{N-2})\,\overline{\delta u}_{N-2}\right)^*\lambda_{N-2} + \left(F_{uy}(\cdot,y_{N-2},\overline{u}_{N-2})\,z_{N-2}\right)^*\lambda_{N-2} + F_u(\cdot,y_{N-2},\overline{u}_{N-2})^*\eta_{N-2} \\
\overline{\delta y}_{N-1} - z_{N-2}(T_{N-1}) \\
\hline
\overline{\delta\lambda}_{N-1} - \eta_{N-1}(T_{N-1}) \\
\nabla_{uu}l(\cdot,y_{N-1},\overline{u}_{N-1})\overline{\delta u}_{N-1} + \nabla_{uy}l(\cdot,y_{N-1},\overline{u}_{N-1})z_{N-1}\cdots \\
\cdots + \left(F_{uu}(\cdot,y_{N-1},\overline{u}_{N-1})\,\overline{\delta u}_{N-1}\right)^*\lambda_{N-1} + \left(F_{uy}(\cdot,y_{N-1},\overline{u}_{N-1})\,z_{N-1}\right)^*\lambda_{N-1} + F_u(\cdot,y_{N-1},\overline{u}_{N-1})^*\eta_{N-1}
\end{array}\right).
$$

The computations required to compute the Fréchet derivative of $\mathbf{G}$ at $\mathbf{x}$ in a direction $\boldsymbol{\delta}\mathbf{x}$ are detailed in Algorithm 14.

Algorithm 14. [Evaluation of $\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x}$]

For $i = 0, \ldots, N-1$ do (in parallel)

1. Compute the solution $y_i = y_i(\cdot; \overline{y}_i, \overline{u}_i)$ of the time subinterval state equation (10.4) (by resolving (10.4) or by retreiving $y_i(\cdot; \overline{y}_i, \overline{u}_i)$ from storage).

2. Compute the solution $\lambda_i = \lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ of the time subinterval adjoint equation (10.8) (by resolving (10.4) or by retreiving $\lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ from storage).

3. Compute the solution $z_i = z_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i)$ of the time subinterval linearized state equation (10.11).

4. Compute the solution $\eta_i = \eta_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1})$ of the time subinterval lineearized adjoint equation (10.12).

5. Evaluate

$$
\begin{aligned}
&\overline{\delta\lambda}_i - \eta_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1}),\\
&\nabla_{uu} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\overline{\delta u}_i(t)\\
&+\nabla_{uy} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t)) z_i(t; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i)\\
&+\Big(F_{uu}(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ \overline{\delta u}_i(t)\Big)^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})\\
&+\Big(F_{uy}(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ z_i(t; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i)\Big)^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})\\
&+F_u(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))^* \eta_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1}) \quad t \in (T_i, T_{i+1}),\\
&\overline{\delta y}_{i+1} - z_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i).
\end{aligned}
\tag{10.13}
$$

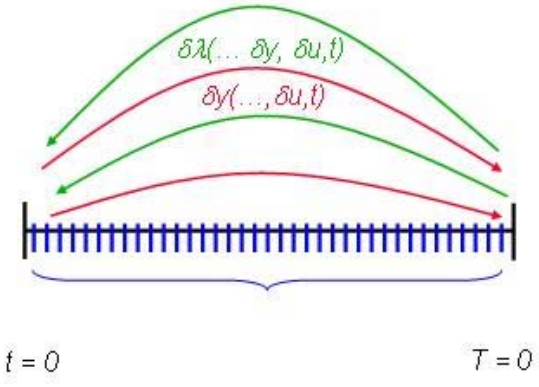
With Algorithms 13 and 14 we are now able to execute an inexact Newton method. The operator $\mathbf{G}'(\mathbf{x})$ cannot be computed explicitly, only operator-vector products can be computed via Algorithm 14. In principle, many Krylov subspace methods can be used to solve the Newton system

$$\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x} = -\mathbf{G}(\mathbf{x}). \tag{10.14}$$

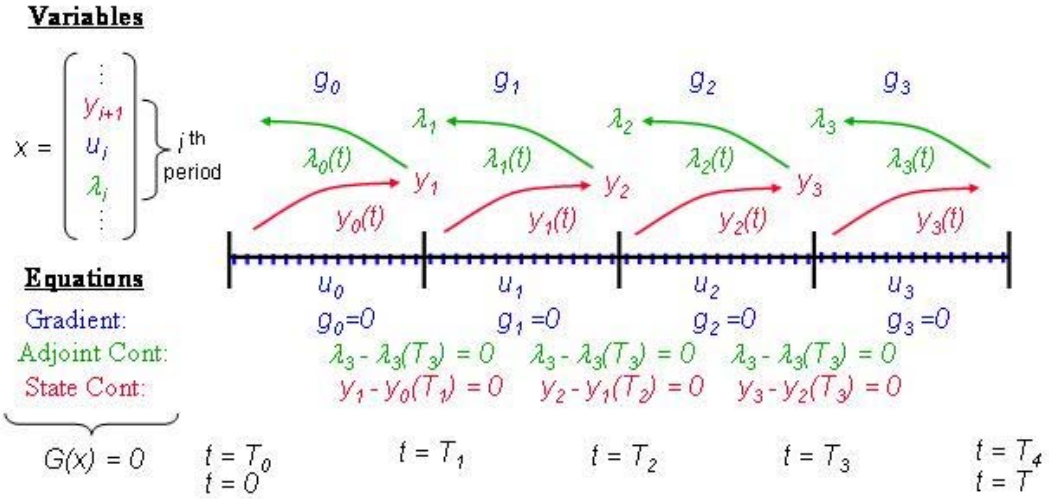
Figure 10.3 provides a general overview of the above described solution procedure in comparison to a standard integration procedure.

In practice, however, one also needs a preconditioner. One class of preconditioners will be discussed in the next section.

Standard Integration Procedure



Multiple Shooting Procedure



10.4 Iterative solution of the Newton system

10.4.1 The Newton system

A closer look at the operator $\mathbf{G}'(\mathbf{x})$ specified in Figure 10.2 will reveal that it has a block tridiagonal structure. This motivates the application of several classical, operator splitting based iterative methods. In our application, they have have nice optimal control interpretations. This section follows [111].

We consider $\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x} = \mathbf{b}$, where $\mathbf{b}$ is an arbitrary vector whose ith block component is given by $(\bar{c}_i, \bar{d}_i, \bar{b}_i)$. The ith block of $\mathbf{G}'(\mathbf{x})\boldsymbol{\delta}\mathbf{x} = \mathbf{b}$ is given by (with obvious modifications for $i = 0$ and $i = N - 1$):

$$\overline{\delta\lambda}_i - \eta_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1}) = \bar{c}_i \tag{10.15a}$$

$$\begin{aligned}
&\nabla_{uu} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\overline{\delta u}_i(t) \\
&+\nabla_{uy} l(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t)) z_i(t; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i) \\
&+\Big(F_{uu}(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ \overline{\delta u}_i(t)\Big)^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}) \\
&+\Big(F_{uy}(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))\ z_i(t; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i)\Big)^* \lambda_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}) \\
&+F_u(t, y_i(t; \overline{y}_i, \overline{u}_i), \overline{u}_i(t))^* \eta_i(t; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1}) = \bar{d}_i(t) \quad t \in (T_i, T_{i+1}),
\end{aligned} \tag{10.15b}$$

$$\overline{\delta y}_{i+1} - z_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i) = \bar{b}_i. \tag{10.15c}$$

The ith block (10.15) depends only on $(\overline{\delta y}_i, \overline{\delta u}_{i-1}, \overline{\delta\lambda}_{i-1})$, $(\overline{\delta y}_{i+1}, \overline{\delta u}_i, \overline{\delta\lambda}_i)$, and $(\overline{\delta y}_{i+2}, \overline{\delta u}_{i+1}, \overline{\delta\lambda}_{i+1})$. Thus, the operator $\mathbf{G}'(\mathbf{x})$ has block-tridiagonal structure.

We will show next that given $(\overline{\delta y}_i, \overline{\delta u}_{i-1}, \overline{\delta\lambda}_{i-1})$ and $(\overline{\delta y}_{i+2}, \overline{\delta u}_{i+1}, \overline{\delta\lambda}_{i+1})$, the ith block (10.15) can be solved for $(\overline{\delta y}_{i+1}, \overline{\delta u}_i, \overline{\delta\lambda}_i)$, and and that this solution of the ith block (10.15) corresponds to solving a linear quadratic optimal control problem on the time subinterval (T_i, T_{i+1}).

By definition, $z_i = z_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\delta y}_i, \overline{\delta u}_i)$ and $\eta_i = \eta_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, \overline{\delta y}_i, \overline{\delta u}_i, \overline{\delta\lambda}_{i+1})$ satisfy (10.11) and (10.12), respectively. Combining these definitions of z_i and η_i with (10.15), we obtain the following system of coupled differential equations (for brevity we set $y_i = y_i(\cdot; \overline{y}_i, \overline{u}_i)$ and $\lambda_i = \lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$)

$$\begin{aligned}
&-\frac{\partial}{\partial t}\eta_i(t) + F_y(t, y_i(t), \overline{u}_i(t))^* \eta_i(t) \\
&= -\Big(F_{yy}(t, y_i(t), \overline{u}_i(t))\ z_i(t)\Big)^* \lambda_i(t) \\
&\quad -\Big(F_{yu}(t, y_i(t), \overline{u}_i(t))\ \overline{\delta u}_i(t)\Big)^* \lambda_i(t) \\
&\quad -\nabla_{yu} l(t, y_i(t), \overline{u}_i(t))\overline{\delta u}_i(t) \\
&\quad -\nabla_{yy} l(t, y_i(t), \overline{u}_i(t)) z_i(t), \quad t \in (T_i, T_{i+1}),
\end{aligned} \tag{10.16a}$$

$$\eta_i(T_{i+1}) = \begin{cases} -\nabla_{yy}\psi(y_i(T))z_i(T), & \text{if } i = N-1, \\ \overline{\delta\lambda}_{i+1}, & \text{else}, \end{cases} \tag{10.16b}$$

$$
\begin{aligned}
&\nabla_{uu} l(t, y_i(t), \overline{u}_i(t)) \overline{\delta u}_i(t) \\
&+\nabla_{uy} l(t, y_i(t), \overline{u}_i(t)) z_i(t) \\
&+\Big(F_{uu}(t, y_i(t), \overline{u}_i(t))\ \overline{\delta u}_i(t) \Big)^* \lambda_i(t) \\
&+\Big(F_{uy}(t, y_i(t), \overline{u}_i(t))\ z_i(t) \Big)^* \lambda_i(t) \\
&+F_u(t, y_i(t), \overline{u}_i(t))^* \eta_i(t) \quad = \quad \bar{d}_i(t) \quad t \in (T_i, T_{i+1}),
\end{aligned} \tag{10.16c}
$$

$$
\begin{aligned}
&\frac{\partial}{\partial t} z_i(t) + F_y(t, y_i(t), \overline{u}_i(t))\ z_i(t) \\
&+F_u(t, y_i(t), \overline{u}_i(t))\ \overline{\delta u}_i(t) \quad = \quad 0, \quad t \in (T_i, T_{i+1})
\end{aligned} \tag{10.16d}
$$

$$
z_i(T_i) \quad = \quad \begin{cases} 0 & \text{if } i = 0, \\ \overline{\delta y}_i, & \text{else,} \end{cases} \tag{10.16e}
$$

and

$$
\overline{\delta \lambda}_i - \eta_i(T_i) \quad = \quad \bar{c}_i(t), \tag{10.16f}
$$

$$
\overline{\delta y}_{i+1} - z_i(T_{i+1}) \quad = \quad \bar{b}_i(t). \tag{10.16g}
$$

Given $\overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1}, y_i, \lambda_i$, and $\overline{\delta y}_i, \overline{\delta \lambda}_{i+1}$, the system (10.16a)–(10.16e) are the optimality conditions for a linear quadratic optimal control problem in the control and state variables $\overline{\delta u}_i$ and z_i, respectively and the corresponding adjoint variables η_i. More precisely, the system (10.16a)–(10.16e) are the necessary optimality conditions for the following linear quadratic optimal control problem in the control and state variables $\overline{\delta u}_i$ and z_i, respectively:

$$
\begin{aligned}
\text{Minimize} \qquad & \int_{T_i}^{T_{i+1}} \Big(\frac{1}{2} \langle \nabla_{yy} l(t, y_i(t), \overline{u}_i(t)) z_i(t), z_i(t) \rangle \\
&+\langle \nabla_{uy} l(t, y_i(t), \overline{u}_i(t)) z_i(t), \overline{\delta u}_i(t) \rangle \\
&+\frac{1}{2} \langle \nabla_{uu} l(t, y_i(t), \overline{u}_i(t)) \overline{\delta u}_i(t), \overline{\delta u}_i(t) \rangle + \langle \bar{d}_i(t), \overline{\delta u}_i(t) \rangle \\
&+\frac{1}{2} \langle F_{yy}(t, y_i(t), \overline{u}_i(t))(z_i(t), z_i(t)), \lambda_i(t) \rangle \\
&+\langle F_{yu}(t, y_i(t), \overline{u}_i(t))(z_i(t), \overline{\delta u}_i(t)), \lambda_i(t) \rangle \\
&+\frac{1}{2} \langle F_{uu}(t, y_i(t), \overline{u}_i(t))(\overline{\delta u}_i(t), \overline{\delta u}_i(t)), \lambda_i(t) \rangle \ \Big)\ dt \\
&+\langle \overline{\delta \lambda}_{i+1}, z_i(T_{i+1}) \rangle
\end{aligned} \tag{10.17a}
$$

subject to

$$
\begin{aligned}
&\frac{\partial}{\partial t} z_i(t) + F_y(t, y_i(t), \overline{u}_i(t))\ z_i(t) \\
&+F_u(t, y_i(t), \overline{u}_i(t))\ \overline{\delta u}_i(t) \quad = \quad 0, \quad t \in (T_i, T_{i+1})
\end{aligned} \tag{10.17b}
$$

$$
z_i(T_i) \quad = \quad \begin{cases} 0 & \text{if } i = 0, \\ \overline{\delta y}_i, & \text{else,} \end{cases} \tag{10.17c}
$$

For $i = N-1$, the term $\langle \overline{\delta\lambda}_{i+1}, z_i(T_{i+1})\rangle$ in the objective function (10.17a) has to be replaced by $\langle \nabla_{yy}\psi(y_i(T))z_i(T), z_i(T)\rangle$.

10.4.2 Gauss–Seidel Iterations

Let $\mathbf{D}$, $-\mathbf{L}$, $-\mathbf{U}$ be the block diagonal part, the strictly lower block triangular part, and the strictly upper block triangular part of $\mathbf{G}'(\mathbf{x})$, respectively. Thus,

$$\mathbf{G}'(\mathbf{x}) = \mathbf{D} - \mathbf{L} - \mathbf{U}.$$

We will look at three block Gauss–Seidel (GS) iterations (see, e.g., [14], [96], [189], [208]): The forward GS method given by

$$\boldsymbol{\delta}\mathbf{x}_{k+1} = (\mathbf{D} - \mathbf{L})^{-1}(\mathbf{b} + \mathbf{U}\boldsymbol{\delta}\mathbf{x}_k), \tag{10.18}$$

the backward GS method

$$\boldsymbol{\delta}\mathbf{x}_{k+1} = (\mathbf{D} - \mathbf{U})^{-1}(\mathbf{b} + \mathbf{L}\boldsymbol{\delta}\mathbf{x}_k), \tag{10.19}$$

and the forward–backward GS method

$$\begin{aligned} \boldsymbol{\delta}\mathbf{x}_{k+1/2} &= (\mathbf{D} - \mathbf{L})^{-1}(\mathbf{b} + \mathbf{U}\boldsymbol{\delta}\mathbf{x}_k), \\ \boldsymbol{\delta}\mathbf{x}_{k+1} &= (\mathbf{D} - \mathbf{U})^{-1}(\mathbf{b} + \mathbf{L}\boldsymbol{\delta}\mathbf{x}_{k+1/2}), \end{aligned} \tag{10.20}$$

where $\mathbf{b} = -\mathbf{G}(\mathbf{x})$, in the case we apply the GS method directly to the solution of (10.14), or $\mathbf{b} = \mathbf{0}$, if we use the GS method as a preconditioner.

In the following we will concentrate on the description of (10.18). From this (10.19), (10.20), the block Jacobi method, and even the block SOR method can be easily derived. The implementation of the forward block GS method is described in Algorithm (15).

Algorithm 15. [Forward Gauss-Seidel Method]

Computation of $\boldsymbol{\delta}\mathbf{x} \leftarrow (\mathbf{D} - \mathbf{L})^{-1}(\mathbf{b} + \mathbf{U}\boldsymbol{\delta}\mathbf{x})$ with $\mathbf{b} = -\mathbf{G}(\mathbf{x})$ or $\mathbf{b} = \mathbf{0}$.

For $i = 0, \ldots, N-1$ do

1. Compute the solution $y_i = y_i(\cdot; \overline{y}_i, \overline{u}_i)$ of the time subinterval state equation (10.4) (by resolving (10.4) or by retreiving $y_i(\cdot; \overline{y}_i, \overline{u}_i)$ from storage).
2. Compute the solution $\lambda_i = \lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ of the time subinterval adjoint equation (10.8) (by resolving (10.4) or by retreiving $\lambda_i(\cdot; \overline{y}_i, \overline{u}_i, \overline{\lambda}_{i+1})$ from storage).
3. If $\mathbf{b} = -\mathbf{G}(\mathbf{x})$, then evaluate

$$\begin{aligned} \bar{c}_i &= -\overline{\lambda}_i + \lambda_i(T_i), \\ \bar{d}_i(t) &= -\nabla_u l(t, y_i(t), \overline{u}_i(t)) - F_u(t, y_i(t), \overline{u}_i(t))^* \lambda_i(t) \quad t \in (T_i, T_{i+1}), \\ \bar{b}_i &= -\overline{y}_{i+1} + y_i(T_{i+1}), \end{aligned}$$

   else (if $\mathbf{b} = \mathbf{0}$), set

$$\bar{c}_i = 0, \bar{d}_i = 0, \bar{b}_i = 0.$$

   (If $i = 0$, then $\bar{c}_0$ is not needed. If $i = N-1$, then $\bar{b}_{N-1}$ is not needed.)

4. Compute the solution $z_i, \overline{\delta u}_i$ and corresponding adjoint variable η_i of (10.17).

5. Set

$$\begin{aligned} \overline{\delta\lambda}_i &= \bar{c}_i + \eta_i(T_i), && \text{(only if i = 1, . . . , N-1)} \\ \overline{\delta y}_{i+1} &= \bar{b}_i + z_i(T_{i+1}; \overline{y}_i, \overline{u}_i) && \text{(only if i = 0, . . . , N-2).} \end{aligned}$$

10.5 Application to the Optimal Control of Burgers Equation

As an example, we consider an optimal boundary control problem for the one-dimensional Burgers equation. The objective function is least squares type and the controls u_0, u_1 are applied through Robin boundary conditions. The control problem can be stated as follows.

$$\begin{aligned} \text{Minimize} \quad & \frac{\alpha_1}{2}\int_0^T \int_0^1 \|y(t,x) - y_d(t,x)\|^2 dx dt + \frac{\alpha_2}{2}\int_0^1 \|y(T,x) - y_T(x)\|^2 dx \\ & + \frac{1}{2}\int_0^T u_0^2(t) + u_1^2(t) dt \end{aligned} \tag{10.21a}$$

subject to

$$\frac{\partial}{\partial t}y(t,x) - \nu y_{xx}(t,x) + \frac{1}{2}(y^2(t,x))_x = f(t,x) \qquad (t,x) \in (0,T)\times(0,1), \tag{10.21b}$$

$$-\nu y_x(t,0) + \sigma_0 y(t,0) = u_0(t) \qquad t \in (0,T), \tag{10.21c}$$

$$\nu y_x(t,1) + \sigma_1 y(t,1) = u_1(t) \qquad t \in (0,T), \tag{10.21d}$$

$$y(0,x) = y_0(x) \qquad x \in (0,1). \tag{10.21e}$$

Here $\alpha_1, \alpha_2 > 0$, $\nu > 0$, $\sigma_0, \sigma_1 > 0$ are given parameters and $y_d, f \in L^2(0,T,L^2(\Omega))$, $y_0 \in L^2(\Omega)$ are given functions. This control problem is considered in [213]; a corresponding distributed control problem is studied in [119]. We refer to those papers for additional details.

The state equation (10.21b)–(10.21d) are interpreted in the standard weak sense, which is introduced next. Define

$$W(0,T) = \left\{ \phi \in L^2(0,T,H^1(\Omega)) \ : \ \frac{\partial}{\partial t}\phi \in L^2(0,T,H^1(\Omega)') \right\}$$

A function $y \in W(0,T)$ is called a weak solution of the state equation (10.21b)–(10.21d) if $y(0) = y_0$ in $L^2(\Omega)$ and

$$\begin{aligned} & \langle \frac{\partial}{\partial t}y(t), \phi\rangle_{(H^1)',H^1} + \sigma_1 y(t,1)\phi(1) + \sigma_0 y(t,0)\phi(0) \\ & + \int_0^1 \nu y_x(t,x)\phi_x(x) + y(t,x)y_x(t,x)\phi(x) dx \\ & - \int_0^1 f(t,x)\phi(x) dx - u_0(t)\phi(0) - u_1(t)\phi(1) = 0 \end{aligned} \tag{10.22}$$

for all $\phi \in H^1(\Omega)$.

The Lagrangian corresponding to (10.21a), (10.22) is given by

$$\begin{aligned}L(y,u_0,u_1,\lambda) =&\frac{\alpha_1}{2}\int_0^T\int_0^1 \|y(t,x)-y_d(t,x)\|^2dxdt+\frac{\alpha_2}{2}\int_0^1\|y(T,x)-y_T(x)\|^2dx\\ &+\frac{1}{2}\int_0^T u_0^2(t)+u_1^2(t)dt\\ &+\int_0^T\Big(\langle\frac{\partial}{\partial t}y(t),\lambda(t)\rangle_{(H^1)',H^1}+\sigma_1 y(t,1)\lambda(t,1)+\sigma_0 y(t,0)\lambda(t,0)\\ &\quad+\int_0^1 \nu y_x(t,x)\lambda_x(t,x)+y(t,x)y_x(t,x)\lambda(t,x)dx\\ &\quad-\int_0^1 f(t,x)\phi(x)dx-u_0(t)\lambda(t,0)-u_1(t)\lambda(t,1)\Big)dt. \end{aligned}\tag{10.23}$$

Optimality conditions for (10.21a), (10.22) consist of the state equations (10.22) with initial conditions $y(0)=y_0$ in $L^2(\Omega)$, the adjoint equations

$$\begin{aligned}&-\langle\frac{\partial}{\partial t}\lambda(t),\phi\rangle_{(H^1)',H^1}+\sigma_1\lambda(t,1)\phi(1)+\sigma_0\lambda(t,0)\phi(0)\\ &+\int_0^1\nu\lambda_x(t,x)\phi_x(x)+y(t,x)\lambda(t,x)\phi_x(x)+y_x(t,x)\lambda(t,x)\phi(x)dx\\ =&-\alpha_1\int_0^1(y(t,x)-y_d(t,x))\phi(x)dx\end{aligned}\tag{10.24a}$$

for all $\phi\in H^1(\Omega)$ with final conditions

$$\lambda(T)=-\alpha_2(y(T)-y_T)\text{ in }L^2(0,1)\tag{10.24b}$$

and the equations

$$u_0-\lambda(.,0)=0,\quad u_1-\lambda(.,1)=0\quad\text{ in }L^2(0,T).\tag{10.25}$$

Equation (10.24) is equivalent to

$$\begin{aligned}&-\langle\frac{\partial}{\partial t}\lambda(t),\phi\rangle_{(H^1)',H^1}+(\sigma_1+y(t,1))\lambda(t,1)\phi(1)+(\sigma_0-y(t,0))\lambda(t,0)\phi(0)\\ &+\int_0^1\nu\lambda_x(t,x)\phi_x(x)-y(t,x)\lambda_x(t,x)\phi(x)dx\\ =&-\alpha_1\int_0^1(y(t,x)-y_d(t,x))\phi(x)dx\qquad\forall\phi\in H^1(\Omega)\end{aligned}\tag{10.26}$$

and are the weak form corresponding to

$$-\frac{\partial}{\partial t}\lambda(t,x)-\nu\lambda_{xx}(t,x)-y(t,x)\lambda_x(t,x)=-\alpha_1(y(t,x)-y_d(t,x),$$

$$(t,x)\in(0,T)\times(0,1),\tag{10.27a}$$

$$-\nu\lambda_x(t,0)+(\sigma_0-y(t,0))\lambda(t,0)=0,\qquad t\in(0,T),\tag{10.27b}$$

$$\nu\lambda_x(t,1)+(\sigma_1+y(t,1))\lambda(t,1)=0,\qquad t\in(0,T),\tag{10.27c}$$

$$\lambda(T,x)=-\alpha_2(y(T,x)-y_T(x))\qquad x\in(0,1).\tag{10.27d}$$

For the time decomposition of (10.21a), (10.22) we select time intervals $0 = T_0 < T_1 \ldots < T_N = T$, we introduce the auxiliary variables

$$\overline{y}_i \in H \stackrel{\text{def}}{=} L^2(0,1), \quad i = 0, \ldots, N-1,$$

where $\overline{y}_0 \stackrel{\text{def}}{=} y_0$ and we set

$$\overline{u}_{0,i} = u_0|_{(T_i,T_{i+1})}, \overline{u}_{1,i} = u_1|_{(T_i,T_{i+1})} \in U_i \stackrel{\text{def}}{=} L^2(T_i, T_{i+1}) \quad i = 0, \ldots, N-1.$$

The time-domain decomposition of (10.21a), (10.22) is now straightforward. The time subdomain state equation (10.4) corresponds to (10.22) restricted to (T_i, T_{i+1}) with initial conditions $y_i(T_i) = \overline{y}_i$, time subdomain adjoint equation (10.8) corresponds to (10.24a) restricted to (T_i, T_{i+1}) with final conditions $\lambda_i(T_{i+1}) = \overline{\lambda}_{i+1}, i = 1, \ldots, N-2, \lambda_i(T_{i+1}) = -\alpha_2(y(T) - y_T), i = N-1$, and the time subdomain gradient equation (10.9b) corresponds to (10.25) restricted (T_i, T_{i+1}) (since we have two control components, there are two equations). We omitt the explicit statement of (10.4), (10.8) and (10.9b) for this application.

For this application, the objective function is quadratic in the states and control and has no cross-terms. The state equation is linear in the controls and the only nonlinearity in the state is given by the advective term yy_x. Hence for this application the equations (10.11), (10.12), (10.16c) and the local linear quadratic optimal control problem (10.17) simplify: For this application the linearized time subinterval state equation (10.11) is given by

$$\begin{aligned}
&\langle \frac{\partial}{\partial t} z_i(t), \phi \rangle_{(H^1)',H^1} + \sigma_1 z_i(t,1)\phi(1) + \sigma_0 z_i(t,0)\phi(0) \\
&+ \int_0^1 \nu (z_i)_x(t,x)\phi_x(x) + z_i(t,x)(y_i)_x(t,x)\phi(x) + y_i(t,x)(z_i)_x(t,x)\phi(x)dx \\
&- \overline{\delta u}_{0,i}(t)\phi(0) - \overline{\delta u}_{1,i}(t)\phi(1) = 0
\end{aligned} \tag{10.28a}$$

for all $\phi \in H^1(\Omega)$ with initial condition

$$z_i(T_i) = \begin{cases} 0, & \text{if } i = 0, \\ \overline{\delta y}_i, & \text{else} \end{cases} \qquad \text{in } L^2(0,1), \tag{10.28b}$$

and the linearized time subinterval adjoint equation (10.12) becomes

$$\begin{aligned}
&- \langle \frac{\partial}{\partial t} \eta_i(t), \phi \rangle_{(H^1)',H^1} + \sigma_1 \eta_i(t,1)\phi(1) + \sigma_0 \eta_i(t,0)\phi(0) \\
&+ \int_0^1 \nu (\eta_i)_x(t,x)\phi_x(x) + y_i(t,x)\eta_i(t,x)\phi_x(x) + (y_i)_x(t,x)\eta_i(t,x)\phi(x)dx \\
&= - \int_0^1 z_i(t,x)\lambda_i(t,x)\phi_x(x) + (z_i)_x(t,x)\lambda_i(t,x)\phi(x)dx - \alpha_1 \int_0^1 z_i(t,x)\phi(x)dx
\end{aligned} \tag{10.29a}$$

for all $\phi \in H^1(\Omega)$ with final condition

$$\eta_i(T_{i+1}) = \begin{cases} -\alpha_2 z_i(T), & \text{if } i = N-1, \\ \overline{\delta\lambda}_{i+1}, & \text{else} \end{cases} \qquad \text{in } L^2(0,1). \tag{10.29b}$$

Equation (10.16c) corresponds to (recall that we have two control components u_0, u_1 and thus two equations)

$$\overline{\delta u}_{0,i} - \eta_i(.,0) = \bar{d}_{0,i}, \quad \text{in } L^2(T_i, T_{i+1}) \tag{10.30a}$$

$$\overline{\delta u}_{1,i} - \eta_i(.,1) = \bar{d}_{1,i}, \quad \text{in } L^2(T_i, T_{i+1}). \tag{10.30b}$$

Finally, the linear-quadratic optimal control problem (10.17) takes the form

$$\begin{aligned}\text{Minimize} \qquad & \frac{\alpha_1}{2}\int_{T_i}^{T_{i+1}}\int_0^1 z_i^2(t,x)dxdt + \frac{1}{2}\int_{T_i}^{T_{i+1}} \overline{\delta u}_{0,i}^2(t) + \overline{\delta u}_{1,i}^2(t)dt \\ & + \int_{T_i}^{T_{i+1}} \bar{d}_{0,i}(t)\overline{\delta u}_{0,i}(t) + \bar{d}_{1,i}(t)\overline{\delta u}_{1,i}(t)dt \\ & + \int_{T_i}^{T_{i+1}}\int_0^1 z_i(t,x)(z_i)_x(t,x)\lambda_i(t,x)dxdt \\ & + \int_0^1 \overline{\delta\lambda}_{i+1}(x)z_i(T_{i+1},x)dx \end{aligned} \tag{10.31a}$$

subject to

$$\begin{aligned}& \langle\frac{\partial}{\partial t}z_i(t),\phi\rangle_{(H^1)',H^1} + \sigma_1 z_i(t,1)\phi(1) + \sigma_0 z_i(t,0)\phi(0) \\ & + \int_0^1 \nu(z_i)_x(t,x)\phi_x(x) + z_i(t,x)(y_i)_x(t,x)\phi(x) + y_i(t,x)(z_i)_x(t,x)\phi(x)dx \\ & - \overline{\delta u}_{0,i}(t)\phi(0) - \overline{\delta u}_{1,i}(t)\phi(1) = 0 \qquad \forall\phi\in H^1(\Omega)\end{aligned} \tag{10.31b}$$

and the initial condition

$$z_i(T_i) = \begin{cases} 0 & \text{if } i=0, \\ \overline{\delta y}_i, & \text{else,}\end{cases} \qquad \text{in } L^2(0,1). \tag{10.31c}$$

For $i = N-1$, the term $\int_0^1 \overline{\delta\lambda}_{i+1}(x)z_i(T_{i+1},x)dx$ in (10.31a) has to be replaced by $\frac{\alpha_2}{2}\int_0^1 z_i^2(T,x)dx$.

10.6 Application to the Dirichlet Control of the Two-Dimensional Heat Equation

Our second example is the linear quadratic optimal control problem

$$\min \frac{1}{2}\int_0^T \|u^2(t,\cdot)\|^2_{L^2(\Gamma)}dt + \frac{\alpha_1}{2}\int_0^T \|(y(t,\cdot)-z_1(t,\cdot)\|^2_{L^2(\Omega)}dt + \frac{\alpha_2}{2}\|y(T,\cdot)-z_2\|^2_{-1},$$

subject to

$$\begin{aligned} \tfrac{\partial}{\partial t}y(t,x) - \nu\Delta y(t,x) &= f(t,x), && t\in(0,T),\ x\in\Omega, \\ y(t,x) &= u(t,x) && t\in(0,T),\ x\in\Gamma_0, \\ y(t,x) &= 0 && t\in(0,T),\ x\in\partial\Omega\setminus\Gamma_0, \\ y(0,x) &= y_0(x) && x\in\Omega. \end{aligned} \tag{10.32}$$

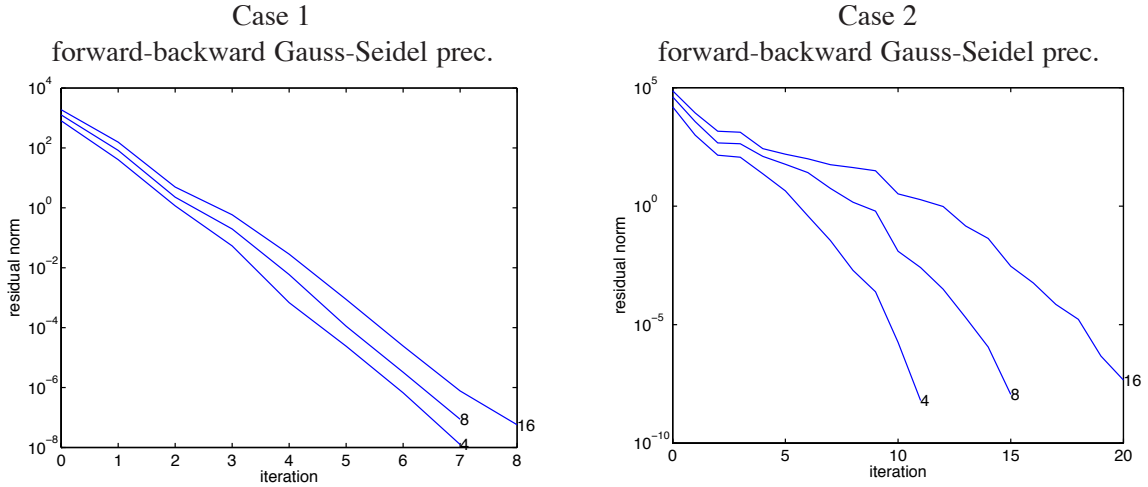
Here $\Omega=(0,1)^2$ and $\|\cdot\|_{-1}$ denotes the norm in $H^{-1}(\Omega)$, which defined by $\|y\|_{-1} = \|\nabla\phi\|_{L^2}$, where $\phi\in H_0^1(\Omega)$ is the solution of $\int_\Omega \nabla\phi\cdot\nabla\theta = \langle y,\theta\rangle_{H^{-1}\times H_0^1}$ for all $\theta\in H_0^1(\Omega)$. The objective function with $\alpha_1 = 0$ is an approximate controllability problem studied in [52], [94, § 2]. For $u\in U = L^2(0,T;L^2(\partial\Omega))$ the problem (10.32) has a unique solution y in $L^2(0,T;L^2(\Omega))\cap C^0([0,T];(H^1(\Omega))^*)$ with y_t in

$L^2(0,T;(H^2(\Omega))^*)$ (see [94, § 2]). Our spatial discretization of the problem follows [52], [94, § 2.6], who use linear finite elements on a uniform triangulation which is constructed by dividing Ω into n_x^2 squares of equal size and then cutting each square from the lower left to the top right into two triangles. In our computations $n_x = 16$. We use the backward Euler in time using $n_t = kN_t = 32$ time steps.

Our problem data are those of the second test problem in [94, p.182]. In particular, $\Gamma_0 = (0,1) \times \{0\}$, $\nu = 1/(2\pi^2)$, $z_1(t,x_1,x_2) = \min\{x_1, x_2, 1-x_1, 1-x_2\}$, $z_2(x_1,x_2) = \min\{x_1, x_2, 1-x_1, 1-x_2\}$, $f = 0$, and $y_0 = 0$. We consider two cases. In case 1 we set $\alpha_1 = \alpha_2 = 10^5$ and in case 2 we set $\alpha_1 = 0, \alpha_2 = 10^5$.

Since, this example problem is linear quadratic, the operator $\mathbf{G}(\mathbf{x})$ is afffine linear, $\mathbf{G}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$ We solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ using GMRES with left preconditioning. The preconditioned GMRES iteration was truncated when the residual norm was less than 10^{-7}. The GMRES iterations with forward–backward GS preconditioning are documented in Figure 10.4. For case 1 we observe at most a slight increase in GMRES iterations, when the number of time–subdomains is increased. For case 2, which is the problem in which only end–time observations are present, and which is significantly more ill–conditioned, the number of GMRES iterations seems to increase roughly linearly with the number of time–subdomains. The total number of preconditioned GMRES iterations, however, still remains reasonably small.

Figure 10.4. Convergence history of GMRES (Example 2, $N_t = 4, 8, 16$)



10.7 Conclusions

We have discussed the application of a time-domain decomposition approach for the solution of parabolic PDE constrained optimal control problems. The time-domain decomposition approach is motivated by the need to cope with the large storage requirements arising out of the strong coupling in time of states, adjoints and controls.

The time decomposition of the original problem leads to a large scale discrete–time optimal control (DTOC) problem in Hilbert space. We have discussed the solution of these DTOCs using a Lagrange-Newton-SQP method. Near a solution of the problem this is equivalent to applying Newton's

method to the first order optimality conditions. The tasks arising in the implementation of the Lagrange-Newton-SQP for the DTOC are related to the solution of the state, linearized state and adjoint equations restricted to time subintervals. The linear system that has to be solved in each iteration of the Lagrange-Newton-SQP method has a block structure, which motivates the application of block Gauss–Seidel (GS) methods for its solution. The forward backward GS method is particularly effective as a preconditioner.

Parallelization of the time-domain decomposition approach is currently under investigation. In particular, the time-domain decomposition approach will be used as an additional layer of parallelization, in addition to existing parallelization of PDE solvers or of the time-subdomain optimal control problems. Potential savings may also result from the need to handle only smaller time subinterval problem. Depending on problem size, this may allow one to reduce the use slow memory and thus result in significant savings of computing time.

Chapter 11

Adjoint-based *a posteriori* error analysis

11.1 Introduction

For many scientific calculations, a numerical solution of a partial differential equation (PDE) is computed to evaluate the actual quantity of interest, a functional of this solution. This quantity of interest could be a weighted average of the solution or the flux of the solution through a boundary. In fluid dynamics the lift and drag of an object are examples functionals of interest. Classical *a posteriori* error estimation focused on estimating the error in a global norm, see [6, 201]. These estimates in global error may not be of much help when we are only interested in the output functional. Moreover these global norms might not have any physical meaning. Adjoint based error estimation techniques for functionals of interest have been developed by a number of people to address these concerns. For a good review of the history and recent developments see [29, 90]. Adjoint based error estimates can be derived for a number of different discretizations of the problem. They have also been used in a couple of ways. These error estimates can be used as a correction to the functional increasing its accuracy, e.g. [89]. The other main use for the adjoint based error estimation is for mesh adaption. Here the goal is to obtain a quasi optimal mesh to solve the PDE on and evaluate the functional of interest. For examples of hp-adaptive schemes for discontinuous Galerkin methods see [122, 105]. Also interesting is the use of the error estimation framework for discontinuous Galerkin in a higher order Godunov finite volume setting, e.g. [21].

Here we present an adjoint based *a posteriori* error analysis for Laplace's equation to get the flavor of this approach. We are going to follow [122] where applicable and simplify matters if possible. We will generate our formulation for the Local Discontinuous Galerkin (LDG) method [62] for solving Laplace's equation. Using this error estimate, different refinement strategies are explored for both h- and p-refinement. Using heuristics described in the paper, we obtain exponential convergence in a solution weighted average functional of a singular solution to Laplace's equation on an L-shaped domain.

Using the ideas learned for the steady state problems presented here, our future work will focus on error estimation in the time-domain where the ultimate goal is to develop error estimates and mesh adaption for transient optimal control problems. The idea would be to use the adjoint solution that is already being computed for optimization and for error estimation. However, error estimates in Galerkin schemes need the adjoint to be computed on a different mesh due to a technical detail called Galerkin orthogonality. This is unfortunate because one would like to compute the adjoint solution on the same mesh as the primal solution. Other, non-Galerkin discretizations such as finite-volume and finite-difference methods, do not

have this problem. One idea to overcome the Galerkin orthogonality is to use reconstructed solutions similar to those used in high-order finite-volume methods [211] and this is an interesting area for future research.

11.2 Formulation: An Example from Linear Algebra

We will first develop a goal-orientated *a posteriori* error estimate for linear systems as in [89]. This same framework will be applied later in the paper to partial differential equation. We are going to explore the linear algebra framework first to help build intuition for adjoint-based error estimates.

Let $A \in \mathbb{R}^{m \times n}$ be a matrix. Let $\boldsymbol{g} \in \mathbb{R}^n$ and $\boldsymbol{f} \in \mathbb{R}^m$ be given vectors. We now define the *primal problem* as

$$\text{Determine } \boldsymbol{g}^T\boldsymbol{u} \text{ where } \boldsymbol{u} \in \mathbb{R}^n \text{ such that } A\boldsymbol{u} = \boldsymbol{f}. \tag{11.1}$$

Note that the goal of the primal problem is to compute the functional of interest $\boldsymbol{g}^T\boldsymbol{u}$ not just to compute $\boldsymbol{u}$. For example, we could be interested in just the ith component of $\boldsymbol{u}$. In this case $\boldsymbol{g}$ would be the vector with all components equal to zero except the ith which would be one. We will define the *dual problem* as

$$\text{Determine } \boldsymbol{v}^T\boldsymbol{f} \text{ where } \boldsymbol{v} \in \mathbb{R}^m \text{ such that } A^T\boldsymbol{v} = \boldsymbol{g}. \tag{11.2}$$

With these problem definitions we have the *primal-dual equivalence*

$$\boldsymbol{v}^T\boldsymbol{f} = \boldsymbol{v}^T(A\boldsymbol{u}) = (A^T\boldsymbol{v})^T\boldsymbol{u} = \boldsymbol{g}^T\boldsymbol{u}.$$

Assume that $\tilde{\boldsymbol{u}}$ is an approximate solution to the primal linear system in (11.1) and $\tilde{\boldsymbol{v}}$ is an approximation to the dual linear system in (11.2). An equation for the error in the primal functional of interest is given as

$$\begin{aligned}
\boldsymbol{g}^T\boldsymbol{u} - \boldsymbol{g}^T\tilde{\boldsymbol{u}} &= \boldsymbol{g}^T(\boldsymbol{u} - \tilde{\boldsymbol{u}}) && \text{by the distributive property of vectors} \\
&= \boldsymbol{v}^T A(\boldsymbol{u} - \tilde{\boldsymbol{u}}) && \text{by the transpose of (11.2)} \\
&= (\boldsymbol{v} - \tilde{\boldsymbol{v}} + \tilde{\boldsymbol{v}})^T A(\boldsymbol{u} - \tilde{\boldsymbol{u}}) && \text{by substituting } \boldsymbol{v} = \boldsymbol{v} + \tilde{\boldsymbol{v}} - \tilde{\boldsymbol{v}} \\
&= \tilde{\boldsymbol{v}}^T A(\boldsymbol{u} - \tilde{\boldsymbol{u}}) + (\boldsymbol{v} - \tilde{\boldsymbol{v}})^T A(\boldsymbol{u} - \tilde{\boldsymbol{u}}) && \text{by the distributive property of vectors} \\
&= \tilde{\boldsymbol{v}}^T(A\boldsymbol{u} - A\tilde{\boldsymbol{u}}) + (\boldsymbol{v} - \tilde{\boldsymbol{v}})^T A(\boldsymbol{u} - \tilde{\boldsymbol{u}}) && \text{by the distributive property of matrices} \\
&= \tilde{\boldsymbol{v}}^T(\boldsymbol{f} - A\tilde{\boldsymbol{u}}) + (\boldsymbol{v} - \tilde{\boldsymbol{v}})^T A(\boldsymbol{u} - \tilde{\boldsymbol{u}}) && \text{by (11.1).}
\end{aligned} \tag{11.3}$$

The first term on the right-hand-side of (11.3) can be computed. However, the second term cannot be computed because it involves $\boldsymbol{v}$, the exact solution of the dual problem (11.2). If bounds of $\boldsymbol{n}\boldsymbol{u} - \tilde{\boldsymbol{u}}$, $\boldsymbol{n}\boldsymbol{v} - \tilde{\boldsymbol{v}}$ and $\boldsymbol{n}A$ exist then Cauchy's inequality can be used to derive a bound of the error in the functional of interest. If we are only interested in an estimate of the error and we assume that our approximate dual solution $\tilde{\boldsymbol{v}}$ is *close enough* to $\boldsymbol{v}$ then

$$\boldsymbol{g}^T\boldsymbol{u} - \boldsymbol{g}^T\tilde{\boldsymbol{u}} \approx \tilde{\boldsymbol{v}}^T(\boldsymbol{f} - A\tilde{\boldsymbol{u}}),$$

can be a reasonable approximation to the error in the functional of interest. Not using Cauchy's inequality provides a potentially sharper approximation of error by giving up the requirement that the estimate bounds the error. In this paper we will follow the same approach to estimating the error in the numerical approximation of output functionals, where the quantity of interest is a functional of a partial differential equation solution.

11.3 Formulation: An Example PDE

Let Ω be a bounded convex polygonal subset of $\mathbb{R}^d$ where d is the dimension of the problem. Let $(\cdot,\cdot)_\Omega$ denote the inner product in $L^2(\Omega)$ and $(\cdot,\cdot)_{\partial\Omega}$ denote the inner product in $L^2(\partial\Omega)$. Let $f, g \in L^2(\Omega)$ and $e, h \in T(\partial\Omega)$. The space $T(\partial\Omega)$ corresponds to all of the functions that are the trace of $H^2(\Omega)$. We consider the following *primal problem*

$$\text{Determine } J(u) = (g,u)_\Omega + (h, \frac{\partial u}{\partial n})_{\partial\Omega} \tag{11.4a}$$

$$\text{where } u \in H^2(\Omega) \text{ is a weak solution of}$$

$$\Delta u = f \quad \text{in } \Omega \tag{11.4b}$$

$$u = e \quad \text{on } \partial\Omega. \tag{11.4c}$$

Here $J(u)$ is the functional of interest that consists of a weighted average of the solution on Ω and a weighted average of the normal derivative of the solution along the boundary. The *dual problem* formulation is

$$\text{Determine } l(v) = (v,f)_\Omega + (\frac{\partial v}{\partial n}, e)_{\partial\Omega} \tag{11.5a}$$

$$\text{where } v \in H^2(\Omega) \text{ is a weak solution of}$$

$$\Delta v = g \quad \text{in } \Omega \tag{11.5b}$$

$$v = h \quad \text{on } \partial\Omega. \tag{11.5c}$$

One thing to note is that Poisson's equation with Dirichlet boundary conditions is self-adjoint. In other words, the adjoint problem is also Poisson's equation with Dirichlet boundary conditions. We will show how to formulate this adjoint problem in the next section.

11.4 Continuous Adjoint Formulation

Suppose that we have the following general PDE *primal problem*

$$\text{Determine } J(u) = (g,u)_\Omega + (h, Cu)_{\partial\Omega} \tag{11.6a}$$

$$\text{where } u \in V \text{ is a weak solution of}$$

$$Lu = f \quad \text{in } \Omega \tag{11.6b}$$

$$Bu = e \quad \text{on } \partial\Omega, \tag{11.6c}$$

where L is a linear PDE operator, B is a boundary operator, C is an algebraic or differential operator, and V is an appropriate Hilbert space for the problem. We assume that f, g, e and h are given functions in the appropriate spaces so that the primal and dual problems have unique solutions. The general *dual problem* is

$$\text{Determine } l(v) = (v,f)_\Omega + (C^* v, e)_{\partial\Omega}$$

$$\text{where } v \in V \text{ is a weak solution of}$$

$$L^* v = g \quad \text{in } \Omega$$

$$B^* v = h \quad \text{on } \partial\Omega,$$

where L^* is the linear PDE operator adjoint to L, B^* is a boundary operator and C^* is an algebraic or differential operator. The primal and dual problems are related by the *primal-dual equivalence*

$$\begin{aligned} l(v) &= (v, f)_\Omega + (C^* v, e)_{\partial\Omega} \\ &= (v, Lu)_\Omega + (C^* v, Bu)_{\partial\Omega} \\ &= (L^* v, u)_\Omega + (B^* v, Cu)_{\partial\Omega} \\ &= (g, u)_\Omega + (h, Cu)_{\partial\Omega} \\ &= J(u). \end{aligned} \tag{11.8}$$

To derive the adjoint formulation given L, B and C we find L^*, B^* and C^* such that the primal dual equivalence (11.8) is satisfied. In [88] the restrictions on J, L, B, and C for there to exist an adjoint formulation of (11.6) are given. They also calculate adjoint equations for inviscid and viscous compressible flow.

We now derive the adjoint equation for our model problem (11.4). We need the following identity to hold for all $w, z \in V$

$$(w, \Delta z)_\Omega + (C^* w, z)_{\partial\Omega} = (L^* w, z)_\Omega + (B^* w, \frac{\partial z}{\partial n}). \tag{11.9}$$

For the model problem, integration by parts gives $\forall v, z \in V$

$$(w, \Delta z)_\Omega = (\Delta w, z)_\Omega + (w, \frac{\partial z}{\partial n})_{\partial\Omega} - (\frac{\partial w}{\partial n}, z)_{\partial\Omega}.$$

If we let $L^* w = \Delta w$, $B^* w = w$ and $C^* w = \frac{\partial w}{\partial n}$ then (11.9) holds. So, the adjoint formulation of the model problem is (11.5).

11.5 Local Discontinuous Galerkin Discretization

Although the error estimate framework is independent of the discretization used to approximate the PDE in (11.4), we will first specify the discretization to make the error estimation more concrete. Let $\mathcal{T}_h = K$ be a triangulation of Ω. Here h is the characteristic size of the elements. Let Γ_h be the union of all edges of the elements of $\mathcal{T}_h$. Let $\boldsymbol{p}_h = \{p_K\}_{K \in \mathcal{T}_h}$ be a degree vector where for each $K \in \mathcal{T}_h$ the polynomial-order of the element is given by $p_K \geq 1$. Define $S^{\boldsymbol{p}_h}(\mathcal{T}_h) := \{u \in L^2(\Omega) \colon u|_K \in \mathcal{P}_{p_K}(K), \forall K \in \mathcal{T}_h\}$ where $\mathcal{P}_{p_K}(K)$ is the space of all polynomials on K of degree at most p_K. Let $V_h := S^{\boldsymbol{p}}(\mathcal{T}_h)$ and $\Sigma_h := S^{\boldsymbol{p}}(\mathcal{T}_h)^d$. Finally, the local derivative is defined as $\nabla_h u|_K := \nabla u \quad \forall K \in \mathcal{T}_h$.

The local discontinuous Galerkin (LDG) [62] bilinear form for the Laplace operator is

$$\begin{aligned} B_{\mathsf{dg}}(u_h, w) = &\int_\Omega \nabla_h \cdot \nabla_h u_h w \, dx - \sum_{K \in \mathcal{T}_h} \int_{\partial K} (\hat{u}_K - u_h) \nabla_h w \cdot \boldsymbol{n}_K \, ds \\ &- \sum_{K \in \mathcal{T}_h} \int_{\partial K} (\nabla_h u_h - \hat{\boldsymbol{\sigma}}_K) \cdot \boldsymbol{n}_K w \, ds, \end{aligned}$$

where $\boldsymbol{n}_K$ is the outward pointing normal to element K. Let K_1 and K_2 be two elements with a common edge where $\boldsymbol{n}_1$ and $\boldsymbol{n}_2$ are outward pointing normals for the elements then for $q \in V_h$ with $q_i := q|_{\partial K_i}$ and

$\boldsymbol{\phi} \in \Sigma_h$ with $\boldsymbol{\phi}_i := \boldsymbol{\phi}|_{\partial K_i}$ we have

$$\{\!\{q\}\!\} = \frac{1}{2}(q_1 + q_2), \qquad [\![q]\!] = q_1 \boldsymbol{n}_1 + q_2 \boldsymbol{n}_2,$$
$$\{\!\{\boldsymbol{\phi}\}\!\} = \frac{1}{2}(\boldsymbol{\phi}_1 + \boldsymbol{\phi}_2), \qquad [\![\boldsymbol{\phi}]\!] = \boldsymbol{\phi}_1 \cdot \boldsymbol{n}_1 + \boldsymbol{\phi}_2 \cdot \boldsymbol{n}_2 \,.$$

Also σ_h is defined by

$$\int_\Omega \boldsymbol{\sigma}_h \cdot \boldsymbol{\tau} \, dx = -\int_\Omega u_h \nabla \cdot \boldsymbol{\tau} \, dx + \sum_{K \in \mathcal{T}_h} \int_{\partial K} \hat{u}_K \boldsymbol{n}_K \cdot \boldsymbol{\tau} \, ds \quad \forall \boldsymbol{\tau} \in \Sigma_h.$$

The numerical fluxes $\hat{u}_K$ and $\hat{\boldsymbol{\sigma}}_K$ are defined as

$$\hat{u}_K = \{\!\{u_h\}\!\} - \boldsymbol{\beta} \cdot [\![u_h]\!] \text{ on } \Gamma_h \setminus \partial\Omega, \qquad \hat{u}_K = 0 \text{ on } \partial\Omega,$$
$$\hat{\boldsymbol{\sigma}}_K = \{\!\{\boldsymbol{\sigma}_h\}\!\} + \boldsymbol{\beta}[\![\boldsymbol{\sigma}_h]\!] - \alpha([\![u_h]\!]) \text{ on } \Gamma_h \setminus \partial\Omega, \qquad \hat{\boldsymbol{\sigma}}_K = \boldsymbol{\sigma}_h - \alpha(u_h \boldsymbol{n}) \text{ on } \partial\Omega.$$

Here $\boldsymbol{\beta} \in \mathbb{R}^d$ and α is on the order of p^2/h where h is the characteristic size of the mesh and p is the maximum polynomial-order on the mesh.

For error estimation, we require both state and adjoint consistency. State consistent means that if u is a solution to the primal problem (11.4) then

$$B_{\mathsf{dg}}(u, z) = l(z) \quad \forall z \in H^2(\mathcal{T}_h). \tag{11.12}$$

Adjoint consistent means that if v is a solution to the dual problem (11.5) then

$$B_{\mathsf{dg}}(w, v) = J(w) \quad \forall w \in H^2(\mathcal{T}_h). \tag{11.13}$$

It turns out that LDG is both state and adjoint consistent see [12] for details as well as other numerical fluxes that are state and adjoint consistent.

11.6 Computation of Error Estimates

Following a similar path as [122], we define global and local error estimates to the error in the functional $J(u) - J(u_h)$ where u_h is the solution of (11.14b). We have the following *discrete primal problem*

$$\text{Determine } \; J(u_h) = (g, u_h)_\Omega + (h, \frac{\partial u_h}{\partial n})_{\partial\Omega} \tag{11.14a}$$
$$\text{where } \; u_h \in V_h \text{ such that}$$
$$B_{\mathsf{dg}}(u_h, w) = l(w) \quad \forall w \in V_h. \tag{11.14b}$$

We also have the *discrete dual problem*

$$\text{Determine } \; l(v_H) = (v_H, f)_\Omega + (C^* v_H, e)_{\partial\Omega} \tag{11.15a}$$
$$\text{where } \; v_H \in V_H \text{ such that}$$
$$B_{\mathsf{dg}}(v_H, z) = J(z) \quad \forall z \in V_H, \tag{11.15b}$$

where $V_H = S^{\boldsymbol{p}_H}(\mathcal{T}_H)$. Here p_H and $\mathcal{T}_H$ are chosen such that V_H is not a subspace of V_h. Combining (11.12) and (11.14) we have *Galerkin orthogonality*

$$B_{\mathsf{dg}}(u - u_h, w) = 0 \quad \forall w \in V_h. \tag{11.16}$$

Using (11.12), (11.13), (11.14) and (11.15a) we have the following estimate of the error in the functional of interest

$$\begin{aligned} J(u) - J(u_h) &= B_{\mathsf{dg}}(u, v) - B_{\mathsf{dg}}(u_h, v) \\ &= B_{\mathsf{dg}}(u, v_H) - B_{\mathsf{dg}}(u_h, v_H) + B_{\mathsf{dg}}(u - u_h, v - v_H) \\ &= l(v_H) - B_{\mathsf{dg}}(u_h, v_H) + B_{\mathsf{dg}}(u - u_h, v - v_H) \\ &\approx l(v_H) - B_{\mathsf{dg}}(u_h, v_H) = \sum_{K \in \mathcal{T}_h} \eta_K, \end{aligned} \tag{11.17}$$

where

$$\begin{aligned} \eta_K = &\int_K r_{h,p} v_H \, dx + \int_{\partial K \setminus \partial\Omega} \mu_{h,p} \nabla_h v_H \cdot \boldsymbol{n}_K \, ds \\ &+ \int_{\partial K} \boldsymbol{\nu}_{h,p} \cdot \boldsymbol{n}_K v_H \, ds + \int_{\partial K \cap \partial\Omega} \xi_{h,p} \nabla_h v_H \cdot \boldsymbol{n}_K \, ds. \end{aligned}$$

Here $r_{h,p}|_K = (f - \Delta u_h)|_K$ is the interior residual, $\xi_{h,p}|_{\partial K \cap \partial\Omega} = (g - u_h)|_{\partial K \cap \partial\Omega}$ is the boundary residual, $\mu_{h,p}|_{\partial K \setminus \partial\Omega} = (\hat{u}_K - u_h)|_{\partial K \setminus \partial\Omega}$ is the flux residual, and $\boldsymbol{\nu}_{h,p}|_{\partial K \setminus \partial\Omega} = (\nabla u_h - \hat{\boldsymbol{\sigma}}_K)|_{\partial K \setminus \partial\Omega}$ is the gradient flux residual. Also, η_K is the local error estimate on element K.

Note that if $V_H \subseteq V_h$ then from Galerkin orthogonality we would always have $\sum_{K \in \mathcal{T}_h} \eta_K = 0$, which would not be of much use as an error estimate. Another consideration is that we are going to choose to adapt an element K based on the size of $|\eta_K|$ which will disregard any cancellation in the local error estimates that exists in $\sum_{K \in \mathcal{T}_h} \eta_K$.

11.6.1 hp-adaption

Local error estimates can be used to decide which elements are responsible for the largest errors. There are two options for improving the solution on a given element. We can increase the polynomial order of the element — called p-refinement. On the other hand we can divide the element into multiple elements — called h-refinement. If the solution is sufficiently *smooth*, p-refinement will be most beneficial. The idea is to start with a coarse mesh and use multiple levels of h- and p-refinement to obtain a quasi-optimal mesh to solve (11.14).

We have implemented the following mesh adaptation algorithm. For the ith level of refinement we are given a mesh $\mathcal{T}_{h_i}$ and order vector p_{h_i}. We then solve the primal problem (11.14b) obtaining $u_{h_i} \in V_{h_i}$ and create a new order vector, $p_{H_i} = p_{h_i} + a$, where a is a positive integer added to each component of p_{h_i} to get p_{H_i} to use in the adjoint solve. We let $\mathcal{T}_{H_i} = \mathcal{T}_{h_i}$ and solve the adjoint problem (11.15b) in the space V_{H_i} obtaining v_{H_i}. We then project u_{h_i} in V_{H_i} to calculate the residual R_{H_i}. To get the error indicator on element $K \in \mathcal{T}_{h_i}$ and we compute $\eta_K = (R_{H_i}, v_{H_i})_K$. Next, one of two refinement strategies (see below) are used to mark the elements requiring refining where a smoothness indicator is used to decide between h and p refinement. If the solution on element K is not smooth, h refinement is chosen and element K is split into four elements. If the solution on K is smooth, p refinement is selected and polynomial-order is increased on element K by one. After this has been performed for each element that marked for refinement, a new mesh $\mathcal{T}_{h_{i+1}}$ and order vector $p_{h_{i+1}}$ are available and the processes can be repeated.

11.6.2 Refinement Strategies

Global TOL Let N_i be the total number of elements in $\mathcal{T}_i$. For a given TOL we select for refinement the elements whose absolute value of the error indicator, $|\eta_K|$, is greater than TOL$/N_i$. We stop refining when $\sum_{K \in \mathcal{T}_h} \eta_K <$ TOL. For this scheme is it also possible to use coarsening. With a given coarsening fraction, c_f, we select for coarsening the elements whose absolute value of the error indicator, $|\eta_K|$, is such that $|\eta_K| < c_f(\text{TOL}/N_i)$. Currently we only coarsen in p by decreasing the polynomial order on the elements that are selected for coarsening.

Top Fraction For this refinement strategy we are given a refinement percentage, r_p. For each refinement level we refine r_p as the percentage of the elements with the greatest error indicator. We do this for a given number of refinements. For this scheme it is also possible to use coarsening. For each refinement level we coarsen c_p percent of the elements with the smallest error indicators. Currently we only coarsen in p by decreasing the polynomial order on the elements that are selected for coarsening.

11.6.3 Smoothness Indicator

To calculate the smoothness of the appropriated solution we do something similar to what Mavriplis [158] proposed. For a given element K we expand our solution in the Proriol–Koornwinder–Dubiner–Owens [138] orthonormal basis. We assume $a_i \sim C \exp -\sigma i$ where a_i is the magnitude of the expansion coefficients for the polynomials of order i. We find C and σ by solving a least squares problem. If $\sigma > 1$ we say the solution is smooth in that element and otherwise we say it is not smooth. One thing to note about the smoothness indicator is that it works well when p is greater than 4 or so.

11.7 Implementation

We have implemented adjoint-based error estimates in Sandia's modal discontinuous Galerkin framework called SAGE [65] and a nonconforming nodal discontinuous Galerkin finite element framework called SLEDGE [70, 115]. Results have been obtained and compared using both implementations for p-adaptation and (as expected the results are identical). For brevity, we only report results using SLEDGE here, since the nonconforming capability of SLEDGE allows easy h and p refinement. SLEDGE uses nodal elements where the $N = \frac{1}{2}(p+1)(p+2)$ nodes $\{x_i\}$ for polynomial interpolation of order p are given in [114]. Note that the nodes along the edge are exactly the Legendre-Gauss-Lobatto points. On element K, we use the following representation of the solution

$$q_N^K(\boldsymbol{x}) = \sum_{i=0}^{N} u_i^K L_i(\boldsymbol{x}),$$

where $L_i(x_j) = \delta_{i,j}$ are the Lagrange polynomials for the nodal set $\{x_i\}$. This error estimate capability, developed in both SAGE and SLEDGE, are considered preliminary prototyping for Sandia production codes, such as codes in the SIERRA framework.

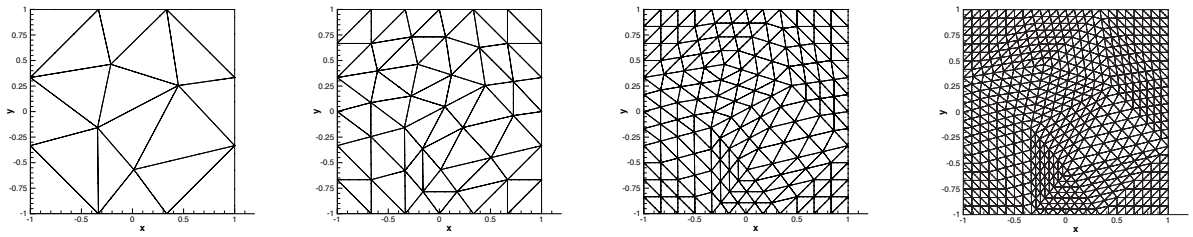


Figure 11.1. Element meshes for the average functional test and the Gaussian weighted average functional test.

11.8 Numerical Results

11.8.1 Average Functional

In this example, there were a total of four h-refinement levels. At each level the next grid is computed by splitting each element into four elements. We are solving (11.14) on $\Omega = [-1, 1] \times [-1, 1]$ with f set so that $u = \sin(\pi x)\sin(\pi y)$. For our computations the functional of interest is $J(w) = \int_\Omega \omega w \, dx$ where $\omega(x, y) = 1$. The adjoint and fine residual error estimate are calculated at a different order, $+a$, on the same $\mathcal{T}_h$ that is used for the coarse mesh. The meshes this test uses are given in Figure 11.1. A contour plot of the primal and dual solutions can be found in Figure 11.2. We have presented the numerical results in Table 11.1. Note that the error in the functional does not decrease when going from even to odd orders. For this reason, to obtain reliable error estimates one needs to take the fine mesh two-orders greater than the coarse mesh. We also note that we see that the convergence rate is order $p + 1$ when we compare the L^2 error to the square root of the degrees of freedom, as expected [53].

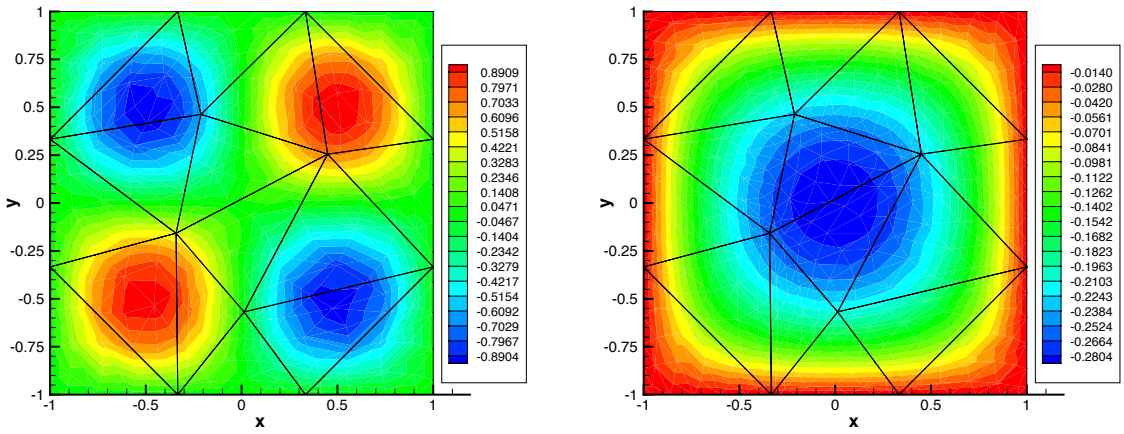
11.8.2 Gaussian Weighted Average Functional

This model problem is the same as the average functional test except for the functional $J(w) = \int_\Omega \omega w \, dx$ where $\omega(x, y) = \exp(-(x + 0.5)^2 - (y + 0.5)^2)$. The meshes used for each refinement level are shown in Figure 11.1 and contour plots of the solutions of the primal and dual problems are in Figure 11.3. The detailed converenge and error estimates are documented in Table 11.2. Looking at the efficiency index, we are still doing a fairly good job of estimating the error. Things are not quite as good as in the average test because this dual problem is more difficult to solve accurately due to the higher spatial gradients.

11.8.3 Boundary Flux Functional

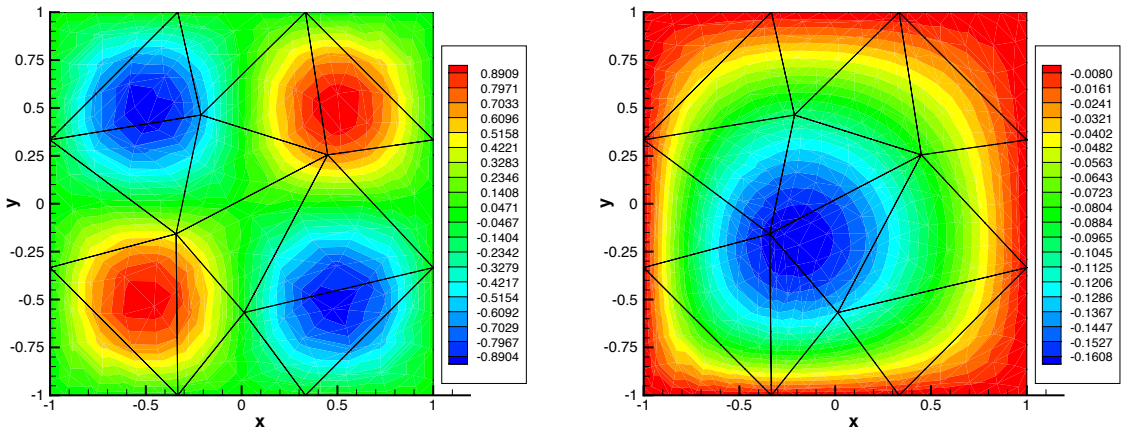
We are solving (11.14) on $\Omega = [-1, 1] \times [-1, 1] \setminus [-0.75, -0.25] \times [-0.75, -0.25]$ with f set so that $u = \sin(\pi x)\sin(\pi y)$. In this test our functional of interest is

$$J(w) = \int_{\partial\Omega} \kappa \frac{\partial w}{\partial n} \, ds$$



(a) (b)

Figure 11.2. Contour plots of the primal and dual solutions for the average functional test case on the initial mesh. (a) Primal solution for $p = 5$ and $u = \sin(\pi x)\sin(\pi y)$. (b) Dual solution $p = 7$ and $J(w) = \int_\Omega w \, dx$



(a) (b)

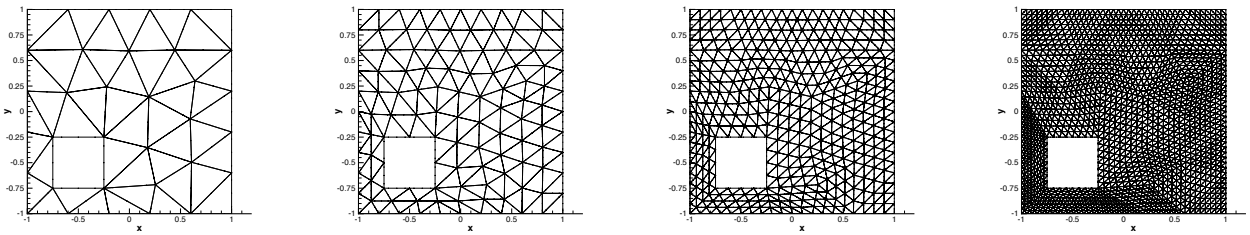
Figure 11.3. Contour plots of the primal and dual solutions for the Gaussian weighted average functional test case on the initial mesh. (a) Primal solution for $p = 5$ and $u = \sin(\pi x)\sin(\pi y)$. (b) Dual solution for $p = 7$ and $J(w) = \int_\Omega \exp\left(-(x+\frac{1}{2})^2 - (y+\frac{1}{2})^2\right) w \, dx$.

Table 11.1. The error, $|J(u) - J(u_h)|$, error estimates, $|\sum_{K\in\mathcal{T}_h} \eta_K|$, and the efficiency index $\theta = (\sum_{K\in\mathcal{T}_h} \eta_K)/J(u - u_h)$ for the average functional test where the primal solution is computed with interpolation order p and the adjoint solution is computed with interpolation order $p + a$. N is the number of elements using the meshes shown in Figure 11.1.

			$a = 1$		$a = 2$		
p	N	$\lvert J(u) - J(u_h)\rvert$	$\lvert\sum_{K\in\mathcal{T}_h} \eta_K\rvert$	θ	$\lvert\sum_{K\in\mathcal{T}_h} \eta_K\rvert$	θ	L^2 error
1	18	1.17385e+00	1.31386e+00	1.11927e+00	1.19681e+00	1.01956e+00	5.59881e-01
	72	2.24101e-01	2.26449e-01	1.01048e+00	2.25277e-01	1.00525e+00	1.52940e-01
	288	5.63805e-02	5.65033e-02	1.00218e+00	5.64515e-02	1.00126e+00	4.32824e-02
	1152	1.41491e-02	1.41565e-02	1.00052e+00	1.41535e-02	1.00031e+00	1.12708e-02
2	18	1.40010e-01	1.17053e-01	8.36032e-01	1.41490e-01	1.01057e+00	8.48875e-02
	72	2.34857e-03	1.17197e-03	4.99012e-01	2.34596e-03	9.98885e-01	1.16150e-02
	288	1.22798e-04	5.17498e-05	4.21422e-01	1.22735e-04	9.99489e-01	1.69646e-03
	1152	7.40629e-06	3.00180e-06	4.05304e-01	7.40524e-06	9.99859e-01	2.23675e-04
3	18	2.29571e-02	2.44371e-02	1.06447e+00	2.36433e-02	1.02989e+00	2.62144e-02
	72	1.17661e-03	1.17399e-03	9.97775e-01	1.18422e-03	1.00647e+00	1.38084e-03
	288	7.10483e-05	7.09855e-05	9.99116e-01	7.11627e-05	1.00161e+00	8.95763e-05
	1152	4.40449e-06	4.40344e-06	9.99762e-01	4.40626e-06	1.00040e+00	5.56033e-06
4	18	1.47994e-03	7.93746e-04	5.36336e-01	1.49315e-03	1.00892e+00	1.91294e-03
	72	2.61811e-06	1.02310e-05	3.90777e+00	2.66171e-06	1.01665e+00	1.57893e-04
	288	6.28102e-08	1.77121e-07	2.81994e+00	6.29982e-08	1.00300e+00	5.01434e-06
	1152	1.04785e-09	2.81794e-09	2.68927e+00	1.04867e-09	1.00072e+00	1.56622e-07
5	18	6.86194e-04	6.99401e-04	1.01925e+00	6.91325e-04	1.00748e+00	1.07330e-03
	72	7.61285e-06	7.56925e-06	9.94272e-01	7.62702e-06	1.00186e+00	1.36188e-05
	288	1.14311e-07	1.14123e-07	9.98354e-01	1.14364e-07	1.00047e+00	2.30722e-07
	1152	1.77009e-09	1.76931e-09	9.99557e-01	1.77002e-09	9.99950e-01	3.68012e-09

Table 11.2. The error, $|J(u) - J(u_h)|$, error estimates, $|\sum_{K\in\mathcal{T}_h} \eta_K|$, and the efficiency index $\theta = (\sum_{K\in\mathcal{T}_h} \eta_K)/J(u - u_h)$ for the Gaussian weighted average functional test where the primal solution is computed with interpolation order p and the adjoint solution is computed with interpolation order $p + a$. N is the number of elements using the meshes shown in Figure 11.1.

			$a = 1$		$a = 2$		
p	N	$\lvert J(u) - J(u_h)\rvert$	$\lvert\sum_{K\in\mathcal{T}_h} \eta_K\rvert$	θ	$\lvert\sum_{K\in\mathcal{T}_h} \eta_K\rvert$	θ	L^2 error
1	18	4.33713e-01	5.30148e-01	1.22235e+00	4.77241e-01	1.10036e+00	5.59881e-01
	72	6.76363e-02	7.93109e-02	1.17261e+00	7.75451e-02	1.14650e+00	1.52940e-01
	288	1.51120e-02	1.79162e-02	1.18556e+00	1.78124e-02	1.17869e+00	4.32824e-02
	1152	3.63399e-03	4.33925e-03	1.19407e+00	4.33278e-03	1.19229e+00	1.12708e-02
2	18	6.64777e-02	5.46756e-02	8.22466e-01	6.73927e-02	1.01376e+00	8.48875e-02
	72	2.44187e-03	1.74262e-03	7.13640e-01	2.28208e-03	9.34561e-01	1.16150e-02
	288	1.45373e-04	1.03109e-04	7.09275e-01	1.34235e-04	9.23383e-01	1.69646e-03
	1152	9.07516e-06	6.45938e-06	7.11765e-01	8.36297e-06	9.21523e-01	2.23675e-04
3	18	1.19640e-02	1.27152e-02	1.06279e+00	1.23587e-02	1.03300e+00	2.62144e-02
	72	5.20624e-04	5.39692e-04	1.03662e+00	5.40232e-04	1.03766e+00	1.38084e-03
	288	2.99345e-05	3.11268e-05	1.03983e+00	3.11375e-05	1.04019e+00	8.95763e-05
	1152	1.82849e-06	1.90360e-06	1.04107e+00	1.90376e-06	1.04116e+00	5.56033e-06
4	18	7.04011e-04	3.56134e-04	5.05864e-01	7.01378e-04	9.96260e-01	1.91294e-03
	72	3.61131e-06	5.39076e-07	1.49274e-01	3.03010e-06	8.39059e-01	1.57893e-04
	288	5.12753e-08	1.07044e-08	2.08763e-01	4.15313e-08	8.09967e-01	5.01434e-06
	1152	7.96075e-10	1.61912e-10	2.03387e-01	6.41207e-10	8.05454e-01	1.56622e-07
5	18	3.38714e-04	3.45245e-04	1.01928e+00	3.42217e-04	1.01034e+00	1.07330e-03
	72	3.53008e-06	3.56917e-06	1.01108e+00	3.58349e-06	1.01513e+00	1.36188e-05
	288	5.14128e-08	5.22356e-08	1.01600e+00	5.22964e-08	1.01719e+00	2.30722e-07
	1152	7.89335e-10	8.03144e-10	1.01750e+00	8.03219e-10	1.01767e+00	3.68012e-09



(a) (b) (c) (d)

Figure 11.4. Meshes for the boundary flux functional test where all of the elements are refined between each level. (a) starting mesh, (b) refinement level 1, (c) refinement level 2, (d) refinement level 3.

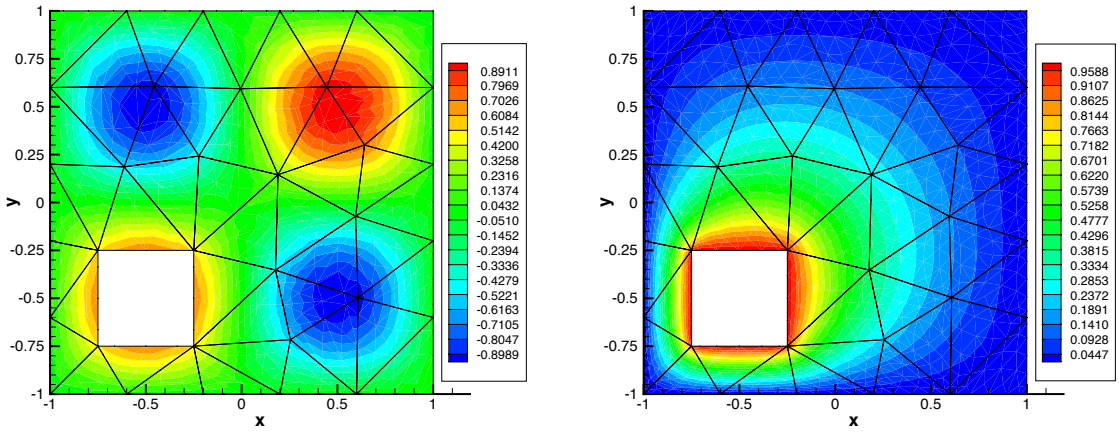
Table 11.3. The error, $|J(u) - J(u_h)|$, error estimates, $|\sum_{K\in\mathcal{T}_h} \eta_K|$, and the efficiency index $\theta = (\sum_{K\in\mathcal{T}_h} \eta_K)/J(u-u_h)$ for the boundary flux functional test where the primal solution is computed with interpolation order p and the adjoint solution is computed with interpolation order $p+a$. N is the number of elements in the simulation.

			$a=2$		
p	N	$\lvert J(u)-J(u_h)\rvert$	$\lvert\sum_{K\in\mathcal{T}_h}\eta_K\rvert$	θ	L^2 error
4	46	2.00761e-02	1.64589e-02	8.19826e-01	2.51347e-04
	184	1.03799e-03	8.39250e-04	8.08533e-01	8.51998e-06
	736	5.95473e-05	4.71253e-05	7.91393e-01	2.67050e-07
	2944	3.49455e-06	2.72934e-06	7.81027e-01	8.32271e-09

where $\kappa(x,y) = 1$ on the interior hole boundary, see Figure 11.4, and $\kappa(x,y) = 0$ elsewhere. The meshes this test uses are given in Figure 11.4 for the global refinement all elements are refined at each stage. A contour plot of the primal and dual solution can be found in Figure 11.5. We have presented the numerical results in Table 11.3. The efficiency index is around 0.8 for all of these cases which is likely due to inaccuracy of the dual solution which likely requires a finer h-mesh. In some situations it might be necessary to adapt both the dual and primal mesh to get accurate error estimates.

11.8.4 L-Shaped Domain

In this model problem, (11.14) is solved on an L-shaped domain as shown in Figure 11.6. The boundary conditions are set so that the exact solution is given, in polar coordinates, as $u(r,\theta) = r^{2/3}\sin(2\pi\theta/3)$. This problem has been studied by many researchers in the context of error estimation and mesh adaptation, see, e.g. [79]. In our simulations shown in Figure 11.6, we have used the global TOL refinement strategy with TOL $= 10^{-7}$ and $c_f = 0.1$. This produces a final mesh that has h-adaption near the derivative-singularity at $r = 0$, and p-adaption elsewhere. Figure 11.7 shows the error in the functional, $J(w) = \int_\Omega uw\ dx$, as a function of the square root of the degrees of freedom. We see that for just h-adaption the best choice is $p = 1$ and higher-order interpolation does not help and can actually hurt the performance. However, for hp-refinement strategies exponential convergence in the functional is observed.



(a) (b)

Figure 11.5. Contour plots of the primal and dual solutions for the boundary flux functional test case on the initial mesh. (a) Primal solution for $p = 4$ and $u = \sin(\pi x)\sin(\pi y)$. (b) Dual solution $p = 6$ and $J(w) = \int_{\partial\Omega} \kappa \frac{\partial w}{\partial n}\, ds$ where $\kappa(x, y) = 1$ on the interior hole boundary.

11.9 Conclusion

We have implemented an adjoint based error estimator for the local discontinuous Galerkin discretization of Poisson's problem. Here the error is measured in some functional of the computed solution. This functional can be a weighted integral of the solution in the domain or a weighted integral of the flux of the solution across a boundary. A smoothness indicator has also been written that is based on the convergence of the coefficients of the projection of the solution on the orthonormal basis for a triangle. Using the error estimates and the smoothness indicator we have obtained exponential decrease in the error of a weighted integral of a singular solution of Laplace's equation on an L-shaped domain.

For future work we would like to apply adjoint based error estimates to time dependent problems, non-linear problems, other discretizations and then to real applications such as the compressible fluid flow problems. This error estimate work developed here is also considered preliminary prototyping for Sandia production codes, such as codes in the SIERRA framework.

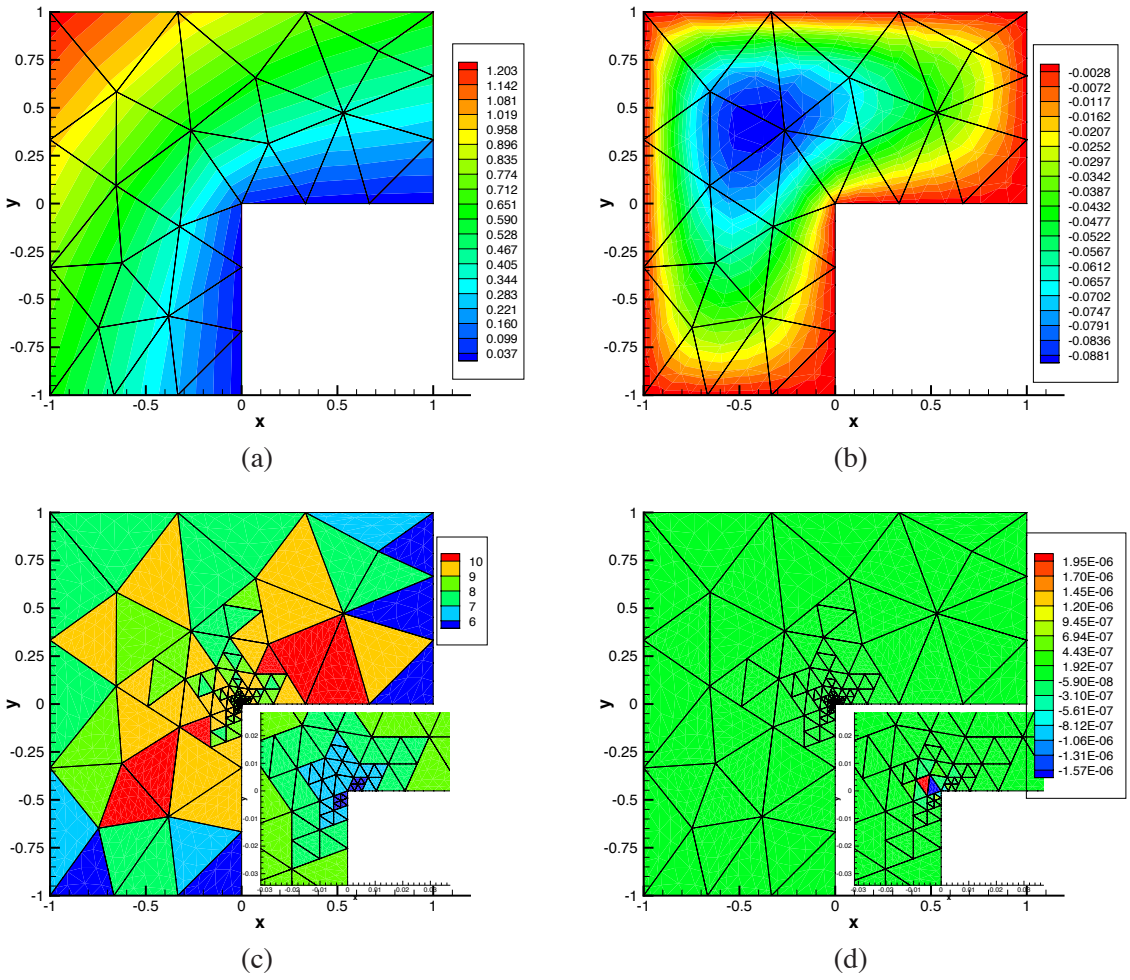


Figure 11.6. (a) Contours of the primal solution on the staring mesh defined in polar coordinates as $u(r,\theta) = r^{2/3}\sin(\frac{2\pi}{3}\theta)$. (b) Contours of the dual solution for $J(w) = \int_\Omega uw \; dx$. (c) The final mesh where different colors represent the polynomial-order of the element $p+1$. We have used the global TOL refinement strategy with TOL $= 10^{-7}$ and $c_f = 0.1$. (d) Here is a contour plot of the error estimate η_K. Most of the error is concentrated at the corner.

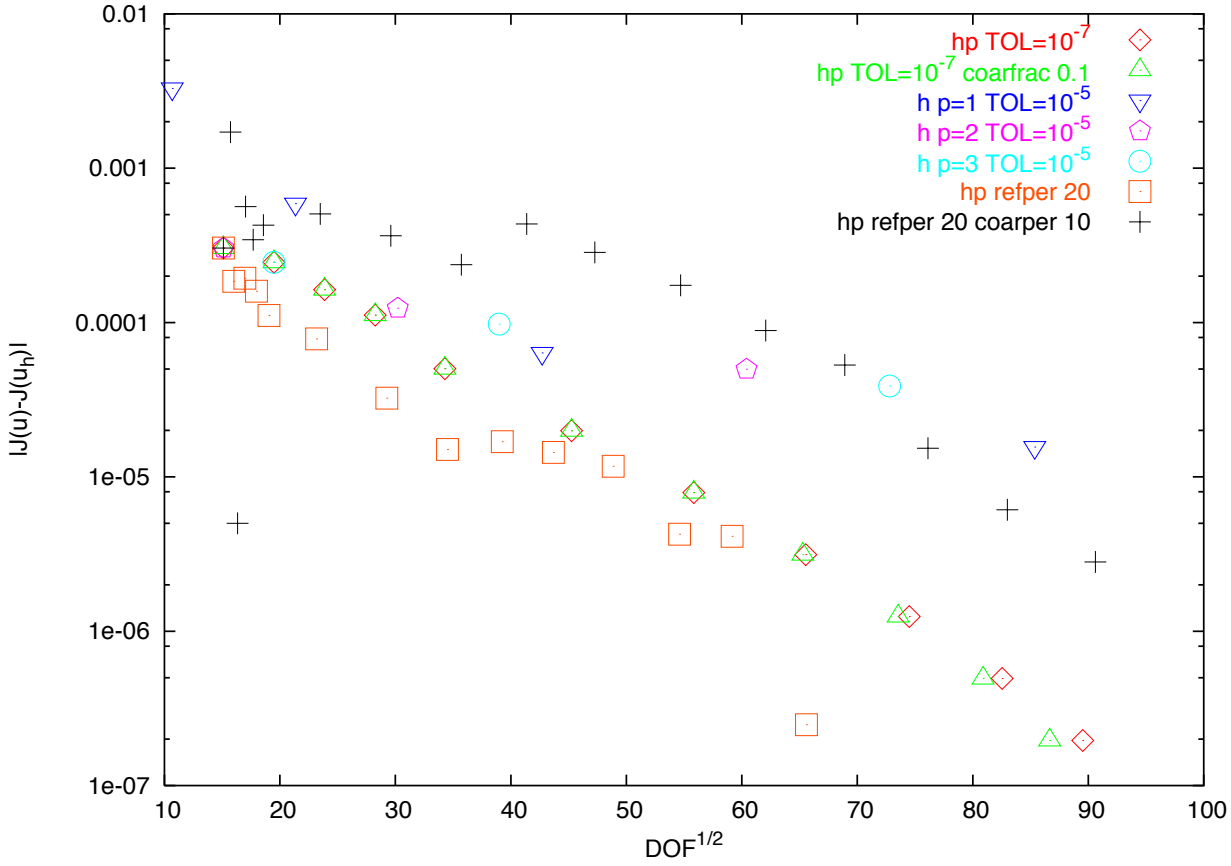


Figure 11.7. The error in the functional, $J(w) = \int_\Omega uw \; dx$ as a function of the square root of the degrees of freedom. We see that for just h-adaption the best choice is $p = 1$ and higher order interpolation does not help — it actually hurts the performance. We also see that for both refinement strategies we get exponential convergence in the functional of interest. If we put a straight-line fit through ◇ it has a slope of 2.8.

Chapter 12

Continuous versus Discrete Adjoints

In Section 1.1 the relative advantages and disadvantages of continuous and discrete adjoint formulations are outlined. The preceding Chapter 11 on adjoint-based error estimation used a continuous adjoint approach to develop *a posteriori* error estimates for functionals of the solution. A continuous adjoint formulation was natural in this context, since the theory of *a posteriori* error estimation formally requires the *exact* adjoint solution for a given functional. Obviously, this is not practical, and one typically solves the adjoint problem either on a finer mesh or using a higher-order reconstruction and both approaches have shown success. In the case of Galerkin methods (such as finite-elements and discontinuous Galerkin), the direct use of the discrete adjoint is not possible due to Galerkin orthogonality, although one can use the discrete adjoint computed on a finer mesh[1]. The purpose of the current section is to provide a brief comparison of discrete and continuous adjoint formulations for a simple model problem that illustrates the important issues.

12.1 Formulation

As a prototypical example, consider the transient one-dimensional heat-equation

$$y_{,t} - (\nu y_{,x})_{,x} = 0 \tag{12.1a}$$

in $\Omega = (t, x) \in (0, T] \times [0, 1]$ with initial condition

$$y(0, x) = -\sin\left(\pi \frac{\tan\left(m(2x-1)\right)}{\tan(m)}\right) \qquad x \in [0, 1] \tag{12.1b}$$

where $m = 1.3$. We specify Robin boundary conditions

$$-\nu y_{,x} + y = 0 \qquad \text{at } x = 0, \quad t \in (0, T] \tag{12.1c}$$

$$\nu y_{,x} + y = u \qquad \text{at } x = 1, \quad t \in (0, T] \tag{12.1d}$$

where u is the control variable. The objective function is

$$J(u, y) = \int_0^1 \left(\frac{\alpha}{2}u^2 + \frac{\alpha_t}{2}u_{,t}^2\right)\, dt + \frac{\beta}{2}\int_0^1 \left(y(x,1) - y(x,0)\right)^2\, dx\,. \tag{12.2}$$

[1] This is what is actually reported in Section 11.8

For all the results shown here, $\nu = 0.01$, $\alpha = 1$ and $\beta = 1000$. When using time-regularization $\alpha_t = 0.1$ and otherwise, $\alpha_t = 0$. The optimization problem is to minimize $J(\phi, u)$ for all $u \in \mathcal{U}$ subject to (12.1a) where $\mathcal{U}$ is the space of admissible controls.

Equation (12.1a) is discretized using the (local) discontinuous Galerkin (DG) method (see Section 11.5) in space with backward Euler time-integration. For consistency, the time integral in (12.2) is also evaluated using backward Euler and the spatial integral is computed consistent with the DG element representation.

As described in Section 11.5, the LDG method is adjoint consistent meaning that the LDG discretization of the continuous adjoint is exactly the same as the discrete adjoint of the original LDG discretized state. In general, Galerkin discretizations can have this correspondence although care must be taking with DG methods in formulating the numerical fluxes to ensure adjoint consistency. Likewise, we use the adjoint of backward Euler to integrate the adjoint equation backward in time.

In order to illustrate the differences between discrete and continuous adjoints in this context, we take as our reference the adjoint consistent LDG method with adjoint backward Euler time-integration and refer to this as a *discrete adjoint* (DA) method when both the state and adjoint are computed using the same LDG discretization and same time-step.

While the discrete adjoint is unique for a given state discretization, there are many ways to construct a continuous adjoint approximation and we consider two such approaches here. Our code (i.e. SAGE) implements the adjoint as a discretization of the adjoint PDE. In the case of LDG, the continuous adjoint and the discrete adjoint are identical as long as the mesh and polynomial-order used for each are the same. Thus, our first *continuous adjoint* (CA) method is to force the adjoint solver to use a different polynomial-order from that used in the state. In the examples shown here, we set the polynomial-order of the adjoint to be one less than that of the state and this is referred to as the CA($p-1$) method.

A second *continuous adjoint* formulation is obtained by altering the means by which boundary conditions are set. In general, there are many ways that boundary conditions can be weakly enforced within DG methods. One approach (described in Section 11.5) is to replace the numerical fluxes at the boundary by the prescribed boundary fluxes and this results in the adjoint consistent LDG method. Another approach is to prescribe the boundary conditions indirectly through the arguments of the numerical fluxes. Doing so for both the State and Adjoint equations results in a continuous adjoint formulation, i.e., the discretized adjoint equation is no-longer identical to the adjoint of the discretized state. However, both the discretized state and adjoint equations are consistent (and spectrally accurate) discretizations of the original PDE's. We denote this second approach as the CA(BC) method. Of course, there are numerous other ways that a continuous adjoint can be constructed including using a different h-mesh, different time-steps, different time-integration methods, etc.

12.2 Results

All the optimization results discussed here use the Polak-Ribiere variant of the nonlinear conjugate gradient (nCG) algorithm with a secant-type (function value only) line-search. While this combination results in a rather simplistic approach to solving optimization problems, it is appropriate, especially in the presence of inexact gradient information obtained from a continuous adjoint and is a method that has been used successfully in the past for a number of large-scale, transient optimization problems governed by strongly nonlinear physics [35, 56, 55, 64, 58].

Figure 12.1 shows the convergence history of both J and $||\nabla J||$ with and without the time-regularization term. We focus first on the results without time-regularization (i.e., $\alpha_t = 0$) as shown on the left frame of Figure 12.1. Both continuous-adjoint methods lead to a saturation of $||\nabla J||$ after 6 iterations with only 1-2 orders-of-magnitude decrease. This saturation results from inaccuracies in the gradient due to truncation error in the continuous adjoint. Once, saturation of J and $||\nabla J||$ occurs, further iterations are of questionable value. For this linear-quadratic optimization problem, the line-search globalization still results in improved solutions, albeit at a slow convergence rate. However, our experience with large-scale nonlinear optimization problems indicates that the line-search will eventually fail when the continuous adjoint based gradient is dominated by truncation error [55]. Conversely, the discrete adjoint delivers 5 decades of decrease in $||\nabla J||$ in only 10 iterations of nCG with no indication of saturation.

However, returning to Figure 12.1, we see that all three adjoint-methods lead to very similar values of J suggesting that, from a practical standpoint, 6 iterations of any of the methods may be sufficient for engineering purposes. This is verified in Figure 12.2 which shows the final states and controls. The final state profiles are nearly identical for each method although the state computed using the discrete adjoint is slightly closer to the target solution. More difference is observed in the control profiles (right frame of Figure 12.2. While the controls from both continuous-adjoint methods are similar, there is a distinct difference when compared to the converged result obtained using the discrete adjoint. In particular, u_t is greater for the continuous adjoint at late times, and there is a sudden change in slope near the end of the time-interval that is not observed in the discrete adjoint solution. If a larger number of nCG iterations are used (around 18) with the continuous adjoint methods, they do eventually result in profiles that are similar to that obtained using the discrete adjoint. This slow convergence is due to the fact that changes in the control near the end of the time-interval have only a small influence on the final state. Thus, the inaccuracy of the continuous-adjoint based gradient and the lack of sensitivity of the state to changes in the control at late-times leads to slow convergence (an possibly no convergence in nonlinear problems).

12.3 Time Regularization

The fact that the control profiles tend to have large temporal gradients under near-optimal conditions, suggests that the optimization problem is not properly regularized (or equivalently that the space of admissible controls is too large). For this reason, we now add a small amount of temporal-regularization by setting $\alpha_t = 0.1$ in (12.2). Figure 12.1(right) compares the nCG convergence histories using time-regularization with the CA(BC) formulation (denoted as CA(TR)) with the original DA and CA(BC) results. The addition of time-regularization significantly improves the convergence when using a continuous adjoint and this behavior is observed in both J and $||\nabla J||$. Of course, any increase in regularization will tend to improve convergence, however, setting $alpha = 10$ with $\alpha_t = 0$ leads to negligible improvement in convergence when using the continuous adjoint while even a small amount of α_t improves convergence significantly (for both continuous and discrete adjoints).

The state and control using time-regularization are shown in Figure 12.3. The addition of time-regularization leads to a slightly larger difference between the final state and the target state, although, qualitatively the agreement is still good. More importantly, the control profile is now much smoother in time (in fact it is well approximated by a simple quadratic function). The addition of time-regularization prevents the jump in the control at $t = 0$ and also prevents the tendency of the control to develop high gradients near $t = 1$. In fact, the control profile after only 3 iterations is nearly indistinguishable from the final converged result. Similar benefits of higher regularization (both in time and space) have also been

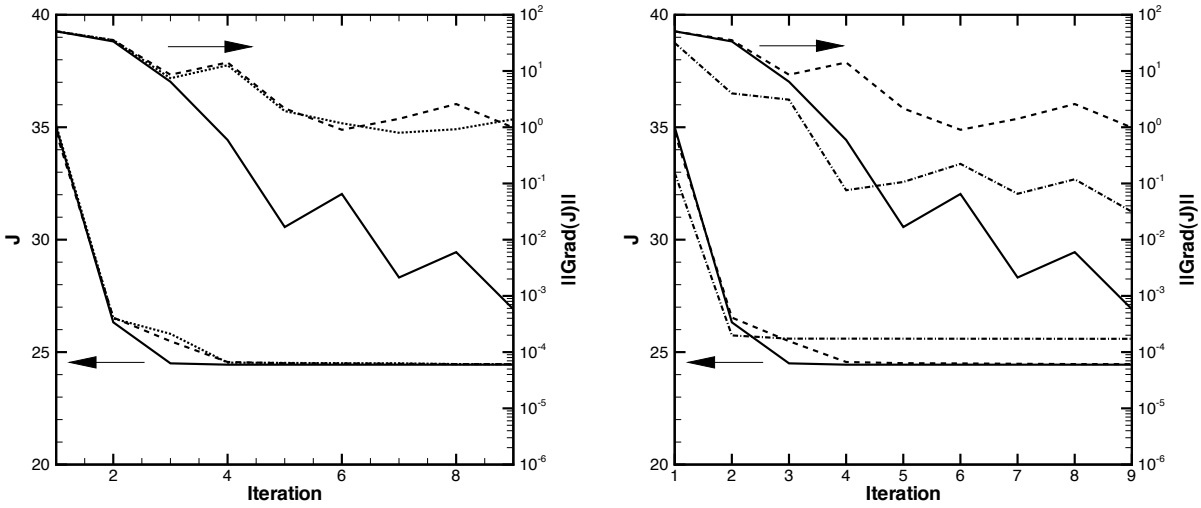


Figure 12.1. Convergence of nonlinear CG for Robin boundary control of the heat equation with and without time-regularization. (Left) without time-regularization: ——— DA, - - - - CA(BC), —·— CA($p-1$). (Right) with time-regularization $\alpha_t = 0.1$: ——— DA, - - - - CA(BC), —·— CA(TR).

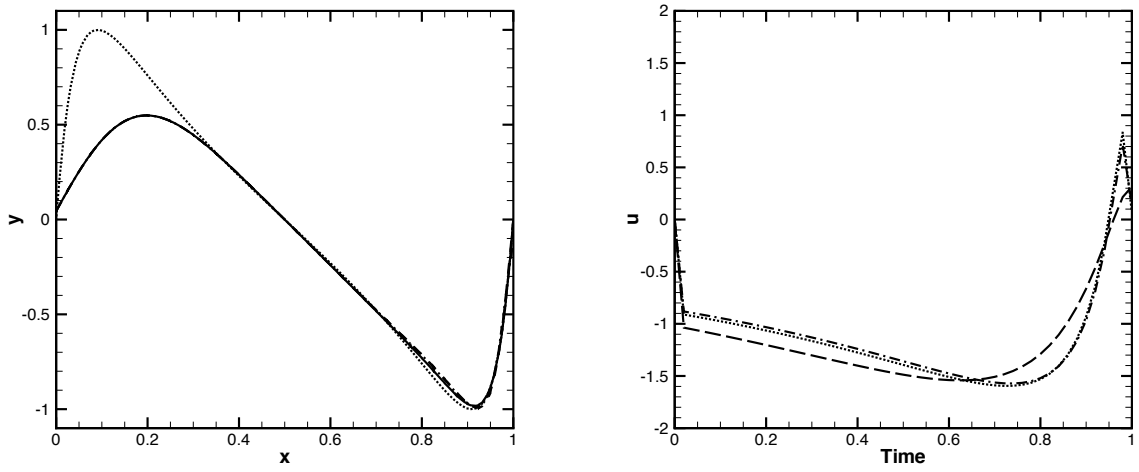


Figure 12.2. Comparison of states and controls for Robin boundary control of the heat equation. (Left) State: ····· Target, ——— DA, - - - - CA(BC), —·— CA($p-1$). (Right) Control: ——— DA, - - - - CA(BC), —·— CA($p-1$)

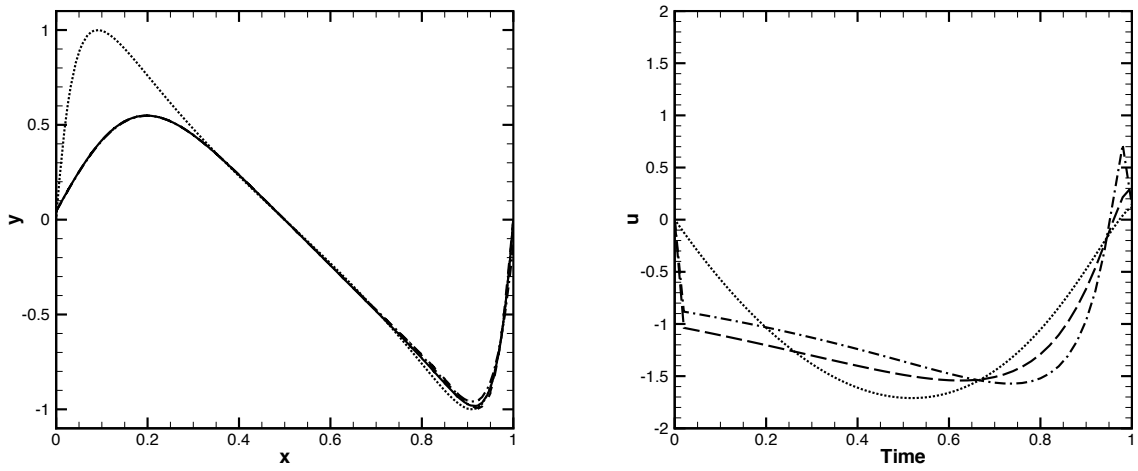


Figure 12.3. Effect of time-regularization ($\alpha_t = 0.1$) on Robin boundary control for the heat equation (results after 6 iterations). (Right) State: ····· Target, ——— DA, - - - - CA(BC), —·— CA(TR). (Left) Control: ——— DA, - - - - CA(BC), —·— CA(TR).

documented for fully nonlinear, compressible Navier–Stokes problems [64, 66] when using continuous adjoints.

12.4 Summary

We have performed a comparison of discrete and continuous adjoints for use in optimization (in this case an optimal boundary control problem). The main conclusion of this study, and extensive prior experience with adjoint methods, is that both discrete and continuous adjoints can be successfully used for large-scale optimization problems. However, there are important issues associated with each choice:

1. A discrete adjoint requires reverse-mode AD or the transpose of the exact Jacobian, neither of which may be available in production simulation codes.
2. There is no unique continuous adjoint – there are many choices that can be made that may or may not affect the optimization solution process.
3. A continuous adjoint offers tremendous flexibility (a pro and a con) in formulating the discretization of the adjoint. In particular, one can (formally) use non-smooth methods (e.g. numerical fluxes and limiter) for both the state and adjoint.
4. The discretization of the continuous adjoint can leverage much of the existing code associated with the forward state solver.
5. A continuous adjoint will (likely) destroy symmetry properties of the optimality conditions that may be required by some optimization algorithms.

6. Optimization algorithms using a continuous adjoint may have degraded convergence or may not converge at all.... However, in practice, a properly regularized optimization problem can yield useful solutions with a continuous adjoint formulation.

We close, by emphasizing that while a discrete adjoint and the resulting "exact" gradient can be convenient for optimization, it is possible that all that one obtains is a very good solution to the *wrong problem*. In the end, we want our methods to produce optimal solutions that are good approximations to the solution of the original continuous PDE-based optimization problem. If there are problems and resolutions for which a continuous adjoint[2] does not lead to useful solutions, then one should be equally cautious about solutions obtained using a discrete adjoint. While one may obtain a converged optimal solution using a discrete adjoint, it can be no better than the underlying accuracy of the original state solution and the failure of the continuous adjoint approach may indicate that numerical resolution is insufficient or that the problem is not properly formulated (i.e. higher regularity of the controls may be warranted).

[2] a consistent discretization of the continuous adjoint PDE that has similar accuracy as the state solver

References

[1] *Methods and applications of interval analysis*. SIAM.

[2] F. Abergel and R. Temam. On some control problems in fluid mechanics. *Theoretical and Computational Fluid Dynamics*, 1:303–325, 1990.

[3] David Abrahams. Generic programming techniques.

[4] Slimane Adjerid, Karen D. Devine, Joseph E. Flaherty, and Lilia Krivodonova. A posteriori error estimation for discontinuous Galerkin solutions of hyperbolic problems. *Comp. Meth. in Appl. Mech. Eng.*, 191(11–12):1097–112, 2002.

[5] Experimental Data Base for Computer Program Assessment. Technical Report 138, AGARD Advisory Report, 1979.

[6] M. Ainsworth and J.T. Oden. A posteriori error estimation in finite element analysis. *Comp. Meth. in Appl. Mech. Eng.*, 142(1–2):1–88, 1997.

[7] V. Akcelik, G. Biros, O. Ghattas, K. R. Long, and B. van Bloemen Waanders. A variational finite element method for source inversion for convective-diffusive transport. *Finite Elements in Analysis and Design. The International Journal of Applied Finite Elements and Computer Aided Engineering*, 39(8):683–705, 2003.

[8] G.A. Newman D.L. Alumbaugh. 3-d electric magnetic inversion using conjugate gradients. Technical Report SAND97-1296C, Sandia National Laboratories, 1997.

[9] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1995.

[10] W. K. Anderson and D. L. Bonhaus. An Implicit Algorithm for Computing Turbulent Flows on Unstructured Grids. *Computer and Fluids*, 23(1):1–21, 1994.

[11] P. Antognetti and G. Massobrio. *Semiconductor Device Modeling with SPICE*. McGraw-Hill, 1988.

[12] D.N. Arnold, F. Brezzi, B. Cockburn, and L.D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SJNA*, 39(5):1749–79, 2001.

[13] U. M. Ascher, R. M. M. Mattheij, and R. D. Russel. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. Classics in Applied Mathematics, Vol. 13. SIAM, Philadelphia, 1995.

[14] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, London, New York, 1994.

[15] Trilinos/aztecoo: Object-oriented aztec linear solver package. http://software.sandia.gov/Trilinos/doc /aztecoo/doc/html/index.html.

[16] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc, portable extensible toolkit for scientific computing (web site). http://www.mcs.anl.gov/petsc.

[17] W. Bangerth and R. Rannacher. Finite element approximation of the acoustic wave equation: error control and mesh adaptation. *East-West J. Numer. Math.*, 7(4):263–282, 1999.

[18] J. Barhen, J. G. Berryman, L. Borcea, J. Dennis, C. de Groot-Hedlin, F. Gilbert, P. Gill, M. Heinkenschloss, L. Johnson, T. McEvilly, J. Moré, G. Newman, D. Oldenburg, P. Parker, B. Porto, M. Sen, V. Torczon, D. Vasco, and N. B. Woodward. Optimization and geophysical inverse problems. Technical Report LBNL–46959, Ernest Orlando Lawrence Berkeley National Laboratories, Earth Science Division, 2000.

[19] T. Barth. The Design and Application of Upwind Schemes on Unstructured Meshes. *AIAA Paper 89-0366*, January 1989.

[20] T. Barth. A 3-D upwind Euler solver for unstructured meshes. *AIAA Paper 91-1548-CP*, 1991.

[21] Timothy J. Barth and Mats G. Larson. A posteriori error estimates for higher order Godunov finite volume methods on unstructured meshes. Technical Report NAS-02-001, NASA Ames Research Center, February 2002.

[22] R. A. Bartlett. *Object Oriented Approaches to Large Scale NonLinear Programming For Process Systems Engineering*. Ph.D Thesis, Chemical Engineering Department, Carnegi Mellon University, Pittsburgh, 2001.

[23] R. A. Bartlett. *MOOCHO : Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide*. Sandia National Labs, 2003.

[24] R. A. Bartlett. TSFCore::Nonlin : An extension of TSFCore for the development of nonlinear abstract numerical algorithms and interfacing to nonlinear applications. Technical report SAND03-xxxx, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2003.

[25] R. A. Bartlett, M. A. Heroux, and K. R. Long. Tsfcore : A package of light-weight object-oriented abstractions for the development of abstract numerical algorithms and interfacing to linear algebra libraries and applications. Technical report SAND03-xxxx, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2003.

[26] Roscoe A. Bartlett. *Object Oriented Approaches to Large-Scale Nonlinear Programming for Process Systems Engineering*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, 2001.

[27] Achim Basserman. Private communication. 2002.

[28] R. Becker, C. Johnson, and R. Rannacher. Adaptive error control for multigrid finite element methods. *Computing*, 55(4):271–88, 1995.

[29] Roland Becker and Rolf Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.

[30] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic di erentiation for the machine besm. Technical report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, 1959.

[31] Steve Benson, Lois Curfman McInnes, and Jorge Moré. TAO : Toolkit for advanced optimization (web page).

[32] M. Berggren. Discrete adjoint equations and gradients for edge-based discretizations of the euler equations. Technical report, Sandia National Laboratories, 2003. Unpublished.

[33] M. Berggren and M. Heinkenschloss. Parallel solution of optimal-control problems by time-domain decomposition. In M.-O. Bristeau, G. Etgen, W. Fitzgibbon, J. L. Lions, J. Periaux, and M. F. Wheeler, editors, *Computational Science for the 21st Century*, pages 102–112, Chichester, 1997. J. Wiley.

[34] J. T. Betts. *Practical Methods for Optimal Control using Nonlinear Programming*. Advances in Design and Control. SIAM, Philadelphia, 2001.

[35] Thomas R. Bewley, Parviz Moin, and Roger Teman. DNS-based predictive control of turbulence: an optimal target for feedback algorithms. *J. Fluid Mech.*, 447:179–225, 2001.

[36] G. Biros and O. Ghattas. Parallel Lagrange–Newton–Krylov–Schur methods for PDE–constrained optimization. part I: The Krylov–Schur solver. Technical report, Computational Mechanics Lab, Department of Civil and Environmental Engineering, Carnegie Mellon University, 2000. http://www.cs.cmu.edu/∼oghattas.

[37] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. Adifor - generating derivative codes from fortran programs. *SCientific Programming*, 1992.

[38] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[39] L. S. Blackford, J. Choi, A. Cleary, E.D. 'Azevedo, J. Demmel, I. Dhilon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walder, and R.C. Whaley. *ScalLAPACK User's Guide*. SIAM, Philadelphia, PA, 1997.

[40] J. Blazek. *Computational Fluid Dynamics: Principles and Applications*. Elsevier, 2001.

[41] H. G. Bock. Randwertprobleme zur Parameteridentifizierung in Systemen nichtlinearer Differentialgleichungen. Preprint Nr. 442, Universität Heidelberg, Institut für Angewandte Mathematik, SFB 123, D–6900 Heidelberg, Germany, 1988.

[42] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[43] A. N. Brooks and T. J. R. Hughes. Streamline upwind/Petrov–Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier–Stokes equations. *Comp. Meth. Appl. Mech. Engng.*, 32:199–259, 1982.

[44] P.N. Brown and Y. Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM Journal of Statistical Computing*, 11:450–481, 1990.

[45] R. Bulirsch. Die Mehrzielmethode zur numerischen Lösung von nichtlinearen Randwertproblems und Aufgaben der optimalen Steuerung. Technical report, Report of the Carl-Cranz Gesellschaft, 1971.

[46] R. H. Byrd, J. Nocedal, and R.B. Schnabel. Representations of quasi-newton matrices and their use in limited methods. *Math. Prog.*, 63:129–156, 1994.

[47] George D. Byrne and Allan C. Hindmarsh. PVODE, an ODE solver for parallel computers. *Int. J. High Perf. Comput. Applic*, 13:354–365, 1999.

[48] D. G. Cacuci. *Sensitivity and Uncertainty Analysis. Volume 1: Theory*. CRC Press, Boca Raton, 2003.

[49] Xing Cai. Two object-oriented approaches to the parallelism of diffpack, 1999.

[50] Y. Cao, S. Li, and L. Petzold. Adjoint sensitivity analysis for differential-algebraic equations, part I: The adjoint DAE systems. Technical report, Computational Sciences and Engineering, University of California, Santa Barbara, 2000.

[51] Y. Cao, S. Li, and L. Petzold. Adjoint sensitivity analysis for differential-algebraic equations, part II: Numerical solution. Technical report, Computational Sciences and Engineering, University of California, Santa Barbara, 2000.

[52] C. Cartel, R. Glowinski, and J.-L. Lions. On the exact and approximate boundary controllabilities for the heat equation: a numerical approach. *Journal of Optimization Theory and Applications*, 82:429–484, 1994.

[53] Paul Castillo and Bernardo Cockburn. An a priori error analysis of the local discontinuous Galerkin method for elliptic problems. *SJNA*, 38(5):1676–1706, 2000.

[54] N.D. Cesare and O. Pironneau. Flow control problem using automatic differentiation in c++. Technical report, Unversite Pierre et Marie Curie, LAN-UPMC report 99013, 2000.

[55] Yong Chang. *Reduced Order Methods for Optimal Control of Turbulence*. PhD thesis, Rice University, Mechanical Engineering and Materials Science, 2000.

[56] Yong Chang and S. Scott Collis. Active control of turbulent channel flows based on large eddy simulation. *ASME Paper No. FEDSM-99-6929*, 1999.

[57] C. J. Chen and W. S. Feng. Relaxation-based transient sensitivity computations for mosfet circuit. *IEEE Trans. Computer-Aided Design*, 1995.

[58] Guoquan Chen and S. Scott Collis. Toward optimal control of aeroacoustic flows using discontinuous Galerkin discretizations. *AIAA paper 2004-0364*, January 2004.

[59] Leon O. Chua and Pen-Min Lin. *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[60] Robert L. Clay, Kyran D. Mish, Ivan J. Otero, Lee M. Taylor, and Alan B. Williams. An annotated reference guide to the finite-element interface (FEI) specification : Version 1.0. Technical Report SAND99-8229, Sandia National Laboratories, 1999.

[61] Clay, R.L., B.A. Allan, L.D. Mish, A.B. Williams. ISIS++ reference guide (iterative scalable implicit solver in C++) version 1.1. Technical Report SAND99-8231, Sandia National Laboratories, 1999.

[62] Bernardo Cockburn and Chi-Wang Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SJNA*, 35(6):2440–2463, 1998.

[63] S. S. Collis, K. Ghayour, M. Heinkenschloss, M. Ulbrich, and S. Ulbrich. Optimal control of unsteady compressible viscous flows. *International Journal for Numerical Methods in Fluids*, 40(11):1401–1429, 2002.

[64] S. Scott Collis, K. Ghayour, M. Heinkenschloss, M. Ulbrich, and S. Ulbrich. Towards adjoint-based methods for aeroacoustic control. *AIAA Paper 2001-0821*, 2001.

[65] S. Scott Collis and Kaveh Ghayour. Discontinuous Galerkin for compressible DNS. *ASME paper number FEDSM2003-45632*, 2003.

[66] S. Scott Collis, Kaveh Ghayour, Matthias Heinkenschloss, Michael Ulbrich, and Stefan Ulbrich. Optimal control of unsteady compressible viscous flows. *Inter. J. Num. Meth. Fluids*, 40:1401–1429, 2002.

[67] D. N. Daescu and G. R. Carmichael. An adjoint sensitivity method for the adaptive location of the observations in air quality modeling. *journal of Atmospheric Sciences*, 2003.

[68] S.N. Daescu and I.M. Navon. An analysis of a hybrid optimization method for variational data assimilation. *International Journal of Computational Fluid Dynamics*, 2003.

[69] Davinci user's manual. Technical report, Avanti! Corporation TCAD Business Unit, 1998.

[70] D.M. Day, R.S. Patil, and T.C. Warburton. Discontinuous Galerkin descretizations applied to eigenvalue problems on non-conforming meshes. Technical Report SAND2004-0000J, Sandia National Laboratory, PO Box 5800, Albuquerque, NM 87185-5800, 2004. In progress.

[71] James Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[72] J. E. Dennis and M. Heinkenschloss. Trust-region interior-point sqp algorithms for a class of nonlienar programming problems. Technical Report TR94-45, Department of Computational and Applied Mathematics, Rice University, 1996.

[73] P. Deuflhard. Nonlinear equation solvers in boundary value problem codes. In B. Childs, editor, *Codes for the BVPs in ODEs*, pages 40–66. Springer Lecture Notes in Computer Science, Vol. 74, 1979.

[74] S. W. Director and R. A. Rohrer. The generalized adjoint network and network sensitivities. *IEEE Tran. Circuit Theory*, 1969.

[75] H. K. Dirks and K. Eickhoff. Numerical models and table models for mos circuit analysis. *Proc. NASECODE IV*, pages 13–24, June 1985.

[76] H. C. Edwards. SIERRA framework version 3: Core services theory and design. Technical report SAND2002-3616, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2002.

[77] H. C. Edwards and J. R. Stewart. SIERRA : A Software Environment for Developing Complex Multiphysics Applications. In K. J. Bathe, editor, *Proceedings of First MIT Conference on Computational Fluid and Solid Mechanics*. Elsevier Scientific, 2001.

[78] M. J. Eppstein, F. Fedele, J. Laible, C. Zhang, A. Godavarty, and E. M. Sevick-Muraca. A comparison of exact and approximate adjoint sensitivities in fluorescence tomography. *IEEE TRANSACTIONS ON MEDICAL IMAGING*, 2003.

[79] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Press Syndicate of the University of Cambridge, 1996.

[80] Donald J. Estep, Mats G. Larson, and Roy D. Williams. *Estimating the Error of Numerical Solutions of Systems of Reaction-Diffusion Equations*, volume 146 of *Memoirs of the American Mathematical Society*. Americal Mathematical Society, 2000.

[81] H. O Fattorini. *Infinite dimensional optimization and optimal control*. Encyclopedia of Mathematics, Vol. 62. Cambridge University Press, Cambridge, New York, 1999.

[82] P. Feldmann, T. V. Nguyen, S. W. Director, and R. A. Rohrer. Sensitivity computation in piecewise approximate circuit simulation. *IEEE Trans. Computer-Aided Design*, 1991.

[83] P. A. Forsyth and H. Jiang. Nonlinear Iteration Methods for High Speed Laminar Compressible Navier-Stokes Equations. *Computers and Fluids*, 26(3):249–268, 1997.

[84] M. Fowler and K. Scott. *UML Distilled, second edition*. Addison-Wesley, 2000.

[85] G.A.Baker. Error estimates for finite element methods for second order hyperbolic equations. *SIAM J. Numer. Anal.*, 13(4):564–576, 1976.

[86] E. Gamma et al. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[87] Micheal Gertz and Stephen Wright. Object oriented software for quadratic programming. `http://www.cs.wisc.edu/~swright/ooqp/`, 2001.

[88] M.B. Giles and N.A. Pierce. Adjoint equations in CFD: Duality, boundary conditions and solution behaviour. *AIAA*, (Paper 97-1850), 1997.

[89] M.B. Giles and N.A. Pierce. Adjoint error correction for integral outputs. In T. Barth and H. Deconinck, editors, *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, volume 25 of *Lecture Notes in Computational Science and Engineering*, pages 47–96. Springer-Verlag, 2002.

[90] Michael B. Giles and Endre Süli. Adjoint methods for PDEs: A posteriori error analysis and postprocessing by duality. *Acta Numerica*, pages 145–236, 2002.

[91] P. E. Gill, L. O. Jay, M. W. Leonard, L. Petzold, and V. Sharma. An SQP method for the optimal control of large–scale dynamical systems. *J. Comp Appl. Math.*, 20:197–213, 2000.

[92] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, Orlando, San Diego, New-York,. . ., 1981.

[93] S. Glasstone and M.C. Edlund. The elements of nuclear reactor theory. Technical report, Van Nostrand, Princeton, New Jersey, 1952.

[94] R. Glowinski and J.-L. Lions. Exact and approximate controllability for distributed parameter systems. In A. Iserles, editor, *Acta Numerica 1995*, pages 159–333. Cambridge University Press, Cambridge, London, New York, 1995.

[95] M.S. Gockenbach and W.W. Symes. The hilbert class library. http://www.trip.caam.rice.edu/txt/hcldoc/html/index.html.

[96] A. Greenbaum. *Iterative Methods for the Solution of Linear Systems*. SIAM, Philadelphia, 1997.

[97] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[98] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[99] A. Griewank. A mathematical view of automatic differentiation. In A. Iserles, editor, *Acta Numerica 2003*, pages 321–398. Cambridge University Press, Cambridge, London, New York, 2003.

[100] Andreas Griewank, David Juedes, and Jean Utke. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.

[101] M. D. Gunzburger. *Perspectives in Flow Control and Optimization*. SIAM, Philadelphia, 2003.

[102] M. D. Gunzburger, L. S. Hou, and S. S. Ravindran. Analysis and approximation of optimal control problems for a simplified ginzburg-landau model of superconductivity. *Numerische Mathematik*, 77:243–268, 1997. `http://link.springer.de/link/service/journals/00211/bibs/7077002/70770243.htm`.

[103] M. D. Gunzburger and S. Manservisi. The velocity tracking problem for Navier–Stokes flows with bounded distributed controls. *SIAM J. Control Optimization*, 37:1913–1945, 1999.

[104] M. D. Gunzburger and S. Manservisi. The velocity tracking problem for Navier–Stokes flows with boundary control. *SIAM J. Control Optimization*, 39:594–634, 2000.

[105] K. Harriman, P. Houston, B. Senior, and E. Süli. hp-version discontinuous galerkin methods with interior penalty for partial differential equations with nonnegative characteristic form. In C.-W. Shu, T. Tang, and S.-Y. Cheng, editors, *Recent Advances in Scientific Computing and Partial Differential Equations*, volume 330 of *Contemporary Mathematics*, pages 89–119. American Mathematical Society, 2003.

[106] R. Hartmann and P. Houston. Goal-oriented a posteriori error estimation for multiple target functionals. Technical Report 2002/32, University of Leicester, 2002.

[107] A. Haselbacher and J. Blazek. Accurate and Efficient discretization of Navier-Stokes Equations on Mixed Grids. *AIAA Journal*, 38(11):2094–2102, November 2000.

[108] M. Heinkenschloss. Time–domain decomposition iterative methods for the solution of distributed linear quadratic optimal control problems. Technical Report TR00–31, Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005–1892, 2000. appeared as [110].

[109] M. Heinkenschloss. Time–domain decomposition iterative methods for the solution of distributed linear quadratic optimal control problems: The discretized case. Technical Report SAND–xx, Sandia National Laboratories, 2003.

[110] M. Heinkenschloss. Time–domain decomposition iterative methods for the solution of distributed linear quadratic optimal control problems. *Journal of Computational and Applied Mathematics*, xx:xx–yy, 2004. In press.

[111] M. Heinkenschloss. Time–domain decomposition iterative methods for the solution of distributed linear quadratic optimal control problems. *Journal of Computational and Applied Mathematics*, 173:169–198, 2005.

[112] M. Heinkenschloss and F. Tröltzsch. Analysis of the Lagrange–SQP–Newton method for the control of a phase field equation. *Control and Cybernetics*, 28:177–211, 1999.

[113] M. A. Heroux. Epetra : Concrete C++ linear algebra classes for parallel linear algebra. http://software.sandia.gov/Trilinos.

[114] J.S. Hesthaven. From electrostatics to almost optimal nodal sets for polynomial interpolation in a simplex. *SJNA*, 35:655–676, 1998.

[115] J.S. Hesthaven and T. Warburton. Nodal high-order methods on unstructured grids. I. time-domain solution of Maxwell's equations. *JCP*, 181(1):186–221, 2002.

[116] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward.

[117] M. Hinze and K. Kunisch. Three control methods for time-dependent fluid flow. *Flow, Turbulence and Combustion. An International Journal*, 65(3-4):273–298, 2000.

[118] M. Hinze and K. Kunisch. Second order methods for optimal control of time–dependent fluid flow. *SIAM J. Control and Optimization*, 40:925–946, 2001.

[119] M. Hinze and S. Volkwein. Analysis of instantaneous control for the Burgers equation. *Nonlinear Analysis. Theory, Methods & Applications. An International Multidisciplinary Journal. Series A: Theory and Methods*, 50(1, Ser. A: Theory Methods):1–26, 2002.

[120] M. Hinze and A. Walter. An optimal memory-reduced procedure for calculating adjoints of the instationary Navier-Stokes equations. Technical Report MATH-NM–9-2002, Insitut für Numerische Mathematik, Technische Universität Dresden, Zellescher Weg 12-14, D–01062 Dresden, Germany, 2002.

[121] K.-H. Hoffmann and L. Jiang. Optimal control problem of a phase field model for solidification. *Numer. Funct. Anal.*, 13:11–27, 1992.

[122] Paul Houston, Bill Senior, and Endre Süli. Hp-discontinuous Galerkin finite element methods for hyperbolic problems: Error analysis and adaptivity. *Inter. J. Num. Meth. Fluids*, 40:153–169, 2002.

[123] Hspice user's manual. Technical report, Meta-Software, Inc., Campbell, California, 1996.

[124] Scott Hutchinson. Xyce users' guide. Technical report SAND98-xxxx, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2002.

[125] K. Ito and S. S. Ravindran. Optimal control of thermally convected fluid flows. *SIAM J. on Scientific Computing*, 19:1847–1869, 1998.

[126] J. J. Dongarra and J. Du Croz and S. Hammarling and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14:1–17, 1988.

[127] J. Jahn. *Introduction to the Theory of Nonlinear Optimization*. Springer Verlag, Berlin, Heidelberg, New York, second edition, 1996.

[128] A. Jameson. Aerodynamic design via control theory. *journal of Scientific Computing*, (3):233, 1988.

[129] A. Jameson, J. Alonso, J. Reuther, L. Martinelli, and J. Vassberg. Aerodynamic shape optimization techniques based on control theory. In *AIAA 98-2538*, volume 29, Albuquerque, NM, June 15-18 1998. AIAA.

[130] H. Jasak and A.D. Gosman. Element residual error estimate for the finite volume method. *Computers in Fluids*, 32:223–248, 2003.

[131] S.A. Hutchinson G.L. Hennigan K.D Devine A.G. Salinger J.N. Shadid H.K. Moffat. Mpsalsa a finite element computer program for reacting flow problems part 1 - theoretical development. Technical Report SAND95-2752, Sandia National Laboratories, 1995.

[132] T. Kaminski and M. Heimann. A coarse grid three-dimensional global inverse model of the atmospheric transport 1. andjoint model and jacobian matrix. *J. Geophysics*, 1999.

[133] E.R. Keiter, S.A. Hutchinson, R.J.Hoekstra, E.L. Rankin, T.V. Russo, and L.J. Waters. Computational algorithms for device-circuit coupling. Technical Report SAND2003-0080, Sandia National Laboroatories, January 2003.

[134] Eric R. Keiter. Xyce parallel electronic simulator design: Mathematical formulation. Technical report SAND02-xxxx, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2002.

[135] D. D. Knight. A Fully Implicit Navier-Stokes Algorithm Using an Unstructured Grid and Flux Difference Splitting. *Applied Numerical Mathematics*, 16:101–128, 1994.

[136] T. Kolda and R. Pawlowski. Nox: An object oriented, nonlinear solver package. `http://software.sandia.gov/Trilinos`.

[137] G. Kontarev. Adjoint equation technique applied to meteorological problems. Technical report, European Centre for Medium Range Weather Forecasts, 1980.

[138] T. Koornwinder. *Theory and Application of Special Functions*, chapter Two-Variable Analogues of the Classical Orthogonal Polynomials, pages 435–495. Academic Press, 1975.

[139] Kevin M. Kramer and W. Nicholas G. Hitchon. *Semiconductor Devices: A Simulation Approach*. Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[140] K. Kunisch and X. Marduel. Optimal control of non–isothermal viscoelastic fluid flows. *J. of Non-Newtonian Fluid Mechanics*, 88:261–301, 2000.

[141] M. G. Larson and T. J. Barth. A posteriori error estimation for adaptive discontinuous Galerkin approximations of hyperbolic systems. *Lecture Notes in Computational Science and Engineering*, 11:363–368, 2000.

[142] D. B. Leineweber, I. Bauer, H. G. Bock, and J. P. Schlöder. An efficient multiple shooting based reduced SQP strategy for large scale dynamic process optimization. part I: Theoretical aspects. *Comput. Chem. Engng.*, 27:157–166, 2003. http://www.iwr.uni-heidelberg.de/∼Daniel.Leineweber/.

[143] D. B. Leineweber, H. Schäfer, H. G. Bock, and J. P. Schlöder. An efficient multiple shooting based reduced SQP strategy for large scale dynamic process optimization. part II: Software aspects and applications. *Comput. Chem. Engng.*, 27:167–174, 2003.
http://www.iwr.uni-heidelberg.de/∼Daniel.Leineweber/.

[144] H. Levine and J. Schwinger. *Phys Review*, 1949.

[145] S. Li and L. Petzold. Design of new DASPK for sensitivity analysis. Technical report, Computational Sciences and Engineering, University of California, Santa Barbara, 1999. http://www.engineering.ucsb.edu/∼cse/.

[146] Shengtai Li and Linda Petzold. Software and algorithms for sensitivity analysis of large-scale differential algebraic systems. *Journal of Computational and Applied Mathematics*, 125:131–145, 2000.

[147] S. Linnainmaa. Taylor expansion of the accumulated rounding errror. *BIT (Nordisk Tidskrift for Informationsbehandling)*, 1976.

[148] M.-S. Liou and B. van Leer. Choice of Implicit and Explicit operators for the upwind differencing method. *AIAA Paper 1988-0624*, January 1988.

[149] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[150] Andrew Lumsdanie and Jeremy Siek. ITL : the Iterative Template Library. `http://www.osl.iu.edu/research/itl/`, 1998.

[151] H. Luo, D. Baum, J. D. Sharov, and R. Lohner. On the Computation of Compressible Turbulent Flows on Unstructured Grids. *International Journal of Computational Fluid Dynamics*, 14:253–270, 2001.

[152] H. Luo, J. D. Baum, and R. Lohner. Edge-Based Finite Element scheme for the Euler Equations. *AIAA Journal*, 32(6):1180–1190, June 1994.

[153] H. Luo, J. D. Baum, and R. Lohner. A fast, matrix-free implicit method for compressible flows on unstructured grids. *Journal of Computational Science*, 146:664–690, 1998.

[154] H. Luo, J. D. Baum, R. Lohner, and J. Cabello. Implicit Schemes and Boundary Conditions for Compressible Flows on Unstructured Meshes. *AIAA Paper 1994-0816*, January 1994.

[155] T. Maly and L. R. Petzold. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics*, 20:57–79, 1996.

[156] G.I. Marchuk and V.V. Penenko. Study of the sensitivity of discrete models of atmospheric and oceanic dynamics. *Atmospheric and Oceanic Physics*, 1980.

[157] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Trans. Math. Softw.*, 29(3):245–262, 2003.

[158] C. Mavriplis. Adaptive mesh stategies for the spectral element method. *Comp. Meth. in Appl. Mech. Eng.*, 116:77–86, 1994.

[159] Kartikeya Mayaram and Donald O. Pederson. Codecs: A mixed-level device and circuit simulator. *Proc. IEEE Int. Conf. Computer-Aided Design*, pages 112–115, November 1988.

[160] Kartikeya Mayaram and Donald O. Pederson. Coupling algorithms for mixed-level circuit and device simulation. *IEEE Transactions on Computer Aided Design*, II(8):1003–1012, 1992.

[161] J.R.F. McMacken and S.G.Chamberlain. Chord: A modular semiconductor device simulation tool incorporating external network models. *IEEE Transactions on Computer Aided Design*, 8:826–836, August 1989.

[162] Robert C. Melville, Ljiljana Trajkovic, San-Chin Fang, and Layne T. Watson. Globally convergent homotopy methods for the dc operating point problem. Technical report TR 90-61, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.

[163] Robert C. Melville, Ljiljana Trajkovic, San-Chin Fang, and Layne T. Watson. Artificial parameter homotopy methods for the dc operating point problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(6):861–877, 1993.

[164] Mike A. Heroux and Teri Barth and David Day and Rob Hoekstra and Rich Lehoucq and Kevin Long and Roger Pawlowski and Ray Tuminaro and Alan Williams. Trilinos : object-oriented, high-performance parallel solver libraries for the solution of large-scale complex multi-physics engineering and scientific applications. Web page.

[165] M. S. Mock. Time-dependent simulation of coupled devices. *Proc. NASECODE II*, pages 113–131, June 1981.

[166] MPI Fourum, the. *MPI: A Message Passing Interface Standard.* University of Tennessee, 1995. http://www.mpi-fourum.org/docs/docs.html.

[167] Jens-Dominik Müller and Michael B. Giles. Solution adaptive mesh refinement using adjoint error analysis. *AIAA*, (Paper 2001-2550), June 2001.

[168] S. K. Nadarajah and A. Jameson. A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. In *AIAA Paper 2000-0667*, 2000.

[169] Laurence Nagel and Ronald Rohrer. Computer analysis of nonlinear circuits, excluding radiation (cancer). *IEEE Journal of Solid-State Circuits*, sc-6(4):166–182, 1971.

[170] Nash, S. and A. Sofer. *Linear and Nonlinear Programming*. McGraw Hill, 1996.

[171] G. A. Newman and D. L. Alumbaugh. Electromagnetic modelling and inversion on massively parallel computers. *Geophys. J. Int.*, 128:345–354, 1997.

[172] G. A. Newman and G. M. Hoversten. Solution trategies for two and three dimensional electromagnetic inverse problems. *Journal of Inverse Problems*, 2000. to appear.

[173] T. V. Nguyen, A. Devgan, and O. J. Nastov. Adjoint transient sensitivity computation in piecewise linear simulation.

[174] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Verlag, 2000.

[175] J. Nocedal and Stephen Wright. *Numerical Optimization.* Springer, New York, 1999.

[176] H. J. Oberle and W. Grimm. BNDSCO-a program for the numerical solution of optimal control problems. Technical report, Institute for Flight Systems Dynamics, DLR, Oberpfaffenhofen, Germany, 1989.

[177] Department of Energy. Asci: Advanced simulation and computing initiative. `http://www.sandia.gov/ASCI`.

[178] Tom Quarles. Spice 3 version 3f5 users' manual. *University of California, Berkeley, CA*, 1994.

[179] R. Rannacher. Error control in finite element computations. an introduction to error estimation and mesh-size adaptation, 1998.

[180] J. Reuther and A. Jameson. Aerodynamic shape optimization of wing and wing-body congurations using control theory.

[181] J. Reuther, A. Jameson, J. Farmer, L. Martinelli, and D. Saunders. Aerodynamic shape optimization of complex aircraft configurations via an adjoint formulation. In *AIAA Paper 96–0094, 34th Aerospace Sciences Meeting and Exhibit, Jan. 15-18, 1996, Reno, NV*, 1996.

[182] Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the Second Extreme Linux workshop*, 1999.

[183] P. L. Roe. Approximate Riemann Solvers, Parameters Vectors and Difference Schemes. *Journal of Computational Physics*, 43:357–372, 1981.

[184] Gregory J. Rollins and John Choma. Mixed-mode pisces-spice coupled circuit and device solver. *IEEE Transactions on Computer Aided Design*, 7:862–867, August 1988.

[185] Francis Rotella. Mixed circuit and device simulation for analysis, design, and optimization of opto-electronic, radio requency, and high speed semiconductor devices. Phd thesis, Stanford University, 2000.

[186] R. Roussopolos. *Acad Sci*, 1953.

[187] C.W. Rowley. Model reduction for fluids, using balanced proper orthogonal decomposition. *Int Journal on Bifurcation and Chaos*, 2004.

[188] C. W. Ho A. E. Ruehli and P. A. Brennan. The modified nodal approach to network analysis. *IEEE Trans. Circuits Systems*, 22:505–509, 1988.

[189] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, New-York, Singapore, Toronto, 1996.

[190] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 7:856–869, 1986.

[191] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1996.

[192] A. G. Salinger, N.M. Bou-Rabee, E.A. Burroughs, R.B. Lehoucq, R.P. Pawlowski, L.A. Romero, and E.D. Wilkes. LOCA: A library of continuation algorithms - Theroy manual and user's guide. Technical report, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2002. SAND2002-0396.

[193] Sandia National Labs. ESI: Equation Solver Interface. `http://z.ca.sandia.gove/esi`, 2001.

[194] A. Sandu, D.N. Daescu, and G.R. Carmichael. Direct and adjoint sensitivity analysis of chemical kinetic systems with kpp: I - theory and software tools. *Atmospheric Environment*, 2003.

[195] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer-Verlag, New York, 1984.

[196] R. Serban, S. Li, and L. Petzold. Adaptive algorithms for optimal control of time–dependent partial differential–algebraic systems. *International Journal of Numerical Methods in Engineering*, xx:xx–yyy, 2001. To appear http://www.engineering.ucsb.edu/∼cse/.

[197] T. M. Smith, C. C. Ober, and A. A. Lorber. SIERRA/Premo - A New General Purpose Compressible Flow Simulation Code. *AIAA Paper 2002-3292*, June 2002.

[198] P. R. Spalart and S. R. Allmaras. A One-Equation Turbulence Model for Aerodynamic Flows. *AIAA Paper 92-0439*, 1992.

[199] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

[200] W.M. Stacey. Variational methods in nuclear reactor physics. Technical report, Academic, New York, 1974.

[201] James R. Stewart and Thomas J.R. Hughes. A tutorial in elementary finite element error analysis: A systematic presentation of a priori and a posteriori error estimates. *Comp. Meth. in Appl. Mech. Eng.*, 158:1–22, 1998.

[202] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, New York, Berlin, Heidelberg, London, Paris, second edition, 1993.

[203] D. A. Hocevar P. Yang T. N. Trick and B. D. Epler. Transient sensi-tivity computation for mosfet circuits. *IEEE Trans. Computer-Aided Design*, 1985.

[204] R.S. Tuminaro, M.A. Heroux, S.A. Hutchinson, and J.N. Shadid. *Official Aztec User's Guide: Version 2.1*. Albuquerque, NM 87185, 1999.

[205] O. Ghattas V. Akcelik, G. Biros. Parallel multiscale gauss-newton-krylov methods for inverse wave propagation. In *Proceedings of the IEEE/ACM SC2002 Conference, Baltimore*. IEEE/ACM, 2002.

[206] B.G van Bloemen Waanders, R.A. Bartlett, K.R. Long, P.T. Boggs, and A.G. Salinger. Large scale non-linear programming for PDE constrained optimization. Technical report, Sandia National Laboroatories, 2002.

[207] B. van Leer. Towards the Ultimate Conservation Difference Scheme III. Upstream-Centered Finite-Difference Schemes for Ideal Compressible Flow. *Journal of Computational Physics*, 23:263–275, 1977.

[208] R. S. Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1962.

[209] J. Vassberg and A. Jameson. Aerodynamic shape optimization of a reno airplane. *International Journal of Vehicle Design*, 2002.

[210] D. A. Venditti and D. L. Darmofal. A grid adpativity methodology for functional outputs of compressible flow simulations. In *AIAA, 15th Computational Fluis Dynamics Conference*, 2001.

[211] David A. Venditti and David L. Darmofal. Adjoint error estimatino and grid adaptation for functional outputs: Application to quasi-one-dimensional flow. *J. Comp. Phys.*, 164:204–227, 2000.

[212] Jiri Vlach and Kishore Singal. *Computer Methods for Circuit Analysis and Design*. Chapman and Hall, New York, 1994.

[213] S. Volkwein. Boundary control of the Burgers equation: Optimality conditions and reduced order approach. In *Optimal Control of Complex Structures*, International Series of Numerical Mathematics, Vol. 139, pages 267–278. Birkhäuser Verlag, 2002.

[214] R. Laur W. L. Engl and H. K. Dirks. Medusa - a simulator for modular circuits. *IEEE Trans. Computer-Aided Design*, CAD-1:85–93, 1982.

[215] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 1964.

[216] D. L. Whitaker. Three-Dimensional Unstructured Grid Euler Computations Using a Fully-Implicit, Upwind Method. *AIAA Paper 93-3337*, January 1993.

[217] E.P. Wigner. Effects of small perturbations on pile period. Technical report, Chicago Report CP-G-3048, 1945.

[218] K. Willcox and J. Peraire. Balanced model reduction via the proper orthogonal decomposition.

[219] C. S. Woodward, K. E. Grant, and R. Maxwell. Applications of sensitivity analysis to uncertainty quantification in variably saturated flow.

[220] Tor A. Fjeldly Trond Ytterdal and Michael Shur. *Introduction to Device Modeling and Circuit Simulation*. John Wiley and Sons, New York, 1988.

[221] X.D. Zhang, J.-Y. Trépanier, and R. Camarero. A posteriori error estimation for finite-volume solutions of hyperbolic conservation laws. *Comp. Meth. in Appl. Mech. Eng.*, 185:1–19, 2000.

APPENDIX I

Tensor-valued matrices, Jacobians and eigendecompositioins

For the gradient expressions below, we will need Jacobian matrices of various objects. The basic expressions needed for this are collected in this section.

An algebra of tensor-valued matrices

The flux functions in expression (7.14) are 3-by-1 matrices of first and second-order tensors. Various operations may be defined on $\mathbf{f}$. As we already have seen, the dot product with $\boldsymbol{n}$ may be defined by

element-wise operations,

$$\boldsymbol{n}\cdot\mathbf{f}=\mathbf{f}\cdot\boldsymbol{n}=\begin{pmatrix}\rho\boldsymbol{u}\cdot\boldsymbol{n}\\ \rho\boldsymbol{u}\boldsymbol{u}\cdot\boldsymbol{n}+\boldsymbol{n}p\\ \rho H\boldsymbol{u}\cdot\boldsymbol{n}\end{pmatrix}, \tag{3}$$

$$\boldsymbol{n}\cdot\mathbf{f}^{\mathrm{w}}=\mathbf{f}^{\mathrm{w}}\cdot\boldsymbol{n}=\begin{pmatrix}0\\ p\,\boldsymbol{n}\\ 0\end{pmatrix}. \tag{4}$$

This yields 3-by-1 matrices of the same dimension as the matrix of conservative variables (7.12). Note that the second-order tensors in the second component of $\mathbf{f}$ and $\mathbf{f}^{\mathrm{w}}$ are symmetric, which allows $\boldsymbol{n}$ and $\mathbf{f}$ to commute. Note also that these expressions are well defined with $\boldsymbol{n}$ being any first-order tensor, not just a unit normal.

Expression (3) may be differentiated with respect to the conservative variables to obtain the Jacobian matrix with respect to conservative variables

$$\frac{\partial(\mathbf{f}\cdot\boldsymbol{n})}{\partial\mathbf{w}}=\begin{pmatrix}0 & \boldsymbol{n} & 0\\ -\boldsymbol{u}\cdot\boldsymbol{n}\,\boldsymbol{u}+\boldsymbol{n}(\gamma-1)\dfrac{|\boldsymbol{u}|^2}{2} & \boldsymbol{u}\cdot\boldsymbol{n}\,\boldsymbol{I}+\boldsymbol{u}\otimes\boldsymbol{n}-(\gamma-1)\boldsymbol{n}\otimes\boldsymbol{u} & (\gamma-1)\boldsymbol{n}\\ \boldsymbol{u}\cdot\boldsymbol{n}\left(-H+\dfrac{\gamma-1}{2}|\boldsymbol{u}|^2\right) & \boldsymbol{n}H-\boldsymbol{u}\cdot\boldsymbol{n}(\gamma-1)\boldsymbol{u} & \gamma\boldsymbol{u}\cdot\boldsymbol{n}\end{pmatrix}, \tag{5}$$

Successively choosing $\boldsymbol{n}$ to be unit vectors in the Cartesian coordinate directions, we obtain three matrices that in the literature usually are referred to as the Jacobian matrices with respect to the conservative variables.

Differentiating the pressure with respect to the conservative variables yields

$$\frac{\partial p}{\partial\mathbf{w}}=(\gamma-1)\left(\frac{1}{2}|\boldsymbol{u}|^2,-\boldsymbol{u},1\right), \tag{6}$$

a 1-by-3 matrix of tensors of order zero, one and zero. Moreover, differentiating the "boundary flux" function (4), making use of expression (6) gives

$$\frac{\partial(\mathbf{f}^{\mathrm{w}}\cdot\boldsymbol{n})}{\partial\mathbf{w}}=\begin{pmatrix}0\\ \boldsymbol{n}\\ 0\end{pmatrix}\frac{\partial p}{\partial\mathbf{w}}=(\gamma-1)\begin{pmatrix}0 & \mathbf{0} & 0\\ \frac{1}{2}|\boldsymbol{u}|^2\boldsymbol{n} & -\boldsymbol{n}\otimes\boldsymbol{u} & \boldsymbol{n}\\ 0 & \mathbf{0} & 0\end{pmatrix}. \tag{7}$$

On the 3-by-1 and 3-by-3 matrix objects introduced above, we need to define standard elementary operations, slightly generalized to take into account the presence of tensor objects in the matrices. The first operation is an inner product involving two 3-by-1 matrices of dimensions like the conservative variables in (7.12),

$$\mathbf{w}=(w_1,\boldsymbol{w}_2,w_3)^T,\quad \mathbf{z}=(z_1,\boldsymbol{z}_2,z_3)^T.$$

The operation $\mathbf{w}^T\mathbf{z}=\mathbf{z}^T\mathbf{w}$ may be defined through the formula

$$\mathbf{w}^T\mathbf{z}=w_1z_1+\boldsymbol{w}_2\cdot\boldsymbol{z}_2+w_3z_3.$$

Note the dot product involved for the second components of $\mathbf{w}$ and $\mathbf{z}$.

A second operation involves, in addition to a matrix like $\mathbf{w}$ above, also a 3-by-1 matrix of dimension like the flux function, that is,

$$\mathbf{f} = (\boldsymbol{f}_1, \boldsymbol{F}_2, \boldsymbol{f}_3)^T,$$

in which $\boldsymbol{f}_1$ and $\boldsymbol{f}_3$ are first-order tensors and $\boldsymbol{F}_2$ a second-order tensor. The operations $\mathbf{f}^T\mathbf{w}$ and $\mathbf{w}^T\mathbf{f}$, returning first-order tensors may be defined by

$$\begin{aligned}\mathbf{f}^T\mathbf{w} &= \boldsymbol{f}_1 w_1 + \boldsymbol{F}_2\boldsymbol{w}_2 + \boldsymbol{f}_3 w_3,\\ \mathbf{w}^T\mathbf{f} &= w_1\boldsymbol{f}_1 + \boldsymbol{w}_2 \cdot \boldsymbol{F}_2 + w_3\boldsymbol{f}_3.\end{aligned}$$

Note the dot product involved for the second components, and note that $\mathbf{f}^T\mathbf{w} = \mathbf{w}^T\mathbf{f}$ if and only if the second-order tensor $\boldsymbol{F}_2$ is *symmetric*, which is the case for $\mathbf{f}$ being the flux function.

A third type of operation concerns matrices $\mathbf{A}$ with structure as the Jacobians (5) and (7) above. Such a matrix may be partitioned by columns as

$$\mathbf{A} = (\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \mathbf{a}^{(3)}),$$

in which $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(3)}$ are 3-by-1 matrices of the same dimension as the conservative variables and $\mathbf{a}^{(2)}$ is of the same dimension as the flux function. If now $\mathbf{g} = (g_1, \boldsymbol{g}_2, g_3)^T$ is a 3-by-1 matrix containing tensors of dimensions like the matrix $\mathbf{w}$ of conservative variables in (7.12), we may define the matrix operation

$$\mathbf{A}\mathbf{g} = \mathbf{a}^{(1)} g_1 + \mathbf{a}^{(2)} \cdot \boldsymbol{g}_2 + \mathbf{a}^{(3)} g_3 \tag{8}$$

returning an object of dimensions like the conservative variable $\mathbf{w}$. The dot product in expression (8) is applied to each row of $\mathbf{a}^{(2)}$. An analogous definition holds for the operation $\mathbf{g}^T\mathbf{A}$, returning an object of dimensions like $\mathbf{w}^T$.

A fourth and final operation concerns multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$ with structure as the Jacobians (5),

$$\mathbf{A} = \begin{pmatrix} a_{11} & \boldsymbol{a}_{12} & a_{13} \\ \boldsymbol{a}_{21} & \boldsymbol{A}_{22} & \boldsymbol{a}_{23} \\ a_{31} & \boldsymbol{a}_{32} & a_{33} \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} b_{11} & \boldsymbol{b}_{12} & b_{13} \\ \boldsymbol{b}_{21} & \boldsymbol{B}_{22} & \boldsymbol{b}_{23} \\ b_{31} & \boldsymbol{b}_{32} & b_{33} \end{pmatrix}.$$

Multiplication is defined by

$$\begin{aligned}&\mathbf{A}\mathbf{B} = \\ &= \begin{pmatrix} a_{11}b_{11} + \boldsymbol{a}_{12}\cdot\boldsymbol{b}_{21} + a_{13}b_{31} & a_{11}\boldsymbol{b}_{12} + \boldsymbol{a}_{12}\cdot\boldsymbol{B}_{22} + a_{13}\boldsymbol{b}_{32} & a_{11}b_{13} + \boldsymbol{a}_{12}\cdot\boldsymbol{b}_{23} + a_{13}b_{33} \\ \boldsymbol{a}_{21}b_{11} + \boldsymbol{A}_{22}\boldsymbol{b}_{21} + \boldsymbol{a}_{23}b_{31} & \boldsymbol{a}_{21}\otimes\boldsymbol{b}_{12} + \boldsymbol{A}_{22}\boldsymbol{B}_{22} + \boldsymbol{a}_{23}\otimes\boldsymbol{b}_{32} & \boldsymbol{a}_{21}b_{13} + \boldsymbol{A}_{22}\boldsymbol{b}_{23} + \boldsymbol{a}_{23}b_{33} \\ a_{31}b_{11} + \boldsymbol{a}_{32}\cdot\boldsymbol{b}_{21} + a_{33}b_{31} & a_{31}\boldsymbol{b}_{12} + \boldsymbol{a}_{32}\cdot\boldsymbol{B}_{22} + a_{33}\boldsymbol{b}_{32} & a_{31}b_{13} + \boldsymbol{a}_{32}\cdot\boldsymbol{b}_{23} + a_{33}b_{33} \end{pmatrix}.\end{aligned}$$

Change of variables and eigendecomposition

The 3-by-1 matrix of *primitive variables* is

$$\mathbf{v} = (\rho, \boldsymbol{u}, p)^T. \tag{9}$$

When performing variable changes in the Jacobians, we need the transformation matrices

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \begin{pmatrix} 1 & \mathbf{0} & 0 \\ \boldsymbol{u} & \rho \boldsymbol{I} & \mathbf{0} \\ \frac{1}{2}|\boldsymbol{u}|^2 & \rho \boldsymbol{u} & \frac{1}{\gamma - 1} \end{pmatrix}$$

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}} = \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right)^{-1} = \begin{pmatrix} 1 & \mathbf{0} & 0 \\ -\frac{\boldsymbol{u}}{\rho} & \frac{1}{\rho}\boldsymbol{I} & \mathbf{0} \\ (\gamma - 1)\frac{|\boldsymbol{u}|^2}{2} & -(\gamma - 1)\boldsymbol{u} & \gamma - 1 \end{pmatrix}$$

The Jacobian with respect to the primitive variables is

$$\left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right)^{-1} \frac{\partial (\mathbf{f} \cdot \boldsymbol{n})}{\partial \mathbf{w}} \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right) = \begin{pmatrix} \boldsymbol{u} \cdot \boldsymbol{n} & \rho \boldsymbol{n} & 0 \\ \mathbf{0} & \boldsymbol{u} \cdot \boldsymbol{n} \boldsymbol{I} & \frac{1}{\rho}\boldsymbol{n} \\ 0 & \rho c^2 \boldsymbol{n} & \boldsymbol{u} \cdot \boldsymbol{n} \end{pmatrix}, \tag{10}$$

where c denotes the speed of sound,

$$c^2 = \gamma \frac{p}{\rho}.$$

Comparing expressions (5) and (10), we see that the Jacobian with respect to the primitive variables has a simpler structure, and its eigendecomposition is also easier to compute. Since the two Jacobians are related by the similarity transformation $\partial \mathbf{w} / \partial \mathbf{v}$, they share the same eigenvalues, and their eigenvectors are related through the similarity transformation.

Denote by $\widehat{\boldsymbol{n}}$ the first-order tensor $\boldsymbol{n}$ scaled to unit length, that is,

$$\widehat{\boldsymbol{n}} = \frac{\boldsymbol{n}}{|\boldsymbol{n}|}.$$

The eigendecomposition of the Jacobian with respect to the primitive variables is

$$|\boldsymbol{n}| \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right)^{-1} \frac{\partial (\mathbf{f} \cdot \widehat{\boldsymbol{n}})}{\partial \mathbf{w}} \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right) = |\boldsymbol{n}| \, \mathbf{R} \boldsymbol{\Lambda} \mathbf{L}^T, \tag{11}$$

where $\mathbf{R}$ and $\mathbf{L} (= \mathbf{R}^{-T})$ are the matrices of right and left eigenvectors with respect to the primitive variables,

$$\mathbf{R} = \frac{1}{c^2} \begin{pmatrix} 1 & \widehat{\boldsymbol{n}} & 1 \\ -\frac{c}{\rho}\widehat{\boldsymbol{n}} & \frac{c}{\rho}\boldsymbol{S}_{\widehat{\boldsymbol{n}}} & \frac{c}{\rho}\widehat{\boldsymbol{n}} \\ c^2 & \mathbf{0} & c^2 \end{pmatrix}, \qquad \mathbf{L} = \begin{pmatrix} 0 & c^2 \widehat{\boldsymbol{n}} & 0 \\ -\frac{\rho c}{2}\widehat{\boldsymbol{n}} & \rho c \boldsymbol{S}_{\widehat{\boldsymbol{n}}} & \frac{\rho c}{2}\widehat{\boldsymbol{n}} \\ \frac{1}{2} & -\widehat{\boldsymbol{n}} & \frac{1}{2} \end{pmatrix} \tag{12}$$

$$\Lambda = \begin{pmatrix} \lambda^{(1)} & \mathbf{0} & 0 \\ \mathbf{0} & \lambda^{(2)} \boldsymbol{I} & \mathbf{0} \\ 0 & \mathbf{0} & \lambda^{(3)} \end{pmatrix},$$

where

$$\lambda^{(1)} = \boldsymbol{u} \cdot \widehat{\boldsymbol{n}} - c,$$
$$\lambda^{(2)} = \boldsymbol{u} \cdot \widehat{\boldsymbol{n}},$$
$$\lambda^{(3)} = \boldsymbol{u} \cdot \widehat{\boldsymbol{n}} + c.$$

Note the difference in dimensions of the three columns of $\mathbf{R}$ and $\mathbf{L}$ in expression (12): the first and third columns are of dimension like the conservative variable $\mathbf{w}$, whereas the second columns are of dimension like the flux function $\mathbf{f}$. This is a consequence of the d-multiplicity of the eigenvalue $\lambda^{(2)} = \boldsymbol{u} \cdot \boldsymbol{n}$. The second columns of $\mathbf{R}$ and $\mathbf{L}$ signify an *eigenspace* of dimension d associated with $\lambda^{(2)}$.

There are several possible choices of the second-order tensor $\boldsymbol{S}_{\widehat{\boldsymbol{n}}}$ that occurs in the second column of $\mathbf{R}$ and $\mathbf{L}$ in expression (12), each corresponding to a different choice of basis for the eigenspace of dimension d associated with eigenvalue $\lambda^{(2)}$. The second column of matrix $\mathbf{L}$ is a (block) eigenvector to Jacobian (10) for *any* second order tensor $\boldsymbol{S}_{\widehat{\boldsymbol{n}}}$ of rank $d-1$ such that $\boldsymbol{n} \cdot \boldsymbol{S}_{\widehat{\boldsymbol{n}}} = 0$. Corresponding (block) left eigenvector has the particular simple form given by the second column of matrix $\mathbf{R}$ if, in addition, $\boldsymbol{S}_{\widehat{\boldsymbol{n}}}$ satisfies

$$\boldsymbol{S}_{\widehat{\boldsymbol{n}}}^T \boldsymbol{S}_{\widehat{\boldsymbol{n}}} = \boldsymbol{I} - \widehat{\boldsymbol{n}} \otimes \widehat{\boldsymbol{n}}. \tag{13}$$

A natural choice of $\boldsymbol{S}_{\widehat{\boldsymbol{n}}}$, having the required properties, is the orthogonal projector

$$\boldsymbol{S}_{\widehat{\boldsymbol{n}}} = \boldsymbol{I} - \widehat{\boldsymbol{n}} \otimes \widehat{\boldsymbol{n}}. \tag{14}$$

Another possibility is to choose $\boldsymbol{S}_{\widehat{\boldsymbol{n}}} = \boldsymbol{S}^c_{\widehat{\boldsymbol{n}}}$, where $\boldsymbol{S}^c_{\widehat{\boldsymbol{n}}}$ is the skew-symmetric second-order-tensor representation of the cross product, that is,

$$\boldsymbol{S}^c_{\widehat{\boldsymbol{n}}} \boldsymbol{a} = \widehat{\boldsymbol{n}} \times \boldsymbol{a} \tag{15}$$

for each first-order tensor $\boldsymbol{a}$. The choice (15) also satisfies the properties outlined above.

Using the notation $[\cdot]$ for components of a tensor in a Cartesian coordinate system, we have that, if $[\widehat{\boldsymbol{n}}] = (n_1, n_2, n_3)^T$,

$$[\boldsymbol{S}_{\widehat{\boldsymbol{n}}}] = \begin{pmatrix} 1 - n_1^2 & -n_1 n_2 & -n_1 n_3 \\ -n_2 n_1 & 1 - n_2^2 & -n_2 n_3 \\ -n_3 n_1 & -n_3 n_2 & 1 - n_3^2 \end{pmatrix},$$

whereas

$$[\boldsymbol{S}^c_{\widehat{\boldsymbol{n}}}] = \begin{pmatrix} 0 & -n_3 & n_2 \\ n_3 & 0 & -n_1 \\ -n_2 & n_1 & 0 \end{pmatrix}.$$

Thus, the choice (15) leads to simpler and sparser expressions in the eigenvectors. However the choice (15) only makes sense for $d = 3$ (and, trivially, for $d = 1$, by setting $\boldsymbol{S}^c_{\boldsymbol{n}} = 0$) whereas the choice (14) is fine in all space dimensions. In the following, we will therefore assume the choice (14), although most expressions given below also hold for the choice (15). The only exception is for certain expressions occuring within derivatives of eigenvectors with respect to the normal.

Let us partition the eigenvector matrices in expression (11) by columns,

$$\mathbf{R} = (\mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \mathbf{r}^{(3)}), \qquad \mathbf{L} = (\mathbf{l}^{(1)}, \mathbf{l}^{(2)}, \mathbf{l}^{(3)}). \tag{16}$$

As explained above, the second column of $\mathbf{R}$ and $\mathbf{L}$ will be of different dimensions than the first and third column. Expanding in a Cartesian basis with basis vectors $\boldsymbol{e}_\alpha$, $\alpha = 1, \ldots, d$, we may alternatively write the eigenvector matrices partitioned as $2 + d$ columns of the same dimension, that is,

$$\begin{aligned} \mathbf{R} &= (\mathbf{r}^{(1)}, \mathbf{r}^{(2)} \cdot \boldsymbol{e}_1, \mathbf{r}^{(2)} \cdot \boldsymbol{e}_2, \mathbf{r}^{(2)} \cdot \boldsymbol{e}_3, \mathbf{r}^{(3)}), \\ \mathbf{L} &= (\mathbf{l}^{(1)}, \mathbf{l}^{(2)} \cdot \boldsymbol{e}_1, \mathbf{l}^{(2)} \cdot \boldsymbol{e}_2, \mathbf{l}^{(2)} \cdot \boldsymbol{e}_3, \mathbf{l}^{(3)}), \end{aligned} \tag{17}$$

in 3D.

Using partitioning (16), the eigendecomposition (11) may also be written

$$\begin{aligned} &|\boldsymbol{n}| \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}}\right)^{-1} \frac{\partial (\mathbf{f} \cdot \widehat{\boldsymbol{n}})}{\partial \mathbf{w}} \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}}\right) \\ &\quad = |\boldsymbol{n}| \left[\lambda^{(1)} \mathbf{r}^{(1)} \mathbf{l}^{(1)\,T} + \lambda^{(2)} \mathbf{r}^{(2)} \cdot \mathbf{l}^{(2)\,T} + \lambda^{(3)} \mathbf{r}^{(3)} \mathbf{l}^{(3)\,T}\right]. \end{aligned} \tag{18}$$

Note the dot product for the second term in the expansion. Using partitioning (17), an alternative way of writing expansion (18) is

$$\begin{aligned} &|\boldsymbol{n}| \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}}\right)^{-1} \frac{\partial (\mathbf{f} \cdot \widehat{\boldsymbol{n}})}{\partial \mathbf{w}} \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}}\right) \\ &\quad = |\boldsymbol{n}| \left[\lambda^{(1)} \mathbf{r}^{(1)} \mathbf{l}^{(1)\,T} + \sum_{\alpha=1}^{d} \lambda^{(2)} \left(\mathbf{r}^{(2)} \cdot \boldsymbol{e}_\alpha\right) \left(\mathbf{l}^{(2)} \cdot \boldsymbol{e}_\alpha\right)^T + \lambda^{(3)} \mathbf{r}^{(3)} \mathbf{l}^{(3)\,T}\right]. \end{aligned}$$

The matrix of right eigenvectors with respect to the conservative variables is

$$\mathbf{R}_w = \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}}\right) \mathbf{R} = \frac{1}{c^2} \begin{pmatrix} 1 & \widehat{\boldsymbol{n}} & 1 \\ \boldsymbol{u} - c\widehat{\boldsymbol{n}} & \boldsymbol{u} \otimes \widehat{\boldsymbol{n}} + c\boldsymbol{S}_{\widehat{\boldsymbol{n}}} & \boldsymbol{u} + c\widehat{\boldsymbol{n}} \\ H - c\boldsymbol{u} \cdot \widehat{\boldsymbol{n}} & \widehat{\boldsymbol{n}} \dfrac{|\boldsymbol{u}|^2}{2} + c\boldsymbol{u} \cdot \boldsymbol{S}_{\widehat{\boldsymbol{n}}} & H + c\boldsymbol{u} \cdot \widehat{\boldsymbol{n}}, \end{pmatrix}, \tag{19}$$

which, like $\mathbf{R}$, can be partitioned by columns,

$$\begin{aligned} \mathbf{R}_w &= (\mathbf{r}_w^{(1)}, \mathbf{r}_w^{(2)}, \mathbf{r}_w^{(3)}) \\ &= (\mathbf{r}_w^{(1)}, \mathbf{r}_w^{(2)} \cdot \boldsymbol{e}_1, \mathbf{r}_w^{(2)} \cdot \boldsymbol{e}_2, \mathbf{r}_w^{(2)} \cdot \boldsymbol{e}_3, \mathbf{r}_w^{(3)}). \end{aligned} \tag{20}$$

Roe-type dissipation

We now define the dissipation term $\mathbf{d}(\mathbf{w}_{i+}, \mathbf{w}_{j-}, \boldsymbol{n}_{ij})$. Here we simply refer to right and left states $\mathbf{w}_{i+}$ and $\mathbf{w}_{j-}$ respectively. How these right and left states are defined is not of concern here as all of the properties that we consider here are independent of how the states are defined. We begin with a few definitions. For each edge $\xrightarrow{i\,j}$ in the mesh, attached to nodes i and j, define the *Roe averaged* quantities

$$\begin{aligned} \underset{\sim}{\rho} &= (\rho_{i+}\rho_{j-})^{1/2}, \\ \widetilde{\boldsymbol{u}} = \frac{\rho_{i+}^{1/2}\boldsymbol{u}_{i+} + \rho_{j-}^{1/2}\boldsymbol{u}_{j-}}{\rho_{i+}^{1/2} + \rho_{j-}^{1/2}}, &\qquad \widetilde{H} = \frac{\rho_{i+}^{1/2}H_{i+} + \rho_{j-}^{1/2}H_{j-}}{\rho_{i+}^{1/2} + \rho_{j-}^{1/2}}. \end{aligned} \tag{21}$$

From these, we may define

$$\widetilde{c} = \left[(\gamma-1)(\widetilde{H} - \frac{1}{2}\,|\widetilde{\boldsymbol{u}}|^2)\right]^{1/2}.$$

(Equivalently, the quantity $\widetilde{c}^2$ can also be computed by averaging c_{i+}^2 and c_{j-}^2 similarly as is done for $\widetilde{\boldsymbol{u}}$ and $\widetilde{H}$.)

The Roe-averaged matrix of right eigenvectors with respect to the conservative variables is defined as

$$\widetilde{\mathbf{R}}_w = \frac{1}{\widetilde{c}^2}\begin{pmatrix} 1 & \widehat{\boldsymbol{n}}_{ij} & 1 \\ \widetilde{\boldsymbol{u}} - \widetilde{c}\,\widehat{\boldsymbol{n}}_{ij} & \widetilde{\boldsymbol{u}} \otimes \widehat{\boldsymbol{n}}_{ij} + \widetilde{c}\boldsymbol{S}_{\widehat{\boldsymbol{n}}_{ij}} & \widetilde{\boldsymbol{u}} + \widetilde{c}\,\widehat{\boldsymbol{n}}_{ij} \\ \widetilde{H} - \widetilde{c}\,\widetilde{\boldsymbol{u}}\cdot\widehat{\boldsymbol{n}}_{ij} & \widehat{\boldsymbol{n}}_{ij}\dfrac{|\widetilde{\boldsymbol{u}}|^2}{2} + \widetilde{c}\,\widetilde{\boldsymbol{u}}\cdot\boldsymbol{S}_{\widehat{\boldsymbol{n}}_{ij}} & \widetilde{H} + \widetilde{c}\,\widetilde{\boldsymbol{u}}\cdot\widehat{\boldsymbol{n}}_{ij}, \end{pmatrix}, \tag{22}$$

that is, matrix (19) with the entries replaced by corresponding Roe-averaged quantities and using the unit normal $\widehat{\boldsymbol{n}}_{ij}$ associated with the current edge. Similarly, we may define the Roe-averaged matrix of left eigenvectors with respect to the primitive variables as

$$\widetilde{\mathbf{L}} = \begin{pmatrix} 0 & \widetilde{c}^2\,\widehat{\boldsymbol{n}}_{ij} & 0 \\ -\dfrac{\underset{\sim}{\rho}\,\widetilde{c}}{2}\widehat{\boldsymbol{n}}_{ij} & \underset{\sim}{\rho}\,\widetilde{c}\,\boldsymbol{S}_{\widehat{\boldsymbol{n}}_{ij}} & \dfrac{\underset{\sim}{\rho}\,\widetilde{c}}{2}\widehat{\boldsymbol{n}}_{ij} \\ \frac{1}{2} & -\widehat{\boldsymbol{n}}_{ij} & \frac{1}{2} \end{pmatrix}. \tag{23}$$

We may partition $\widetilde{\mathbf{R}}_w$ and $\widetilde{\mathbf{L}}$ by columns, similarly as was done for $\mathbf{R}$ and $\mathbf{L}$ in expressions (16) and (17),

$$\begin{aligned} \widetilde{\mathbf{R}}_w &= (\widetilde{\mathbf{r}}_w^{(1)}, \widetilde{\mathbf{r}}_w^{(2)}, \widetilde{\mathbf{r}}_w^{(3)}) \\ &= (\widetilde{\mathbf{r}}_w^{(1)}, \widetilde{\mathbf{r}}_w^{(2)}\cdot\boldsymbol{e}_1, \widetilde{\mathbf{r}}_w^{(2)}\cdot\boldsymbol{e}_2, \widetilde{\mathbf{r}}_w^{(2)}\cdot\boldsymbol{e}_3, \widetilde{\mathbf{r}}_w^{(3)}), \\ \widetilde{\mathbf{L}} &= (\widetilde{\mathbf{l}}^{(1)}, \widetilde{\mathbf{l}}^{(2)}, \widetilde{\mathbf{l}}^{(3)}) \\ &= (\widetilde{\mathbf{l}}^{(1)}, \widetilde{\mathbf{l}}^{(2)}\cdot\boldsymbol{e}_1, \widetilde{\mathbf{l}}^{(2)}\cdot\boldsymbol{e}_2, \widetilde{\mathbf{l}}^{(2)}\cdot\boldsymbol{e}_3, \widetilde{\mathbf{l}}^{(3)}). \end{aligned} \tag{24}$$

The Roe-averaged eigenvalues are obtained similarly,

$$\begin{aligned} \widetilde{\lambda}^{(1)} &= \widetilde{\boldsymbol{u}}\cdot\widehat{\boldsymbol{n}}_{ij} - \widetilde{c}, \\ \widetilde{\lambda}^{(2)} &= \widetilde{\boldsymbol{u}}\cdot\widehat{\boldsymbol{n}}_{ij}, \\ \widetilde{\lambda}^{(3)} &= \widetilde{\boldsymbol{u}}\cdot\widehat{\boldsymbol{n}}_{ij} + \widetilde{c}. \end{aligned}$$

We introduce a simple *entropy fix* to prevent the eigenvalues to become exactly zero,

$$|\widetilde{\lambda}_e^{(I)}| = \begin{cases} \dfrac{(\widetilde{\lambda}^{(I)})^2 + \epsilon^2}{2\epsilon}, & \text{for } |\widetilde{\lambda}^{(I)}| \le \epsilon, \\ |\widetilde{\lambda}^{(I)}| & \text{otherwise}, \end{cases} \qquad I = 1, 2, 3, \tag{25}$$

and define the matrix

$$|\widetilde{\Lambda}_e| = \begin{pmatrix} |\widetilde{\lambda}_e^{(1)}| & \mathbf{0} & 0 \\ \mathbf{0} & |\widetilde{\lambda}_e^{(2)}|\boldsymbol{I} & \mathbf{0} \\ 0 & \mathbf{0} & |\widetilde{\lambda}_e^{(3)}| \end{pmatrix}.$$

The dissipation matrix can now be specified as

$$\mathbf{d}\left(\mathbf{w}_{i+},\mathbf{w}_{j-},\boldsymbol{n}_{ij}\right)=-\frac{1}{2}\left|\boldsymbol{n}_{ij}\right|\widetilde{\mathbf{R}}_{w}|\widetilde{\Lambda}_{e}|\widetilde{\mathbf{L}}^{T}\left(\mathbf{v}_{j-}-\mathbf{v}_{i+}\right). \tag{26}$$

Alternatively, using the partitioning (24) by columns, the dissipation can be written similarly as in expression (18),

$$\begin{aligned}\mathbf{d}\left(\mathbf{w}_{i+},\mathbf{w}_{j-},\boldsymbol{n}_{ij}\right)&=-\frac{1}{2}\left|\boldsymbol{n}_{ij}\right|\Big(|\widetilde{\lambda}_{e}^{(1)}|\,\widetilde{\mathbf{r}}_{w}^{(1)}\widetilde{\mathbf{l}}^{(1)\,T}+|\widetilde{\lambda}_{e}^{(2)}|\,\widetilde{\mathbf{r}}_{w}^{(2)}\cdot\widetilde{\mathbf{l}}^{(2)\,T}+|\widetilde{\lambda}_{e}^{(3)}|\,\widetilde{\mathbf{r}}_{w}^{(3)}\widetilde{\mathbf{l}}^{(3)\,T}\Big).\\&=-\frac{1}{2}\left|\boldsymbol{n}_{ij}\right|\Big(|\widetilde{\lambda}_{e}^{(1)}|\,\widetilde{\mathbf{r}}_{w}^{(1)}\widetilde{\mathbf{l}}^{(1)\,T}\\&\qquad+\sum_{\alpha=1}^{d}|\widetilde{\lambda}_{e}^{(2)}|\,(\widetilde{\mathbf{r}}_{w}^{(2)}\cdot\boldsymbol{e}_{\alpha})(\widetilde{\mathbf{l}}^{(2)}\cdot\boldsymbol{e}_{\alpha})^{T}+|\widetilde{\lambda}_{e}^{(3)}|\,\widetilde{\mathbf{r}}_{w}^{(3)}\widetilde{\mathbf{l}}^{(3)\,T}\Big).\end{aligned} \tag{27}$$

Note that the following symmetry property holds

Lemma 3.

$$\mathbf{d}\left(\mathbf{w}_{i+},\mathbf{w}_{j-},\boldsymbol{n}_{ij}\right)=-\mathbf{d}\left(\mathbf{w}_{j-},\mathbf{w}_{i+},-\boldsymbol{n}_{ij}\right)$$

Derivation of partial derivatives of the reconstruction

Differentiating the limiters (7.21) yields

$$\phi_{b}(a,b)=\begin{cases}2a\dfrac{a^{2}-b^{2}+\epsilon}{(a^{2}+b^{2}+\epsilon)^{2}}\,\chi(ab) & \text{(van Albada),}\\[2ex] 4a\dfrac{a^{2}-b^{2}+\epsilon}{\left[(a+b)^{2}+\epsilon\right]^{2}}\,\chi(ab) & \text{(van Leer).}\end{cases} \tag{28}$$

Expressions (7.21) and (28) imply the symmetry properties

$$\phi(b,a)=\phi(a,b),\qquad\phi_{a}(b,a)=\phi_{b}(a,b),\qquad\phi_{b}(-a,-b)=-\phi_{b}(a,b). \tag{29}$$

Differentiating the reconstruction expressions (7.19), (7.22) yields

$$\begin{aligned}\delta\mu_{i+}&=\delta\mu_{i}+\Big\{\phi_{a}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})\left[\delta\mu_{j}-\delta\mu_{i}\right]\\&\qquad+\phi_{b}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})\big[2\boldsymbol{t}_{ij}\cdot\nabla_{h}\delta\mu_{i}-(\delta\mu_{j}-\delta\mu_{i})\big]\Big\}\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_{h}\mu_{i}\\&\quad+\phi(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_{h}\delta\mu_{i}\\&=\Big\{1+\big[\phi_{b}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})-\phi_{a}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})\big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_{h}\mu_{i}\Big\}\delta\mu_{i}\\&\quad-\Big\{\big[\phi_{b}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})-\phi_{a}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})\big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_{h}\mu_{i}\Big\}\delta\mu_{j}\\&\quad+\Big[\phi(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})+\phi_{b}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})2\boldsymbol{t}_{ij}\cdot\nabla_{h}\mu_{i}\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_{h}\delta\mu_{i}\\&=\big(1+\alpha_{ij}^{i}\big)\delta\mu_{i}-\alpha_{ij}^{i}\,\delta\mu_{j}\\&\quad+\Big[\phi(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})+\phi_{b}(\Delta^{+}\mu_{i},\Delta^{-}\mu_{i})2\boldsymbol{t}_{ij}\cdot\nabla_{h}\mu_{i}\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_{h}\delta\mu_{i},\end{aligned} \tag{30}$$

$$
\begin{aligned}
\delta\mu_{j-} &= \delta\mu_j - \Big\{\phi_a(\Delta^+\mu_j, \Delta^-\mu_j)\big[2\boldsymbol{t}_{ij}\cdot\nabla_h\delta\mu_j - (\delta\mu_j - \delta\mu_i)\big] \\
&\qquad + \phi_b(\Delta^+\mu_j, \Delta^-\mu_j)(\delta\mu_j - \delta\mu_i)\Big\}\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\mu_j \\
&\quad - \phi(\Delta^+\mu_j, \Delta^-\mu_j)\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\delta\mu_j \\
&= \Big\{1 + \big[\phi_a(\Delta^+\mu_j, \Delta^-\mu_j) - \phi_b(\Delta^+\mu_j, \Delta^-\mu_j)\big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\mu_j\Big\}\delta\mu_j \\
&\quad - \Big\{\big[\phi_a(\Delta^+\mu_j, \Delta^-\mu_j) - \phi_b(\Delta^+\mu_j, \Delta^-\mu_j)\big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\mu_j\Big\}\delta\mu_i \\
&\quad - \Big[\phi(\Delta^+\mu_j, \Delta^-\mu_j) + \phi_a(\Delta^+\mu_j, \Delta^-\mu_j)2\boldsymbol{t}_{ij}\cdot\nabla\mu_j\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\delta\mu_j \\
&= \big(1 - \alpha_{ij}^j\big)\delta\mu_j + \alpha_{ij}^j\,\delta\mu_i \\
&\quad - \Big[\phi(\Delta^+\mu_j, \Delta^-\mu_j) + \phi_a(\Delta^+\mu_j, \Delta^-\mu_j)2\boldsymbol{t}_{ij}\cdot\nabla\mu_j\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\delta\mu_j,
\end{aligned}
\tag{31}
$$

where

$$
\alpha_{ij}^k = \Big[\phi_b(\Delta^+\mu_k, \Delta^-\mu_k) - \phi_a(\Delta^+\mu_k, \Delta^-\mu_k)\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\mu_k \qquad k \in \{\, i, j \,\}. \tag{32}
$$

Moreover, by definition (7.22),

$$
\phi_b(\Delta^+\mu_i, \Delta^-\mu_i) = \phi_b(\mu_j - \mu_i, 2\boldsymbol{t}_{ij}\cdot\nabla_h\mu_i - (\mu_j - \mu_i)), \tag{33}
$$

and by property (29) and definition (7.22),

$$
\phi_a(\Delta^+\mu_j, \Delta^-\mu_j) = \phi_b(\Delta^-\mu_j, \Delta^+\mu_j) = \phi_b(\mu_j - \mu_i, 2\boldsymbol{t}_{ij}\cdot\nabla_h\mu_j - (\mu_j - \mu_i)). \tag{34}
$$

Defining

$$
\beta_{ij}^k = \phi_b\big(\mu_j - \mu_i, 2\boldsymbol{t}_{ij}\cdot\nabla_h\mu_k - (\mu_j - \mu_i)\big)\,2\boldsymbol{t}_{ij}\cdot\nabla_h\mu_k. \tag{35}
$$

implies that, by expressions (33) and (34),

$$
\begin{aligned}
\phi_b(\Delta^+\mu_i, \Delta^-\mu_i)\,2\boldsymbol{t}_{ij}\cdot\nabla_h\mu_i &= \beta_{ij}^i \\
\phi_a(\Delta^+\mu_j, \Delta^-\mu_j)\,2\boldsymbol{t}_{ij}\cdot\nabla_h\mu_j &= \beta_{ij}^j
\end{aligned}
\tag{36}
$$

Substituting expression (36) into expressions (30) and (31) yields that

$$
\begin{aligned}
\delta\mu_{i+} &= \big(1 + \alpha_{ij}^i\big)\delta\mu_i - \alpha_{ij}^i\,\delta\mu_j + \Big[\phi(\Delta^+\mu_i, \Delta^-\mu_i) + \beta_{ij}^i\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\delta\mu_i, \\
\delta\mu_{j-} &= \big(1 - \alpha_{ij}^j\big)\delta\mu_j + \alpha_{ij}^j\,\delta\mu_i - \Big[\phi(\Delta^+\mu_j, \Delta^-\mu_j) + \beta_{ij}^j\Big]\frac{\boldsymbol{t}_{ij}}{2}\cdot\nabla_h\delta\mu_j,
\end{aligned}
\tag{37}
$$

By comparing (7.39) and (37) it is clear that the partial derivatives of the reconstruction are:

$$
\begin{aligned}
\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i} &= \text{diag}\left\{1 + \alpha_{ij}^i\right\} \\
\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j} &= \text{diag}\left\{-\alpha_{ij}^i\right\} \\
\frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^i} &= \text{diag}\left\{+\tfrac{1}{2}\left[\phi(\Delta^+\mu_i, \Delta^-\mu_i) + \beta_{ij}^i\right]\right\} \\
\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i} &= \text{diag}\left\{+\alpha_{ij}^j\right\} \\
\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j} &= \text{diag}\left\{1 - \alpha_{ij}^j\right\} \\
\frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^j} &= \text{diag}\left\{-\tfrac{1}{2}\left[\phi(\Delta^+\mu_j, \Delta^-\mu_j) + \beta_{ij}^j\right]\right\}
\end{aligned}
$$

where each term is defined component-wise.

APPENDIX II

Implementation of "assignment to scalar" `RTOp` transformation operator.

```
// ////////////////////////////////////////////////////
// RTOp_TOp_assign_scalar.h

#ifndef RTOP_TOP_ASSIGN_SCALAR_H
#define RTOP_TOP_ASSIGN_SCALAR_H

#include "RTOpPack/include/RTOp.h"

#ifdef __cplusplus
extern "C" {
#endif

extern const struct RTOp_RTOp_vtbl_t
    RTOp_TOp_assign_scalar_vtbl;
int RTOp_TOp_assign_scalar_construct( RTOp_value_type alpha,
    struct RTOp_RTOp* op );
int RTOp_TOp_assign_scalar_destroy( struct RTOp_RTOp* op );
int RTOp_TOp_assign_scalar_set_alpha( RTOp_value_type alpha,
    struct RTOp_RTOp* op );

#ifdef __cplusplus
}
#endif

#endif  // RTOP_TOP_ASSIGN_SCALAR_H

// ////////////////////////////////////////////////////
// RTOp_TOp_assign_scalar.c
```

```
#include "RTOpStdOpsLib/include/RTOp_TOp_assign_scalar.h"
#include "RTOpPack/include/RTOp_obj_value_vtbl.h"
#include "RTOpPack/include/RTOp_obj_null_vtbl.h"

static int RTOp_TOp_assign_scalar_apply_op(
    const struct RTOp_RTOp_vtbl_t* vtbl, const void* obj_data
    ,const int num_vecs, const struct RTOp_SubVector vecs[]
    ,const int num_targ_vecs
    ,const struct RTOp_MutableSubVector targ_vecs[]
    ,RTOp_ReductTarget targ_obj )
{
    RTOp_value_type   alpha = *((RTOp_value_type*)obj_data);
    RTOp_index_type   z_sub_dim = targ_vecs[0].sub_dim;
    RTOp_value_type   *z_val    = targ_vecs[0].values;
    ptrdiff_t         z_val_s   = targ_vecs[0].values_stride;
    RTOp_index_type   k;
    if( num_vecs != 0 || vecs != NULL )
        return RTOp_ERR_INVALID_NUM_VECS;
    if( num_targ_vecs != 1 || targ_vecs == NULL )
        return RTOp_ERR_INVALID_NUM_TARG_VECS;
    for( k = 0; k < z_sub_dim; ++k, z_val += z_val_s )
        *z_val = alpha;
    return 0;
}

const struct RTOp_RTOp_vtbl_t RTOp_TOp_assign_scalar_vtbl =
{
    &RTOp_obj_value_vtbl
    ,&RTOp_obj_null_vtbl
    ,"TOp_assign_scalar"
    ,NULL
    ,RTOp_TOp_assign_scalar_apply_op
    ,NULL
    ,NULL
};

int RTOp_TOp_assign_scalar_construct( RTOp_value_type alpha
    ,struct RTOp_RTOp* op )
{
    op->vtbl = &RTOp_TOp_assign_scalar_vtbl;
    op->vtbl->obj_data_vtbl->obj_create(NULL,NULL
        ,&op->obj_data);
    *((RTOp_value_type*)op->obj_data) = alpha;
    return 0;
}

int RTOp_TOp_assign_scalar_destroy( struct RTOp_RTOp* op )
{
    op->vtbl->obj_data_vtbl->obj_free(NULL,NULL,&op->obj_data);
    op->vtbl = NULL;
    return 0;
}

int RTOp_TOp_assign_scalar_set_alpha( RTOp_value_type alpha
    ,struct RTOp_RTOp* op )
{
    *((RTOp_value_type*)op->obj_data) = alpha;
    return 0; // success?
}
```

Automatic generation of RTOp subclasses in C

To make the process of creating an RTOp C subclass easier and because there is boiler-plate code that is needed, a Perl script called `new_rtop.pl` has been implemented. This script automates most of the work required to create a new RTOp subclass. This script prompts the user for the answers to a set of questions about the operation being performed. In many cases, the output header and source files will be ready to compile and use. In other cases, the user will have to finish the implementation.

Many different types of specialized operators, including all of the example operators in (8.1)–(8.4), can be completely implemented with the script. Below, we show the use of this script in generating the source code for C RTOp subclasses for the example specialized operators in (8.2) and (8.4).

Example transformation operator

The Perl script is first demonstrated on the transformation operator in (8.4). Note that all of the data required to perform the operation is contained in the four input vectors a, b, u and w. The only exception is the value of ∞ which may be platform dependent. Therefore, we will allow the ANA to define the value of ∞ as an operator object instance data member called `inf_val`. Before running the script, the vector arguments are ordered and mapped into the generic names $v^0 = a$, $v^1 = b$, $v^2 = u$, $v^3 = w$, $z^0 = d$ and then (8.4) is restated as:

$$z_i^0 \leftarrow \begin{cases} (v^l - v^2)_i^{1/2} & \text{if } v_i^3 < 0 \text{ and } v_i^1 < +\infty \\ 1 & \text{if } v_i^3 < 0 \text{ and } v_i^1 = +\infty \\ (v^2 - v^0)_i^{1/2} & \text{if } v_i^3 \geq 0 \text{ and } v_i^0 > -\infty \\ 1 & \text{if } v_i^3 \geq 0 \text{ and } v_i^0 = -\infty \end{cases} . \tag{38}$$

We will call this operator subclass `TOp_trice_diag_scal`. The interactive session with the script `new_rtop.pl` is shown below:

```
*******************************************
*** Create a new C RTOp operator subclass ***
*******************************************

1) What is the name of your operator subclass?
: TOp_trice_diag_scal

2) Is your operator coordinate invariant?
[y] or [n] : y

3) Give the number of nonmutable input vectors (vi, i=0...num_vecs-1)?
: 4

4) Give the number of mutable input/output vectors (zi, i=0...num_targ_vecs)?
: 1

5) Does your operator require extra data which is not in the input vectors?
```

```
[y] or [n] : y

Choose the structure of the data:
1: {index}
2: {value}
3: {value,index}
4: {value,value}
5: other
Choose 1-5? 2

Give name for {value} member?
: inf_val

6) Does your operator perform a reduction?
[y] or [n] : n

7.a) Does your element-wise operation(s) need temporary variables?
[y] or [n] : n

7.c) Give the C statement(s) for element-wise transformation operation?
   You can choose:
        Non-mutable operator data (don't change here)   : inf_val
        Non-mutable vector elements (don't change here) : v0, v1, v2, v3
        Mutable vector elements (must modify here)      : z0
? if(v3 < 0 && v1 < +inf_val )
?   z0 = sqrt(v1-v2);
? else if(v3 < 0 && v1 >= +inf_val )
?   z0 = 1;
? else if(v3 >= 0 && v0 > -inf_val )
?   z0 = sqrt(v2-v0);
? else if(v3 >= 0 && v0 <= -inf_val )
?   z0 = 1;
?

The implementation files RTOp_TOp_trice_diag_scal.h and
RTOp_TOp_trice_diag_scal.c should be complete!
```

After the script creates these files, they just need to be integrated into the build system (i.e. added to the makefile) and compiled. The only part of the implemented RTOp subclass that is more than just boiler-plate code is the loop that actually performs the element-wise transformation. Below is a snippet of code from the static function `RTOp_TOp_trice_diag_scal_apply_op(...)` for the loop that actually performs the user-defined element-wise transformation.

```
for( k = 0; k < sub_dim; ++k, v0_val += v0_val_s, v1_val += v1_val_s
    ,v2_val += v2_val_s, v3_val += v3_val_s, z0_val += z0_val_s )
{
    // Element-wise transformation
    if((*v3_val) < 0 && (*v1_val) < +(*inf_val))
        (*z0_val) = sqrt((*v1_val)-(*v2_val));
    else if((*v3_val) < 0 && (*v1_val) >= +(*inf_val))
        (*z0_val) = 1;
    else if((*v3_val) >= 0 && (*v0_val) > -(*inf_val))
        (*z0_val) = sqrt((*v2_val)-(*v0_val));
    else if((*v3_val) >= 0 && (*v0_val) <= -(*inf_val))
```

```
        (*z0_val) = 1;
}
```

The above code loops over a chunk of vector elements using BLAS-compatible strided iterators (of dimension `sub_dim` which are provided by the vector implementation) and performs the transformation operation. The generated source code can then be manually post-modified (and perhaps better optimized).

The developer of an ANA implemented in C++, for instance, can include the header file `RTOp_TOp_trice_diag_scal.h` and then a RTOp object for this transformation operator can be created, used and destroyed as:

```
#include "RTOp_TOp_trice_diag_scal.h"

...

void trice_diag_scale( const AbstractVector& a, const AbstractVector& b
    ,const AbstractVector& u, const AbstractVector& w, const AbstractVector& d )
{
    // Create and initialize the instance data for the operator
    const RTOp_value_type  inf_val = 1e+50;
    const RTOp_RTOp        trice_scal_op;
    RTOp_TOp_trice_diag_scal_construct(inf_val,&trice_scal_op);
    // Apply the operator to the existing vectors a, b, u, w and d
    const AbstractVector*  vecs[] = { &a, &b, &u, &w };
    AbstractVector*  targ_vecs[] = { &d };
    apply_op( trice_scal_op, 4, vecs, 1, targ_vecs, RTOp_REDUCT_OBJ_NULL );
    // Destroy the operator and clean up memory
    RTOp_TOp_trice_diag_scal_destroy(&trice_scal_op);
}
```

The above constructor and destructor are declared in the generated header file and are automatically implemented in the source file by the script. The above code snippet uses the C++ vector interface `AbstractVector` that is included in the example code. The `apply_op(...)` function simply calls the `apply_op(...)` method on the first `vecs[0]` object. Note that the order of the vector arguments `a`, `b`, `u` and `w` matches the order defined in (38). The ordering of the vector arguments must match and this order is determined by the developer that created the RTOp subclass.

Example reduction operator

The next example operator we consider is the reduction operation in (8.2). First we rewrite the operation in generic standard form as:

$$\alpha \leftarrow \{\max \alpha \ : \ v^0 + \alpha v^1 \geq \beta\}, \tag{39}$$

This reduction operation is more complex than the previous example transformation operation and requires a little more thought. If we can assume that $v_i^0 \geq \beta, \ \text{for } i = 1 \ldots n$ before going in, what the above reduction is really asking for is the minimum α_i where:

$$\alpha_i = \max((\beta - v_i^0)/v_i^1, 0).$$

The reduction operation (39) can then be reexpressed as:

$$\alpha \leftarrow \min\{\max((\beta - v_i^0)/v_i^1, 0), \text{ for } i = 1 \ldots n\}.$$

This reduction operation requires the scalar operator data β (`beta`) and produces the scalar reduction object α (`alpha`). To make this operator work correctly, we must initialize the reduction object `alpha` to a very large value. In this implementation we will assume that `1e+200` will be larger than any reasonable values of the reduction. In general, whenever using `min(...)` for the reduction of intermediate reduction objects, we generally want to initialize the reduction object to some large value before performing the first reduction.

We will call this operator `ROp_max_feas_step` and the following is the interactive session with the `new_rtop.pl` script used to create the implementation files.

```
*******************************************
*** Create a new C RTOp operator subclass ***
*******************************************

1) What is the name of your operator subclass?
: ROp_max_feas_step

2) Is your operator coordinate invariant?
[y] or [n] : y

3) Give the number of nonmutable input vectors (vi, i=0...num_vecs-1)?
: 2

4) Give the number of mutable input/output vectors (zi, i=0...num_targ_vecs)?
: 0

5) Does your operator require extra data which is not in the input vectors?
[y] or [n] : y

Choose the structure of the data:
1: {index}
2: {value}
3: {value,index}
4: {value,value}
5: other
Choose 1-5? 2

Give name for {value} member?
: beta

6) Does your operator perform a reduction?
[y] or [n] : y
```

```
Choose the structure of the data:
1: {index}
2: {value}
3: {value,index}
4: {value,value}
5: other
Choose 1-5? 2

Give name for {value} member?
: alpha

6.a) Does the reduction object require nonzero initialization?
[y] or [n] : y

6.b) Give the initial values for the reduction object data:
alpha ? 1e+200

6.c) Choose the reduction of intermediate reduction objects:
1: sum{value,value}
2: min{value,value}
3: max{value,value}
4: other
Choose 1-4? 2

7.a) Does your element-wise operation(s) need temporary variables?
[y] or [n] : n

7.b) Give the C statement(s) for element-wise reduction operation?
   You can choose:
        Non-mutable operator data (don't change here)   : beta
        Non-mutable vector elements (don't change here) : v0, v1
        Element-wise reduction data (must be set here)  : alpha_ith
? alpha_ith = ( beta - v0 ) / v1;
? alpha_ith = max( alpha_ith, 0.0 );
?

The implementation files RTOp_ROp_max_feas_step.h and
RTOp_ROp_max_feas_step.c should be complete!
```

The code snippet that loops through the elements and performs the reduction operation is contained in the generated static function `RTOp_ROp_max_feas_step_apply_op(...)` and is shown below.

```
for( k = 0; k < sub_dim; ++k, v0_val += v0_val_s, v1_val += v1_val_s )
{
    // Element-wise reduction
    alpha_ith = ( (*beta) - (*v0_val) ) / (*v1_val);
    alpha_ith = max( alpha_ith, 0.0 );
    // Reduction of intermediates
    (*alpha) = min( (*alpha), alpha_ith );
}
```

Since this is a reduction operator, the ANA code must create the reduction target object before it is passed

into a vector object's `apply_op(...)` method. The follow code snippet shows how a ANA code might use this reduction operator and extract the value of the reduction.

```
#include "RTOp_ROp_max_feas_step.h"

...

RTOp_value_type max_feas_step( const AbstractVector& x, const AbstractVector& d
    ,const RTOp_value_type beta )
{
    // Create and initialize the instance data for the operator
    RTOp_RTOp        max_feas_step_op;
    RTOp_ROp_max_feas_step_construct(beta,&max_feas_step_op);
    // Create the reduction object
    RTOp_ReductTarget max_feas_step_reduct_obj;
    RTOp_reduct_obj_create(&max_feas_step_op,&max_feas_step_reduct_obj);
    // Apply the reduction operator to the existing vectors x and d
    const AbstractVector vecs[] { &x, &d };
    apply_op( max_feas_step_op, 2, vecs, 0, NULL, &max_feas_step_reduct_obj );
    // Extract the value from the reduction object
    RTOp_value_type  alpha = RTOp_ROp_max_feas_step_val(max_feas_step_reduct_obj);
    // Destroy the operator and clean up memory
    RTOp_ROp_max_feas_step_destroy(&max_feas_step_op);
    return alpha;
}
```

The above code snippet also uses the example C++ vector interface `AbstractVector` mentioned above.

That is really all there is to creating most new RTOp operators using the provided script. More details on the use of the script `new_rtop.pl` can be found in the help file `HowTo.CreateNewRTOpSubclass` at `WEBSITE`.

DISTRIBUTION:

1 MS 9018
Central Technical Files, 8940-2

2 MS 0899
Technical Library, 4916

2 MS 0619
Review & Approval Desk, 4916

2 MS 0123
LDRD Office, 1011

1 MS 0321
Bill Camp, 9200

1 MS 9003
Kenneth Washington, 8900

1 MS 1110
David Womble, 9210

1 MS 0370
Scott Mitchell, 9211

1 MS 0316
Sudip Dosanjh, 9233

1 MS 0836
Ray Finley, 6115

1 MS 1310
Steve Kempka, 9113

1 MS 0817
Jim Ang, 9224

1 MS 9159
Steve Thomas, 8962

1 MS 1110
Susan Rountree, 9214

1 MS 0382
Jim Stewart, 9143

1 MS 0828
Martin Pilch, 9133

1 MS 0318
Jennifer Nelson, 9209

The report is available from:
www.cs.sandia.gov/bartv/papers.html/SensSAND.pdf
and has been distributed to external
collaborators and staff members electronically
(as per the following electronic distribution
list).

ELECTRONIC DISTRIBUTION:

1 Jan Hesthaven
Division of Applied Mathematics
Brown University, Box F
182 George Street, no. 325
Providence, RI 02912

1 Lucas Wilcox
Division of Applied Mathematics
Brown University, Box F
182 George Street, no. 325
Providence, RI 02912

1 Matthias Heinkenschloss
Department of Computational and
Applied Mathematics - MS 134
Rice University
6100 S. Main Street
Houston, TX 77005 - 1892

1 Tim Warburton
Department of Computational and
Applied Mathematics - MS 134
Rice University
6100 S. Main Street
Houston, TX 77005 - 1892

1 David Keyes
Appl Phys and Appl Math
Columbia University
200 S. W. Mudd Bldg., MC 4701
500 W. 120th Street
New York, NY, 10027

1 Omar Ghattas
Carnegie Mellon University
5000 Forbes Ave., Porter Hall
Pittsburgh, PA 15213

1 Volkan Akcelik
Carnegie Mellon University
5000 Forbes Ave., Porter Hall
Pittsburgh, PA 15213

1 Judy Hill
Carnegie Mellon University
5000 Forbes Ave., Porter Hall
Pittsburgh, PA 15213

1 Larry Biegler
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

1 Martin Berggren
Box 120
S-751 04, Uppsala
Sweden

1 MS 0316
John Shadid, 9233

1 MS 0370
Tim Trucano, 9211

1 MS 1110
Bill Hart, 9215

1 MS 1110
Cindy Phillips, 9215

1 MS 0735
Sean McKenna, 6115

1 MS 0316
Brett Bader, 9233

1 MS 1110
Todd Coffey, 9233

1 MS 0380
Garth Reese, 9142

1 MS 0380
Manoj Bhardwaj, 9142

1 MS 0380
Kendal Pierson, 9142

1 MS 0370
Mike Eldred, 9211

1 MS 1111
Patrick Knupp, 9211

1 MS 0370
Laura Swiler, 9211

1 MS 0370
David Gay, 9211

1 MS 1111
Rich Lehoucq, 9214

1 MS 1110
Mike Heroux, 9214

1 MS 1110
Pavel Bochev, 9214

1 MS 1110
David Day, 9214

1 MS 1110
Ulrich Hetmaniuk, 9214

1 MS 0750
David Aldrich, 6116

1 MS 1110
Adrei Draganescu, 9211

1 MS 0817
Jim Ang, 9224

1 MS 1110
Andy Salinger, 9233

1 MS 0316
Roger Pawloski, 9233

1 MS 0316
Robert Hoekstra, 9233

1 MS 0316
Eric Phipps, 9233

1 MS 0316
Scott Hutchinson, 9233

1 MS 0321
Laurie Frink, 9212

1 MS 9159
Paul Boggs, 8962

1 MS 9159
Kevin Long, 8962