Ecosystem

Version 1.0.0

Generated by Doxygen 1.8.11

Contents

1 Introduction 1

1.1 Copyright Statement . . . 1

1.2 General Information . . . 2

2 Hierarchical Index 2

2.1 Class Hierarchy . . . 2

3 Class Index 4

3.1 Class List . . . 4

4 File Index 7

4.1 File List . . . 7

5 Class Documentation 8

5.1 AdsorptionReaction Class Reference . . . 8

5.1.1 Detailed Description . . . 12

5.1.2 Constructor & Destructor Documentation . . . 12

5.1.3 Member Function Documentation . . . 12

5.1.4 Member Data Documentation . . . 19

5.2 ARNOLDI_DATA Struct Reference . . . 21

5.2.1 Detailed Description . . . 22

5.2.2 Member Data Documentation . . . 22

5.3 Atom Class Reference . . . 23

5.3.1 Detailed Description . . . 25

5.3.2 Constructor & Destructor Documentation . . . 26

5.3.3 Member Function Documentation . . . 26

5.3.4 Member Data Documentation . . . 28

5.4 BACKTRACK_DATA Struct Reference . . . 29

5.4.1 Detailed Description . . . 30

5.4.2 Member Data Documentation . . . 30

5.5 BiCGSTAB_DATA Struct Reference . . . 31

5.5.1 Detailed Description . . . 32
5.5.2 Member Data Documentation . . . 32
5.6 CGS_DATA Struct Reference . . . 34
5.6.1 Detailed Description . . . 36
5.6.2 Member Data Documentation . . . 36
5.7 ChemisorptionReaction Class Reference . . . 38
5.7.1 Detailed Description . . . 41
5.7.2 Constructor & Destructor Documentation . . . 41
5.7.3 Member Function Documentation . . . 41
5.7.4 Member Data Documentation . . . 47
5.8 Document Class Reference . . . 47
5.8.1 Detailed Description . . . 49
5.8.2 Constructor & Destructor Documentation . . . 50
5.8.3 Member Function Documentation . . . 50
5.8.4 Member Data Documentation . . . 53
5.9 DOGFISH_DATA Struct Reference . . . 53
5.9.1 Detailed Description . . . 55
5.9.2 Member Data Documentation . . . 55
5.10 DOGFISH_PARAM Struct Reference . . . 57
5.10.1 Detailed Description . . . 57
5.10.2 Member Data Documentation . . . 57
5.11 Dove Class Reference . . . 58
5.11.1 Detailed Description . . . 62
5.11.2 Constructor & Destructor Documentation . . . 63
5.11.3 Member Function Documentation . . . 63
5.11.4 Member Data Documentation . . . 70
5.12 FINCH_DATA Struct Reference . . . 72
5.12.1 Detailed Description . . . 77
5.12.2 Member Data Documentation . . . 77
5.13 GCR_DATA Struct Reference . . . 85

5.13.1 Detailed Description . . . 87
5.13.2 Member Data Documentation . . . 87
5.14 GMRESLP_DATA Struct Reference . . . 89
5.14.1 Detailed Description . . . 90
5.14.2 Member Data Documentation . . . 90
5.15 GMRESR_DATA Struct Reference . . . 91
5.15.1 Detailed Description . . . 92
5.15.2 Member Data Documentation . . . 92
5.16 GMRESRP_DATA Struct Reference . . . 94
5.16.1 Detailed Description . . . 95
5.16.2 Member Data Documentation . . . 95
5.17 GPAST_DATA Struct Reference . . . 98
5.17.1 Detailed Description . . . 98
5.17.2 Member Data Documentation . . . 98
5.18 GSTA_DATA Struct Reference . . . 99
5.18.1 Detailed Description . . . 100
5.18.2 Member Data Documentation . . . 100
5.19 GSTA_OPT_DATA Struct Reference . . . 100
5.19.1 Detailed Description . . . 101
5.19.2 Member Data Documentation . . . 101
5.20 Header Class Reference . . . 102
5.20.1 Detailed Description . . . 104
5.20.2 Constructor & Destructor Documentation . . . 105
5.20.3 Member Function Documentation . . . 105
5.20.4 Member Data Documentation . . . 108
5.21 KeyValueMap Class Reference . . . 108
5.21.1 Detailed Description . . . 110
5.21.2 Constructor & Destructor Documentation . . . 110
5.21.3 Member Function Documentation . . . 110
5.21.4 Member Data Documentation . . . 112

5.22 KMS_DATA Struct Reference . . . 113
5.22.1 Detailed Description . . . 114
5.22.2 Member Data Documentation . . . 114
5.23 MAGPIE_DATA Struct Reference . . . 115
5.23.1 Detailed Description . . . 116
5.23.2 Member Data Documentation . . . 116
5.24 MassBalance Class Reference . . . 116
5.24.1 Detailed Description . . . 119
5.24.2 Constructor & Destructor Documentation . . . 119
5.24.3 Member Function Documentation . . . 119
5.24.4 Member Data Documentation . . . 122
5.25 MasterSpeciesList Class Reference . . . 123
5.25.1 Detailed Description . . . 124
5.25.2 Constructor & Destructor Documentation . . . 124
5.25.3 Member Function Documentation . . . 125
5.25.4 Member Data Documentation . . . 127
5.26 Matrix< T > Class Template Reference . . . 127
5.26.1 Detailed Description . . . 130
5.26.2 Constructor & Destructor Documentation . . . 130
5.26.3 Member Function Documentation . . . 130
5.26.4 Member Data Documentation . . . 136
5.27 MIXED_GAS Struct Reference . . . 136
5.27.1 Detailed Description . . . 137
5.27.2 Member Data Documentation . . . 137
5.28 Molecule Class Reference . . . 139
5.28.1 Detailed Description . . . 141
5.28.2 Constructor & Destructor Documentation . . . 141
5.28.3 Member Function Documentation . . . 142
5.28.4 Member Data Documentation . . . 145
5.29 MONKFISH_DATA Struct Reference . . . 146

5.29.1 Detailed Description . . . 148
5.29.2 Member Data Documentation . . . 148
5.30 MONKFISH_PARAM Struct Reference . . . 151
5.30.1 Detailed Description . . . 152
5.30.2 Member Data Documentation . . . 152
5.31 mSPD_DATA Struct Reference . . . 153
5.31.1 Detailed Description . . . 154
5.31.2 Member Data Documentation . . . 154
5.32 MultiligandAdsorption Class Reference . . . 154
5.32.1 Detailed Description . . . 157
5.32.2 Constructor & Destructor Documentation . . . 157
5.32.3 Member Function Documentation . . . 157
5.32.4 Member Data Documentation . . . 162
5.33 MultiligandChemisorption Class Reference . . . 164
5.33.1 Detailed Description . . . 166
5.33.2 Constructor & Destructor Documentation . . . 167
5.33.3 Member Function Documentation . . . 167
5.33.4 Member Data Documentation . . . 172
5.34 NUM_JAC_DATA Struct Reference . . . 173
5.34.1 Detailed Description . . . 174
5.34.2 Member Data Documentation . . . 174
5.35 OPTRANS_DATA Struct Reference . . . 174
5.35.1 Detailed Description . . . 174
5.35.2 Member Data Documentation . . . 175
5.36 PCG_DATA Struct Reference . . . 175
5.36.1 Detailed Description . . . 176
5.36.2 Member Data Documentation . . . 176
5.37 PeriodicTable Class Reference . . . 178
5.37.1 Detailed Description . . . 178
5.37.2 Constructor & Destructor Documentation . . . 178

5.37.3 Member Function Documentation . . . 179
5.37.4 Member Data Documentation . . . 179
5.38 PICARD_DATA Struct Reference . . . 179
5.38.1 Detailed Description . . . 180
5.38.2 Member Data Documentation . . . 180
5.39 PJFNK_DATA Struct Reference . . . 181
5.39.1 Detailed Description . . . 183
5.39.2 Member Data Documentation . . . 183
5.40 PURE_GAS Struct Reference . . . 187
5.40.1 Detailed Description . . . 187
5.40.2 Member Data Documentation . . . 188
5.41 QR_DATA Struct Reference . . . 188
5.41.1 Detailed Description . . . 189
5.41.2 Member Data Documentation . . . 189
5.42 Reaction Class Reference . . . 189
5.42.1 Detailed Description . . . 191
5.42.2 Constructor & Destructor Documentation . . . 191
5.42.3 Member Function Documentation . . . 191
5.42.4 Member Data Documentation . . . 193
5.43 SCOPSOWL_DATA Struct Reference . . . 194
5.43.1 Detailed Description . . . 196
5.43.2 Member Data Documentation . . . 196
5.44 SCOPSOWL_OPT_DATA Struct Reference . . . 200
5.44.1 Detailed Description . . . 201
5.44.2 Member Data Documentation . . . 201
5.45 SCOPSOWL_PARAM_DATA Struct Reference . . . 204
5.45.1 Detailed Description . . . 205
5.45.2 Member Data Documentation . . . 205
5.46 SHARK_DATA Struct Reference . . . 207
5.46.1 Detailed Description . . . 211

5.46.2 Member Data Documentation . . . 211
5.47 SKUA_DATA Struct Reference . . . 219
5.47.1 Detailed Description . . . 221
5.47.2 Member Data Documentation . . . 221
5.48 SKUA_OPT_DATA Struct Reference . . . 223
5.48.1 Detailed Description . . . 224
5.48.2 Member Data Documentation . . . 225
5.49 SKUA_PARAM Struct Reference . . . 227
5.49.1 Detailed Description . . . 228
5.49.2 Member Data Documentation . . . 228
5.50 SubHeader Class Reference . . . 229
5.50.1 Detailed Description . . . 230
5.50.2 Constructor & Destructor Documentation . . . 230
5.50.3 Member Function Documentation . . . 231
5.50.4 Member Data Documentation . . . 232
5.51 SYSTEM_DATA Struct Reference . . . 233
5.51.1 Detailed Description . . . 234
5.51.2 Member Data Documentation . . . 234
5.52 TRAJECTORY_DATA Struct Reference . . . 235
5.52.1 Member Data Documentation . . . 237
5.53 UI_DATA Struct Reference . . . 239
5.53.1 Detailed Description . . . 239
5.53.2 Member Data Documentation . . . 239
5.54 UnsteadyAdsorption Class Reference . . . 241
5.54.1 Detailed Description . . . 244
5.54.2 Constructor & Destructor Documentation . . . 244
5.54.3 Member Function Documentation . . . 244
5.54.4 Member Data Documentation . . . 251
5.55 UnsteadyReaction Class Reference . . . 252
5.55.1 Detailed Description . . . 255

5.55.2 Constructor & Destructor Documentation . . . 255
5.55.3 Member Function Documentation . . . 255
5.55.4 Member Data Documentation . . . 261
5.56 ValueTypePair Class Reference . . . 263
5.56.1 Detailed Description . . . 264
5.56.2 Constructor & Destructor Documentation . . . 264
5.56.3 Member Function Documentation . . . 264
5.56.4 Member Data Documentation . . . 265
5.57 yaml_cpp_class Class Reference . . . 266
5.57.1 Detailed Description . . . 266
5.57.2 Constructor & Destructor Documentation . . . 267
5.57.3 Member Function Documentation . . . 267
5.57.4 Member Data Documentation . . . 267
5.58 YamlWrapper Class Reference . . . 268
5.58.1 Detailed Description . . . 269
5.58.2 Constructor & Destructor Documentation . . . 270
5.58.3 Member Function Documentation . . . 270
5.58.4 Member Data Documentation . . . 272

6 File Documentation **272**

6.1 dogfish.h File Reference . . . 272
6.1.1 Detailed Description . . . 273
6.1.2 Function Documentation . . . 274
6.2 dove.h File Reference . . . 277
6.2.1 Detailed Description . . . 279
6.2.2 Macro Definition Documentation . . . 279
6.2.3 Enumeration Type Documentation . . . 279
6.2.4 Function Documentation . . . 281
6.3 eel.h File Reference . . . 286
6.3.1 Detailed Description . . . 287
6.3.2 Function Documentation . . . 287

6.4 egret.h File Reference . . . 287
6.4.1 Detailed Description . . . 289
6.4.2 Macro Definition Documentation . . . 289
6.4.3 Function Documentation . . . 290
6.5 error.h File Reference . . . 291
6.5.1 Detailed Description . . . 292
6.5.2 Macro Definition Documentation . . . 292
6.5.3 Enumeration Type Documentation . . . 292
6.5.4 Function Documentation . . . 294
6.6 finch.h File Reference . . . 294
6.6.1 Detailed Description . . . 296
6.6.2 Enumeration Type Documentation . . . 297
6.6.3 Function Documentation . . . 297
6.7 flock.h File Reference . . . 301
6.7.1 Detailed Description . . . 301
6.8 gsta_opt.h File Reference . . . 302
6.8.1 Detailed Description . . . 303
6.8.2 Macro Definition Documentation . . . 304
6.8.3 Function Documentation . . . 304
6.9 lark.h File Reference . . . 309
6.9.1 Detailed Description . . . 311
6.9.2 Macro Definition Documentation . . . 313
6.9.3 Enumeration Type Documentation . . . 313
6.9.4 Function Documentation . . . 313
6.10 macaw.h File Reference . . . 326
6.10.1 Detailed Description . . . 327
6.10.2 Macro Definition Documentation . . . 328
6.10.3 Function Documentation . . . 328
6.11 magpie.h File Reference . . . 328
6.11.1 Detailed Description . . . 330

6.11.2 Macro Definition Documentation . . . 330
6.11.3 Function Documentation . . . 331
6.12 mola.h File Reference . . . 338
6.12.1 Detailed Description . . . 339
6.12.2 Macro Definition Documentation . . . 345
6.12.3 Enumeration Type Documentation . . . 345
6.12.4 Function Documentation . . . 346
6.13 monkfish.h File Reference . . . 346
6.13.1 Detailed Description . . . 347
6.13.2 Function Documentation . . . 347
6.14 sandbox.h File Reference . . . 350
6.14.1 Detailed Description . . . 350
6.14.2 Function Documentation . . . 351
6.15 school.h File Reference . . . 351
6.15.1 Detailed Description . . . 351
6.16 scopsowl.h File Reference . . . 352
6.16.1 Detailed Description . . . 354
6.16.2 Macro Definition Documentation . . . 354
6.16.3 Function Documentation . . . 355
6.17 scopsowl_opt.h File Reference . . . 363
6.17.1 Detailed Description . . . 364
6.17.2 Function Documentation . . . 365
6.18 shark.h File Reference . . . 368
6.18.1 Detailed Description . . . 372
6.18.2 Macro Definition Documentation . . . 373
6.18.3 Typedef Documentation . . . 374
6.18.4 Enumeration Type Documentation . . . 374
6.18.5 Function Documentation . . . 375
6.19 skua.h File Reference . . . 392
6.19.1 Detailed Description . . . 393

6.19.2 Macro Definition Documentation . . . 394
6.19.3 Function Documentation . . . 394
6.20 skua_opt.h File Reference . . . 400
6.20.1 Detailed Description . . . 401
6.20.2 Function Documentation . . . 402
6.21 Trajectory.h File Reference . . . 405
6.21.1 Detailed Description . . . 406
6.21.2 Function Documentation . . . 407
6.22 ui.h File Reference . . . 408
6.22.1 Detailed Description . . . 410
6.22.2 Macro Definition Documentation . . . 410
6.22.3 Enumeration Type Documentation . . . 411
6.22.4 Function Documentation . . . 411
6.23 yaml_wrapper.h File Reference . . . 416
6.23.1 Detailed Description . . . 418
6.23.2 Typedef Documentation . . . 420
6.23.3 Enumeration Type Documentation . . . 420
6.23.4 Function Documentation . . . 421

Index **423**

1 Introduction

1.1 Copyright Statement

Copyright

1.2 General Information

The source code contained within the ecosystem project was designed as a standalone tool set for performing numerical modeling and data analyses associated with adsorption phenomena in both gaseous and aqueous systems. Many of the lower level tools are general enough to be applied to any system you desire to be modeled. Such algorithms included are Krylov subspace methods for linear systems and a Jacobian-Free Newton-Krylov method for non-linear systems. There is also a templated matrix object for generic matrix construction and modification. For specific information on each individual kernel, navigate through the class and file indices or table of contents.

Warning

Many of these algorithms may still be under development. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

2 Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

AdsorptionReaction 8

ChemisorptionReaction 38

UnsteadyAdsorption 241

ARNOLDI_DATA 21

Atom 23

BACKTRACK_DATA 29

BiCGSTAB_DATA 31

CGS_DATA 34

DOGFISH_DATA 53

DOGFISH_PARAM 57

Dove 58

FINCH_DATA 72

GCR_DATA 85

GMRESLP_DATA 89

GMRESR_DATA 91

GMRESRP_DATA 94

GPAST_DATA 98

GSTA_DATA 99

GSTA_OPT_DATA 100

KeyValueMap 108

KMS_DATA 113

MAGPIE_DATA 115

MassBalance 116

MasterSpeciesList 123

Matrix< T > 127

Matrix< double > 127

Matrix< double(*)(int i, const Matrix< double > &u, double t, const void *data)> 127

Matrix< int > 127

MIXED_GAS 136

Molecule 139

MONKFISH_DATA 146

MONKFISH_PARAM 151

mSPD_DATA 153

MultiligandAdsorption 154

MultiligandChemisorption 164

NUM_JAC_DATA 173

OPTRANS_DATA 174

PCG_DATA 175

PeriodicTable 178

PICARD_DATA 179

PJFNK_DATA 181

PURE_GAS 187

QR_DATA 188

Reaction 189

UnsteadyReaction 252

SCOPSOWL_DATA 194

SCOPSOWL_OPT_DATA 200

SCOPSOWL_PARAM_DATA 204

SHARK_DATA 207

SKUA_DATA 219

SKUA_OPT_DATA 223

SKUA_PARAM 227

SubHeader 229

Document 47

Header 102

SYSTEM_DATA 233

TRAJECTORY_DATA 235

UI_DATA 239

ValueTypePair 263

yaml_cpp_class 266

YamlWrapper 268

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

AdsorptionReaction
Adsorption Reaction Object 8

ARNOLDI_DATA
Data structure for the construction of the Krylov subspaces for a linear system 21

Atom
Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions) 23

BACKTRACK_DATA
Data structure for the implementation of Backtracking Linesearch 29

BiCGSTAB_DATA
Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems 31

CGS_DATA
Data structure for the implementation of the CGS algorithm for non-symmetric linear systems 34

ChemisorptionReaction
Chemisorption Reaction Object 38

Document
Object for the various documents in the yaml file 47

DOGFISH_DATA
Primary data structure for running the DOGFISH application 53

DOGFISH_PARAM
Data structure for species-specific parameters 57

Dove
Dynamic ODE-solver with Various Established methods (DOVE) object 58

FINCH_DATA
Data structure for the FINCH object 72

GCR_DATA
Data structure for the implementation of the GCR algorithm for non-symmetric linear systems 85

GMRESLP_DATA
Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning 89

GMRESR_DATA
Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GM↩ RESR) 91

GMRESRP_DATA
Data structure for the Restarted GMRES algorithm with Right Preconditioning 94

GPAST_DATA
GPAST Data Structure 98

GSTA_DATA
GSTA Data Structure 99

GSTA_OPT_DATA
Data structure used in the GSTA optimization routines 100

Header
Object for headers in a yaml document (inherits from SubHeader) 102

KeyValueMap
Key-Value-Type Map object creating a map of the KeyValuePair objects 108

KMS_DATA
Data structure for the implemenation of the Krylov Multi-Space (KMS) Method 113

MAGPIE_DATA
MAGPIE Data Structure 115

MassBalance
Mass Balance Object 116

MasterSpeciesList
Master Species List Object 123

Matrix< T >
Templated C++ Matrix Class Object (click Matrix to go to function definitions) 127

MIXED_GAS
Data structure holding information necessary for computing mixed gas properties 136

Molecule
C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions) 139

MONKFISH_DATA
Primary data structure for running MONKFISH 146

MONKFISH_PARAM
Data structure for species specific information and parameters 151

mSPD_DATA
MSPD Data Structure 153

MultiligandAdsorption
Multi-ligand Adsorption Reaction Object 154

MultiligandChemisorption
Multi-ligand Chemisorption Reaction Object 164

NUM_JAC_DATA
Data structure to form a numerical jacobian matrix with finite differences 173

OPTRANS_DATA
Data structure for implementation of linear operator transposition 174

PCG_DATA
Data structure for implementation of the PCG algorithms for symmetric linear systems 175

PeriodicTable
Class object that store a digitial copy of all Atom objects 178

PICARD_DATA
Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems 179

PJFNK_DATA
Data structure for the implementation of the PJFNK algorithm for non-linear systems 181

PURE_GAS
Data structure holding all the parameters for each pure gas spieces 187

QR_DATA
Data structure for the implementation of a QR solver given some invertable linear operator 188

Reaction
Reaction Object 189

SCOPSOWL_DATA
Primary data structure for SCOPSOWL simulations 194

SCOPSOWL_OPT_DATA
Data structure for the SCOPSOWL optmization routine 200

SCOPSOWL_PARAM_DATA
Data structure for the species' parameters in SCOPSOWL 204

SHARK_DATA
Data structure for SHARK simulations 207

SKUA_DATA
Data structure for all simulation information in SKUA 219

SKUA_OPT_DATA
Data structure for the SKUA Optimization Routine 223

SKUA_PARAM
Data structure for species' parameters in SKUA 227

SubHeader
Object for the Lowest level of Header for the yaml_wrapper 229

SYSTEM_DATA
System Data Structure 233

TRAJECTORY_DATA 235

UI_DATA
Data structure holding the UI arguments 239

UnsteadyAdsorption
Unsteady Adsorption Reaction Object 241

UnsteadyReaction
Unsteady Reaction Object (inherits from Reaction) 252

ValueTypePair
Value-Type Pair object to recognize data type of a string that was read 263

yaml_cpp_class
Primary object used when reading and digitally storing yaml files 266

YamlWrapper
Object for the entire yaml file holding all documents, header, sub-headers, keys, and values 268

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

dogfish.h
Diffusion Object Governing Fiber Interior Sorption History 272

dove.h
Dynamic ODE solver with Various Established methods 277

eel.h
Easy-access Element Library 286

egret.h
Estimation of Gas-phase pRopErTies 287

error.h
All error types are defined here 291

finch.h
Flux-limiting Implicit Non-oscillatory Conservative High-resolution scheme 294

flock.h
FundamentaL Off-gas Collection of Kernels 301

gsta_opt.h
Generalized Statistical Thermodynamic Adsorption (GSTA) Optimization Routine 302

lark.h
Linear Algebra Residual Kernels 309

macaw.h
MAtrix CAlculation Workspace 326

magpie.h
Multicomponent Adsorption Generalized Procedure for Isothermal Equilibria 328

mola.h
Molecule Object Library from Atoms 338

monkfish.h
Multi-fiber wOven Nest Kernel For Interparticle Sorption History 346

sandbox.h
Coding Test Area 350

school.h
Seawater Codes from a Highly Object-Oriented Library 351

scopsowl.h
Simultaneously Coupled Objects for Pore and Surface diffusion Operations With Linear systems 352

scopsowl_opt.h
Optimization Routine for Surface Diffusivities in SCOPSOWL 363

shark.h
Speciation-object Hierarchy for Adsorption Reactions and Kinetics 368

skua.h
Surface Kinetics for Uptake by Adsorption 392

skua_opt.h
Optimization Routine for the SKUA Model 400

Trajectory.h
Single Particle Trajectory Analysis for Magnetic Filtration 405

ui.h
User Interface for Ecosystem 408

yaml_wrapper.h
C++ Wrapper for the C-YAML Library 416

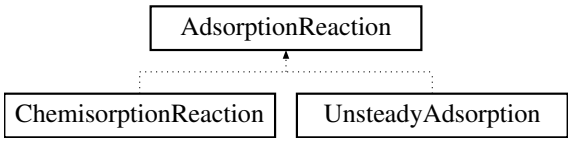
5 Class Documentation

5.1 AdsorptionReaction Class Reference

Adsorption Reaction Object.

`#include <shark.h>`

Inheritance diagram for AdsorptionReaction:



Public Member Functions

- AdsorptionReaction ()

  Default Constructor.
- ~AdsorptionReaction ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int n)

  Function to call the initialization of objects sequentially.
- void Display_Info ()

  Display the adsorption reaction information (PLACE HOLDER)
- void modifyDeltas (MassBalance &mbo)

  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

  Find and set the adsorbed species indices for each reaction object.
- int checkAqueousIndices ()

  Function to check and report errors in the aqueous species indices.
- void setActivityModelInfo (int(*act)(const Matrix< double > &logq, Matrix< double > &activity, const void *data), const void *act_data)

  Function to set the surface activity model and data pointer.
- void setAqueousIndex (int rxn_i, int species_i)

  Set the primary aqueous species index for the ith reaction.
- int setAqueousIndexAuto ()

  Automatically sets the primary aqueous species index based on reactions.
- void setActivityEnum (int act)

  Set the surface activity enum value.
- void setMolarFactor (int rxn_i, double m)

  Set the molar factor for the ith reaction (mol/mol)
- void setVolumeFactor (int i, double v)

  Set the ith volume factor for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

  Set the ith area factor for the species list (m^2/mol)
- void setSpecificArea (double a)

  Set the specific area for the adsorbent (m^2/kg)
- void setSpecificMolality (double a)

  Set the specific molality for the adsorbent (mol/kg)
- void setSurfaceCharge (double c)

  Set the surface charge of the uncomplexed ligands.
- void setTotalMass (double m)

  Set the total mass of the adsorbent (kg)
- void setTotalVolume (double v)

  Set the total volume of the system (L)
- void setAreaBasisBool (bool opt)

  Set the basis boolean directly.
- void setSurfaceChargeBool (bool opt)

  Set the boolean for inclusion of surface charging.
- void setBasis (std::string option)

  Set the basis of the adsorption problem from the given string arg.
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent to the given string.
- void setChargeDensityValue (double a)

  Set the value of the charge density parameter to a (C/m^2)

- void setIonicStrengthValue (double a)

  Set the value of the ionic strength parameter to a (mol/L)
- void setActivities (Matrix< double > &x)

  Set the values of activities in the activity matrix.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- double calculateActiveFraction (const Matrix< double > &x)

  Calculates the fraction of the surface that is active and available.
- double calculateSurfaceChargeDensity (const Matrix< double > &x)

  Function to calculate the surface charge density based on concentrations.
- double calculateLangmuirMaxCapacity (int i)

  Calculates the theoretical maximum capacity for adsorption in reaction i.
- double calculateLangmuirEquParam (const Matrix< double > &x, const Matrix< double > &gama, int i)

  Calculates the equivalent Langmuir isotherm equilibrium parameter.
- double calculateLangmuirAdsorption (const Matrix< double > &x, const Matrix< double > &gama, int i)

  Calculates the equivalent Langmuir adsorption by forming the Langmuir-like parameters.
- double calculatePsi (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateAqueousChargeExchange (int i)

  Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int i)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_↩ perm, int i)

  Calculates the residual for the ith reaction in the system.
- Reaction & getReaction (int i)

  Return reference to the ith reaction object in the adsorption object.
- double getMolarFactor (int i)

  Get the ith reaction's molar factor for adsorption (mol/mol)
- double getVolumeFactor (int i)

  Get the ith volume factor (species not involved return zeros) (cm^3/mol)
- double getAreaFactor (int i)

  Get the ith area factor (species not involved return zeros) (m^2/mol)
- double getActivity (int i)

  Get the ith activity factor for the surface species.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getSpecificMolality ()

  Get the specific molality of the adsorbent (mol/kg)
- double getSurfaceCharge ()

  Get the surface charge of the adsorbent.
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)

- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- int getNumberRxns ()

  Get the number of reactions involved in the adsorption object.
- int getAdsorbIndex (int i)

  Get the index of the adsorbed species in the ith reaction.
- int getAqueousIndex (int i)

  Get the index of the primary aqueous species in the ith reaction.
- int getActivityEnum ()

  Return the enum representing the choosen activity function.
- bool isAreaBasis ()

  Returns true if we are in the Area Basis, False if in Molar Basis.
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getAdsorbentName ()

  Returns the name of the adsorbent as a string.

Protected Attributes

- MasterSpeciesList * List

  Pointer to the MasterSpeciesList object.
- int(* surface_activity)(const Matrix< double > &logq, Matrix< double > &activity, const void *data)

  Pointer to a surface activity model.
- const void * activity_data

  Pointer to the data structure needed for surface activities.
- int act_fun

  Enumeration of the activity function being used for the surface phase.
- std::vector< double > area_factors

  List of the van der Waals areas associated with surface species (m^2/mol)
- std::vector< double > volume_factors

  List of the van der Waals volumes of each surface species (cm^3/mol)
- std::vector< int > adsorb_index

  List of the indices for the adsorbed species in the reactions.
- std::vector< int > aqueous_index

  List of the indices for the primary aqueous species in the reactions.
- std::vector< double > molar_factor

  List of the number of ligands needed to form one mole of adsorption in each reaction.
- Matrix< double > activities

  List of the activities calculated by the activity model.
- double specific_area

  Specific surface area of the adsorbent (m^2/kg)
- double specific_molality

  Specific molality of the adsorbent - moles of ligand per kg sorbent (mol/kg)
- double surface_charge

Charge of the uncomplexed surface ligand species.

- double total_mass

  Total mass of the adsorbent in the system (kg)
- double total_volume

  Total volume of the system (L)
- double ionic_strength

  Ionic Strength of the system used to adjust equilibria constants (mol/L)
- double charge_density

  Surface charge density of the adsorbent used to adjust equilbria (C/m^ 2)
- int num_rxns

  Number of reactions involved in the adsorption equilibria.
- bool AreaBasis

  True = Adsorption on an area basis, False = Adsorption on a ligand basis.
- bool IncludeSurfCharge

  True = Includes surface charging corrections, False = Does not consider surface charge.
- std::string adsorbent_name

  Name of the adsorbent for this object.

Private Attributes

- std::vector< Reaction > ads_rxn

  List of reactions involved with adsorption.

5.1.1 Detailed Description

Adsorption Reaction Object.

C++ Object to handle data and functions associated with forumlating adsorption equilibrium reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 AdsorptionReaction::AdsorptionReaction ( )

Default Constructor.

5.1.2.2 AdsorptionReaction::~AdsorptionReaction ( )

Default Destructor.

5.1.3 Member Function Documentation

5.1.3.1 void AdsorptionReaction::Initialize_Object (MasterSpeciesList & *List,* int *n*)

Function to call the initialization of objects sequentially.

5.1.3.2 void AdsorptionReaction::Display_Info ( )

Display the adsorption reaction information (PLACE HOLDER)

5.1.3.3 void AdsorptionReaction::modifyDeltas (MassBalance & *mbo*)

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.1.3.4 int AdsorptionReaction::setAdsorbIndices ( )

Find and set the adsorbed species indices for each reaction object.

This function searches through the Reaction objects in AdsorptionReaction to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.1.3.5 int AdsorptionReaction::checkAqueousIndices ( )

Function to check and report errors in the aqueous species indices.

5.1.3.6 void AdsorptionReaction::setActivityModelInfo (int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void *data) *act*, const void * *act_data*)

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.1.3.7 void AdsorptionReaction::setAqueousIndex (int *rxn_i*, int *species_i*)

Set the primary aqueous species index for the ith reaction.

5.1.3.8 int AdsorptionReaction::setAqueousIndexAuto ( )

Automatically sets the primary aqueous species index based on reactions.

This function will go through all species and all reactions in the adsorption object and automatically set the primary aqueous species index based on the stoicheometry of the reaction. It will also check and make sure that the primary aqueous index species appears opposite of the adsorbed species in the reactions. Note: This function assumes that the adsorbed indices have already been set.

5.1.3.9 void AdsorptionReaction::setActivityEnum (int *act*)

Set the surface activity enum value.

5.1.3.10 void AdsorptionReaction::setMolarFactor (int *rxn_i*, double *m*)

Set the molar factor for the ith reaction (mol/mol)

5.1.3.11 void AdsorptionReaction::setVolumeFactor (int *i*, double *v*)

Set the ith volume factor for the species list (cm^3/mol)

5.1.3.12 void AdsorptionReaction::setAreaFactor (int *i,* double *a*)

Set the ith area factor for the species list (m^2/mol)

5.1.3.13 void AdsorptionReaction::setSpecificArea (double *a*)

Set the specific area for the adsorbent (m^2/kg)

5.1.3.14 void AdsorptionReaction::setSpecificMolality (double *a*)

Set the specific molality for the adsorbent (mol/kg)

5.1.3.15 void AdsorptionReaction::setSurfaceCharge (double *c*)

Set the surface charge of the uncomplexed ligands.

5.1.3.16 void AdsorptionReaction::setTotalMass (double *m*)

Set the total mass of the adsorbent (kg)

5.1.3.17 void AdsorptionReaction::setTotalVolume (double *v*)

Set the total volume of the system (L)

5.1.3.18 void AdsorptionReaction::setAreaBasisBool (bool *opt*)

Set the basis boolean directly.

5.1.3.19 void AdsorptionReaction::setSurfaceChargeBool (bool *opt*)

Set the boolean for inclusion of surface charging.

5.1.3.20 void AdsorptionReaction::setBasis (std::string *option*)

Set the basis of the adsorption problem from the given string arg.

5.1.3.21 void AdsorptionReaction::setAdsorbentName (std::string *name*)

Set the name of the adsorbent to the given string.

5.1.3.22 void AdsorptionReaction::setChargeDensityValue (double *a*)

Set the value of the charge density parameter to a (C/m^2)

5.1.3.23 void AdsorptionReaction::setIonicStrengthValue (double *a*)

Set the value of the ionic strength parameter to a (mol/L)

5.1.3.24 void AdsorptionReaction::setActivities (Matrix< double > & *x*)

Set the values of activities in the activity matrix.

5.1.3.25 void AdsorptionReaction::calculateAreaFactors ( )

Calculates the area factors used from the van der Waals volumes.

5.1.3.26 void AdsorptionReaction::calculateEquilibria (double *T*)

Calculates all equilibrium parameters as a function of temperature.

5.1.3.27 void AdsorptionReaction::setChargeDensity (const Matrix< double > & *x*)

Calculates and sets the current value of charge density.

5.1.3.28 void AdsorptionReaction::setIonicStrength (const Matrix< double > & *x*)

Calculates and sets the current value of ionic strength.

5.1.3.29 int AdsorptionReaction::callSurfaceActivity (const Matrix< double > & *x*)

Calls the activity model and returns an int flag for success or failure.

5.1.3.30 double AdsorptionReaction::calculateActiveFraction (const Matrix< double > & *x*)

Calculates the fraction of the surface that is active and available.

5.1.3.31 double AdsorptionReaction::calculateSurfaceChargeDensity (const Matrix< double > & *x*)

Function to calculate the surface charge density based on concentrations.

This function is used to calculate the surface charge density of the adsorbed species based on the charges and concentrations of the adsorbed species. The calculation is used to correct the adsorption equilibria constant based on a localized surface charge balance. This requires that you know the molality of the uncomplexed ligand species on the surface, as well as the specific surface area for the adsorbent.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.1.3.32 double AdsorptionReaction::calculateLangmuirMaxCapacity (int *i*)

Calculates the theoretical maximum capacity for adsorption in reaction i.

This function is used to calculate the current maximum capacity of a species for a given adsorption reaction using the concentrations and activities of other species in the system. You must pass the index of the reaction of interest. The index of the species of interest is determined from the adsorb_index object. Note: This is only true if the stoicheometry for the adsorbed species is 1.

Parameters

i	index of the reaction of interest for the adsorption object

5.1.3.33 double AdsorptionReaction::calculateLangmuirEquParam (const Matrix< double > & *x,* const Matrix< double > & *gama,* int *i*)

Calculates the equivalent Langmuir isotherm equilibrium parameter.

This function will take in the current aqueous activities and calculate an effective Langmuir adsorption parameter for use in determining the adsorption in the system. It uses the system temperature as well to calculate equilibrium. Note: This is only true if the stoicheometry for the adsorbed species is 1.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
i	index of the reaction of interest for the adsorption object

5.1.3.34 double AdsorptionReaction::calculateLangmuirAdsorption (const Matrix< double > & *x,* const Matrix< double > & *gama,* int *i*)

Calculates the equivalent Langmuir adsorption by forming the Langmuir-like parameters.

This function will use the calculateLangmuirMaxCapacity and calculateLangmuirEquParam functions to approximate the adsorption of the ith reaction given the concentration of aqueous species, activities, and temperature. Note: This is only true if the stoicheometry for the adsorbed species is 1.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
i	index of the reaction of interest for the adsorption object

5.1.3.35 double AdsorptionReaction::calculatePsi (double *sigma,* double *T,* double *I,* double *rel_epsilon*)

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.1.3.36 double AdsorptionReaction::calculateAqueousChargeExchange (int *i*)

Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.

This function will look at all aqueous species involved in the ith adsorption reaction and sum up their stoicheometries and charges to see what the net change in charge is caused by the adsorption of charged species in solution. It is then used to adjust or correct the equilibrium constant for the given adsorption reaction.

Parameters

i	index of the reaction of interest for the adsorption object

5.1.3.37 double AdsorptionReaction::calculateEquilibriumCorrection (double *sigma,* double *T,* double *I,* double *rel_epsilon,* int *i*)

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
i	index of the reaction of interest for the adsorption object

5.1.3.38 double AdsorptionReaction::Eval_Residual (const Matrix< double > & *x,* const Matrix< double > & *gama,* double *T,* double *rel_perm,* int *i*)

Calculates the residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.1.3.39 Reaction& AdsorptionReaction::getReaction (int *i*)

Return reference to the ith reaction object in the adsorption object.

5.1.3.40 double AdsorptionReaction::getMolarFactor (int *i*)

Get the ith reaction's molar factor for adsorption (mol/mol)

5.1.3.41 double AdsorptionReaction::getVolumeFactor (int *i*)

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

5.1.3.42 double AdsorptionReaction::getAreaFactor (int *i*)

Get the ith area factor (species not involved return zeros) (m^2/mol)

5.1.3.43 double AdsorptionReaction::getActivity (int *i*)

Get the ith activity factor for the surface species.

5.1.3.44 double AdsorptionReaction::getSpecificArea ( )

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.1.3.45 double AdsorptionReaction::getSpecificMolality ( )

Get the specific molality of the adsorbent (mol/kg)

5.1.3.46 double AdsorptionReaction::getSurfaceCharge ( )

Get the surface charge of the adsorbent.

5.1.3.47 double AdsorptionReaction::getBulkDensity ( )

Calculate and return bulk density of adsorbent in system (kg/L)

5.1.3.48 double AdsorptionReaction::getTotalMass ( )

Get the total mass of adsorbent in the system (kg)

5.1.3.49 double AdsorptionReaction::getTotalVolume ( )

Get the total volume of the system (L)

5.1.3.50 double AdsorptionReaction::getChargeDensity ( )

Get the value of the surface charge density (C/m^2)

5.1.3.51 double AdsorptionReaction::getIonicStrength ( )

Get the value of the ionic strength of solution (mol/L)

5.1.3.52 int AdsorptionReaction::getNumberRxns ( )

Get the number of reactions involved in the adsorption object.

5.1.3.53 int AdsorptionReaction::getAdsorbIndex (int *i*)

Get the index of the adsorbed species in the ith reaction.

5.1.3.54 int AdsorptionReaction::getAqueousIndex (int *i*)

Get the index of the primary aqueous species in the ith reaction.

5.1.3.55 int AdsorptionReaction::getActivityEnum ( )

Return the enum representing the choosen activity function.

5.1.3.56 bool AdsorptionReaction::isAreaBasis ( )

Returns true if we are in the Area Basis, False if in Molar Basis.

5.1.3.57 bool AdsorptionReaction::includeSurfaceCharge ( )

Returns true if we are considering surface charging during adsorption.

5.1.3.58 std::string AdsorptionReaction::getAdsorbentName ( )

Returns the name of the adsorbent as a string.

5.1.4 Member Data Documentation

5.1.4.1 MasterSpeciesList* AdsorptionReaction::List [protected]

Pointer to the MasterSpeciesList object.

5.1.4.2 int(* AdsorptionReaction::surface_activity) (const Matrix< double > &logq, Matrix< double > &activity, const void *data) [protected]

Pointer to a surface activity model.

This is a function pointer for a surface activity model. The function must accept the log of the surface concentrations as an argument (logq) and provide the activities for each species (activity). The pointer data is used to pass any additional arguments needed.

Parameters

logq	matrix of the log (base 10) of surface concentrations of all species
activity	matrix of activity coefficients for all surface species (must be overriden)
data	pointer to a data structure needed to calculate activities

5.1.4.3 const void* AdsorptionReaction::activity_data [protected]

Pointer to the data structure needed for surface activities.

5.1.4.4 int AdsorptionReaction::act_fun [protected]

Enumeration of the activity function being used for the surface phase.

5.1.4.5 std::vector<double> AdsorptionReaction::area_factors [protected]

List of the van der Waals areas associated with surface species (m^2/mol)

5.1.4.6 std::vector<double> AdsorptionReaction::volume_factors [protected]

List of the van der Waals volumes of each surface species (cm^3/mol)

5.1.4.7 std::vector<int> AdsorptionReaction::adsorb_index [protected]

List of the indices for the adsorbed species in the reactions.

5.1.4.8 std::vector<int> AdsorptionReaction::aqueous_index [protected]

List of the indices for the primary aqueous species in the reactions.

5.1.4.9 std::vector<double> AdsorptionReaction::molar_factor [protected]

List of the number of ligands needed to form one mole of adsorption in each reaction.

5.1.4.10 Matrix<double> AdsorptionReaction::activities [protected]

List of the activities calculated by the activity model.

5.1.4.11 double AdsorptionReaction::specific_area [protected]

Specific surface area of the adsorbent (m^2/kg)

5.1.4.12 double AdsorptionReaction::specific_molality [protected]

Specific molality of the adsorbent - moles of ligand per kg sorbent (mol/kg)

5.1.4.13 double AdsorptionReaction::surface_charge [protected]

Charge of the uncomplexed surface ligand species.

5.1.4.14 double AdsorptionReaction::total_mass [protected]

Total mass of the adsorbent in the system (kg)

5.1.4.15 double AdsorptionReaction::total_volume [protected]

Total volume of the system (L)

5.1.4.16 double AdsorptionReaction::ionic_strength [protected]

Ionic Strength of the system used to adjust equilibria constants (mol/L)

5.1.4.17 double AdsorptionReaction::charge_density [protected]

Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

5.1.4.18 int AdsorptionReaction::num_rxns [protected]

Number of reactions involved in the adsorption equilibria.

5.1.4.19 bool AdsorptionReaction::AreaBasis [protected]

True = Adsorption on an area basis, False = Adsorption on a ligand basis.

5.1.4.20 bool AdsorptionReaction::IncludeSurfCharge [protected]

True = Includes surface charging corrections, False = Does not consider surface charge.

5.1.4.21 std::string AdsorptionReaction::adsorbent_name [protected]

Name of the adsorbent for this object.

5.1.4.22 std::vector<Reaction> AdsorptionReaction::ads_rxn [private]

List of reactions involved with adsorption.

The documentation for this class was generated from the following file:

- shark.h

5.2 ARNOLDI_DATA Struct Reference

Data structure for the construction of the Krylov subspaces for a linear system.

```
#include <lark.h>
```

Public Attributes

- int k

  Desired size of the Krylov subspace.
- int iter

  Actual size of the Krylov subspace.
- double beta

  Normalization parameter.
- double hp1

  Additional row element of H (separate storage for holding)
- bool Output = true

  True = print messages to console.
- std::vector< Matrix< double > > Vk

  (N) x (k) orthonormal vector basis stored as a vector of column matrices
- Matrix< double > Hkp1

  (k+1) x (k) upper Hessenberg matrix
- Matrix< double > yk

  (k) x (1) vector search direction
- Matrix< double > e1

  (k) x (1) orthonormal vector with 1 in first position
- Matrix< double > w

  (N) x (1) interim result of the matrix_vector multiplication
- Matrix< double > v

  (N) x (1) holding cell for the column entries of Vk and other interims
- Matrix< double > sum

  (N) x (1) running sum of subspace vectors for use in altering w

5.2.1 Detailed Description

Data structure for the construction of the Krylov subspaces for a linear system.

C-style object used in conjunction with the Arnoldi algorithm to construct an orthonormal basis and upper Hessenberg representation of a given linear operator. This is used to solve a linear system both iteratively (i.e., in conjunction with GMRESLP) and directly (i.e., in conjunction with FOM). Alternatively, you can just store the factorized components for later use in another routine.

5.2.2 Member Data Documentation

5.2.2.1 int ARNOLDI_DATA::k

Desired size of the Krylov subspace.

5.2.2.2 int ARNOLDI_DATA::iter

Actual size of the Krylov subspace.

5.2.2.3 double ARNOLDI_DATA::beta

Normalization parameter.

5.2.2.4 double ARNOLDI_DATA::hp1

Additional row element of H (separate storage for holding)

5.2.2.5 bool ARNOLDI_DATA::Output = true

True = print messages to console.

5.2.2.6 std::vector< Matrix<double> > ARNOLDI_DATA::Vk

(N) x (k) orthonormal vector basis stored as a vector of column matrices

5.2.2.7 Matrix<double> ARNOLDI_DATA::Hkp1

(k+1) x (k) upper Hessenberg matrix

5.2.2.8 Matrix<double> ARNOLDI_DATA::yk

(k) x (1) vector search direction

5.2.2.9 Matrix<double> ARNOLDI_DATA::e1

(k) x (1) orthonormal vector with 1 in first position

5.2.2.10 Matrix<double> ARNOLDI_DATA::w

(N) x (1) interim result of the matrix_vector multiplication

5.2.2.11 Matrix<double> ARNOLDI_DATA::v

(N) x (1) holding cell for the column entries of Vk and other interims

5.2.2.12 Matrix<double> ARNOLDI_DATA::sum

(N) x (1) running sum of subspace vectors for use in altering w

The documentation for this struct was generated from the following file:

- lark.h

5.3 Atom Class Reference

Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions)

```
#include <eel.h>
```

Public Member Functions

- Atom ()

  Default Constructor.
- ~Atom ()

  Default Destructor.
- Atom (std::string Name)

  Constructor by Atom Name.
- Atom (int number)

  Constructor by Atomic number.
- void Register (std::string Symbol)

  Register an atom object by symbol.
- void Register (int number)

  Register an atom object by number.
- void editAtomicWeight (double AW)

  Manually changes the atomic weight.
- void editOxidationState (int state)

  Manually changes the oxidation state.
- void editProtons (int proton)

  Manually changes the number of protons.
- void editNeutrons (int neutron)

  Manually changes the number of neutrons.
- void editElectrons (int electron)

  Manually changes the number of electrons.
- void editValence (int val)

  Manually changes the number of valence electrons.
- void editRadii (double r)

  Manually changes the van der Waals radii.
- void removeProton ()

  Manually removes 1 proton and adjusts weight.
- void removeNeutron ()

  Manually removes 1 neutron and adjusts weight.
- void removeElectron ()

  Manually removes 1 electron from valence.
- double AtomicWeight ()

  Returns the current atomic weight (g/mol)
- int OxidationState ()

  Returns the current oxidation state.
- int Protons ()

  Returns the current number of protons.
- int Neutrons ()

  Returns the current number of neutrons.
- int Electrons ()

  Returns the current number of electrons.
- int BondingElectrons ()

  Returns the number of electrons available for bonding.
- double AtomicRadii ()

  Returns the current van der Waals radii (in angstroms)
- std::string AtomName ()

  Returns the name of the atom.
- std::string AtomSymbol ()

Returns the symbol of the atom.

- std::string AtomCategory ()

  Returns the category of the atom.

- std::string AtomState ()

  Returns the state of the atom.

- int AtomicNumber ()

  Returns the atomic number of the atom.

- void DisplayInfo ()

  Displays Atom information to console.

Protected Attributes

- double atomic_weight

  Holds the atomic weight of the atom.

- int oxidation_state

  Holds the oxidation state of the atom.

- int protons

  Holds the number of protons in the atom.

- int neutrons

  Holds the number of neutrons in the atom.

- int electrons

  Holds the number of electrons in the atom.

- int valence_e

  Holds the number of valence electrons in the atom.

- double atomic_radii

  Holds the van der Waals radii of the element (in angstroms)

Private Attributes

- std::string Name

  Holds the name of the atom.

- std::string Symbol

  Holds the atomic symbol for the atom.

- std::string Category

  Holds the category of the atom (e.g., Alkali Metal)

- std::string NaturalState

  Holds the natural state of the atom (e.g., Gas)

- int atomic_number

  Holds the atomic number of the atom.

5.3.1 Detailed Description

Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions)

C++ class object holding data and functions associated with atoms. Objects can be registered at the time of object construction, or after declaring an Atom object. Registration can be done via the atomic symbol or atomic number. Valid atoms go from Hydrogen (1) to Ununoctium (118).

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Atom::Atom ( )

Default Constructor.

5.3.2.2 Atom::~Atom ( )

Default Destructor.

5.3.2.3 Atom::Atom (std::string *Name*)

Constructor by Atom Name.

5.3.2.4 Atom::Atom (int *number*)

Constructor by Atomic number.

5.3.3 Member Function Documentation

5.3.3.1 void Atom::Register (std::string *Symbol*)

Register an atom object by symbol.

5.3.3.2 void Atom::Register (int *number*)

Register an atom object by number.

5.3.3.3 void Atom::editAtomicWeight (double *AW*)

Manually changes the atomic weight.

5.3.3.4 void Atom::editOxidationState (int *state*)

Manually changes the oxidation state.

5.3.3.5 void Atom::editProtons (int *proton*)

Manually changes the number of protons.

5.3.3.6 void Atom::editNeutrons (int *neutron*)

Manually changes the number of neutrons.

5.3.3.7 void Atom::editElectrons (int *electron*)

Manually changes the number of electrons.

5.3.3.8 void Atom::editValence (int *val*)

Manually changes the number of valence electrons.

5.3.3.9 void Atom::editRadii (double *r*)

Manually changes the van der Waals radii.

5.3.3.10 void Atom::removeProton ( )

Manually removes 1 proton and adjusts weight.

5.3.3.11 void Atom::removeNeutron ( )

Manually removes 1 neutron and adjusts weight.

5.3.3.12 void Atom::removeElectron ( )

Manually removes 1 electron from valence.

5.3.3.13 double Atom::AtomicWeight ( )

Returns the current atomic weight (g/mol)

5.3.3.14 int Atom::OxidationState ( )

Returns the current oxidation state.

5.3.3.15 int Atom::Protons ( )

Returns the current number of protons.

5.3.3.16 int Atom::Neutrons ( )

Returns the current number of neutrons.

5.3.3.17 int Atom::Electrons ( )

Returns the current number of electrons.

5.3.3.18 int Atom::BondingElectrons ( )

Returns the number of electrons available for bonding.

5.3.3.19 double Atom::AtomicRadii ( )

Returns the current van der Waals radii (in angstroms)

5.3.3.20 std::string Atom::AtomName ( )

Returns the name of the atom.

5.3.3.21 std::string Atom::AtomSymbol ( )

Returns the symbol of the atom.

5.3.3.22 std::string Atom::AtomCategory ( )

Returns the category of the atom.

5.3.3.23 std::string Atom::AtomState ( )

Returns the state of the atom.

5.3.3.24 int Atom::AtomicNumber ( )

Returns the atomic number of the atom.

5.3.3.25 void Atom::DisplayInfo ( )

Displays Atom information to console.

5.3.4 Member Data Documentation

5.3.4.1 double Atom::atomic_weight [protected]

Holds the atomic weight of the atom.

5.3.4.2 int Atom::oxidation_state [protected]

Holds the oxidation state of the atom.

5.3.4.3 int Atom::protons [protected]

Holds the number of protons in the atom.

5.3.4.4 int Atom::neutrons [protected]

Holds the number of neutrons in the atom.

5.3.4.5 int Atom::electrons [protected]

Holds the number of electrons in the atom.

5.3.4.6 int Atom::valence_e [protected]

Holds the number of valence electrons in the atom.

5.3.4.7 double Atom::atomic_radii [protected]

Holds the van der Waals radii of the element (in angstroms)

5.3.4.8 std::string Atom::Name [private]

Holds the name of the atom.

5.3.4.9 std::string Atom::Symbol [private]

Holds the atomic symbol for the atom.

5.3.4.10 std::string Atom::Category [private]

Holds the category of the atom (e.g., Alkali Metal)

5.3.4.11 std::string Atom::NaturalState [private]

Holds the natural state of the atom (e.g., Gas)

5.3.4.12 int Atom::atomic_number [private]

Holds the atomic number of the atom.

The documentation for this class was generated from the following file:

- eel.h

5.4 BACKTRACK_DATA Struct Reference

Data structure for the implementation of Backtracking Linesearch.

`#include <lark.h>`

Public Attributes

- int fun_call = 0
  Number of function calls made during line search.
- double alpha = 1e-4
  Scaling parameter for determination of search step size.
- double rho = 0.1
  Scaling parameter for to change step size by.
- double lambdaMin =DBL_EPSILON
  Smallest allowable step length.
- double normFkp1
  New residual norm of the Newton step.
- bool constRho = false
  True = use a constant value for rho.
- Matrix< double > Fk
  Old residual vector of the Newton step.
- Matrix< double > xk
  Old solution vector of the Newton step.

5.4.1 Detailed Description

Data structure for the implementation of Backtracking Linesearch.

C-style object used in conjunction with the Backtracking Linesearch algorithm to smooth out convergence of Netwon based iterative methods for non-linear systems of equations. The actual algorithm has been separated from the interior of the Newton method so that it can be included in any future Newton based iterative methods being developed.

5.4.2 Member Data Documentation

5.4.2.1 int BACKTRACK_DATA::fun_call = 0

Number of function calls made during line search.

5.4.2.2 double BACKTRACK_DATA::alpha = 1e-4

Scaling parameter for determination of search step size.

5.4.2.3 double BACKTRACK_DATA::rho = 0.1

Scaling parameter for to change step size by.

5.4.2.4 double BACKTRACK_DATA::lambdaMin =DBL_EPSILON

Smallest allowable step length.

5.4.2.5 double BACKTRACK_DATA::normFkp1

New residual norm of the Newton step.

5.4.2.6 bool BACKTRACK_DATA::constRho = false

True = use a constant value for rho.

5.4.2.7 Matrix<double> BACKTRACK_DATA::Fk

Old residual vector of the Newton step.

5.4.2.8 Matrix<double> BACKTRACK_DATA::xk

Old solution vector of the Newton step.

The documentation for this struct was generated from the following file:

- lark.h

5.5 BiCGSTAB_DATA Struct Reference

Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems.

`#include <lark.h>`

Public Attributes

- int maxit = 0
  Maximum allowable iterations - default = min(2∗vector_size,1000)
- int iter = 0
  Actual number of iterations.
- bool breakdown
  Boolean to determine if the method broke down.
- double alpha
  Step size parameter for next solution.
- double beta
  Step size parameter for search direction.
- double rho
  Scaling parameter for alpha and beta.
- double rho_old
  Previous scaling parameter for alpha and beta.
- double omega
  Scaling parameter and additional step length.
- double omega_old
  Previous scaling parameter and step length.
- double tol_rel = 1e-6
  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6
  Absolution tolerance for convergence - default = 1e-6.
- double res
  Absolute residual norm.
- double relres
  Relative residual norm.
- double relres_base
  Initial residual norm.
- double bestres
  Best found residual norm.
- bool Output = true
  True = print messages to console.
- Matrix< double > x
  Current solution to the linear system.
- Matrix< double > bestx
  Best found solution to the linear system.
- Matrix< double > r
  Residual vector for the linear system.
- Matrix< double > r0
  Initial residual vector.
- Matrix< double > v
  Search direction for p.

- Matrix< double > p
  Search direction for updating.
- Matrix< double > y
  Preconditioned search direction.
- Matrix< double > s
  Residual updating vector.
- Matrix< double > z
  Preconditioned residual updating vector.
- Matrix< double > t
  Search direction for resdidual updates.

5.5.1 Detailed Description

Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems.

C-style object used in conjunction with the Bi-Conjugate Gradient STABalized (BiCGSTAB) algorithm to solve a linear system of equations. This algorithm is generally more efficient than any GMRES or GCR variant, but may not always reduce the residual at each step. However, if used with preconditioning, then this algorithm is very efficient, especially when used for solving grid-based linear systems.

5.5.2 Member Data Documentation

5.5.2.1 int BiCGSTAB_DATA::maxit = 0

Maximum allowable iterations - default = min(2∗vector_size,1000)

5.5.2.2 int BiCGSTAB_DATA::iter = 0

Actual number of iterations.

5.5.2.3 bool BiCGSTAB_DATA::breakdown

Boolean to determine if the method broke down.

5.5.2.4 double BiCGSTAB_DATA::alpha

Step size parameter for next solution.

5.5.2.5 double BiCGSTAB_DATA::beta

Step size parameter for search direction.

5.5.2.6 double BiCGSTAB_DATA::rho

Scaling parameter for alpha and beta.

5.5.2.7 double BiCGSTAB_DATA::rho_old

Previous scaling parameter for alpha and beta.

5.5.2.8 double BiCGSTAB_DATA::omega

Scaling parameter and additional step length.

5.5.2.9 double BiCGSTAB_DATA::omega_old

Previous scaling parameter and step length.

5.5.2.10 double BiCGSTAB_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.5.2.11 double BiCGSTAB_DATA::tol_abs = 1e-6

Absolution tolerance for convergence - default = 1e-6.

5.5.2.12 double BiCGSTAB_DATA::res

Absolute residual norm.

5.5.2.13 double BiCGSTAB_DATA::relres

Relative residual norm.

5.5.2.14 double BiCGSTAB_DATA::relres_base

Initial residual norm.

5.5.2.15 double BiCGSTAB_DATA::bestres

Best found residual norm.

5.5.2.16 bool BiCGSTAB_DATA::Output = true

True = print messages to console.

5.5.2.17 Matrix<double> BiCGSTAB_DATA::x

Current solution to the linear system.

5.5.2.18 Matrix<double> BiCGSTAB_DATA::bestx

Best found solution to the linear system.

5.5.2.19 Matrix<double> BiCGSTAB_DATA::r

Residual vector for the linear system.

5.5.2.20 Matrix<double> BiCGSTAB_DATA::r0

Initial residual vector.

5.5.2.21 Matrix<double> BiCGSTAB_DATA::v

Search direction for p.

5.5.2.22 Matrix<double> BiCGSTAB_DATA::p

Search direction for updating.

5.5.2.23 Matrix<double> BiCGSTAB_DATA::y

Preconditioned search direction.

5.5.2.24 Matrix<double> BiCGSTAB_DATA::s

Residual updating vector.

5.5.2.25 Matrix<double> BiCGSTAB_DATA::z

Preconditioned residual updating vector.

5.5.2.26 Matrix<double> BiCGSTAB_DATA::t

Search direction for resdidual updates.

The documentation for this struct was generated from the following file:

- lark.h

5.6 CGS_DATA Struct Reference

Data structure for the implementation of the CGS algorithm for non-symmetric linear systems.

`#include <lark.h>`

Public Attributes

- int maxit = 0

  Maximum allowable iterations - default = min(2∗vector_size,1000)
- int iter = 0

  Actual number of iterations.
- bool breakdown

  Boolean to determine if the method broke down.
- double alpha

  Step size parameter for next solution.
- double beta

  Step size parameter for search direction.
- double rho

  Scaling parameter for alpha and beta.
- double sigma

  Scaling parameter and additional step length.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolution tolerance for convergence - default = 1e-6.
- double res

  Absolute residual norm.
- double relres

  Relative residual norm.
- double relres_base

  Initial residual norm.
- double bestres

  Best found residual norm.
- bool Output = true

  True = print messages to console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual vector for the linear system.
- Matrix< double > r0

  Initial residual vector.
- Matrix< double > u

  Search direction for v.
- Matrix< double > w

  Updates sigma and u.
- Matrix< double > v

  Search direction for x.
- Matrix< double > p

  Preconditioning result for w, z, and matvec for Ax.
- Matrix< double > c

  Holds the matvec result between A and p.
- Matrix< double > z

  Full search direction for x.

5.6.1 Detailed Description

Data structure for the implementation of the CGS algorithm for non-symmetric linear systems.

C-style object to be used in conjunction with the Conjugate Gradient Squared (CGS) algorithm to solve linear systems of equations. This algorithm is slightly less computational work than BiCGSTAB, but is much less stable. As a result, I do not recommend using this algorithm unless you also use some form of preconditioning.

5.6.2 Member Data Documentation

5.6.2.1 int CGS_DATA::maxit = 0

Maximum allowable iterations - default = min(2∗vector_size,1000)

5.6.2.2 int CGS_DATA::iter = 0

Actual number of iterations.

5.6.2.3 bool CGS_DATA::breakdown

Boolean to determine if the method broke down.

5.6.2.4 double CGS_DATA::alpha

Step size parameter for next solution.

5.6.2.5 double CGS_DATA::beta

Step size parameter for search direction.

5.6.2.6 double CGS_DATA::rho

Scaling parameter for alpha and beta.

5.6.2.7 double CGS_DATA::sigma

Scaling parameter and additional step length.

5.6.2.8 double CGS_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.6.2.9 double CGS_DATA::tol_abs = 1e-6

Absolution tolerance for convergence - default = 1e-6.

5.6.2.10 double CGS_DATA::res

Absolute residual norm.

5.6.2.11 double CGS_DATA::relres

Relative residual norm.

5.6.2.12 double CGS_DATA::relres_base

Initial residual norm.

5.6.2.13 double CGS_DATA::bestres

Best found residual norm.

5.6.2.14 bool CGS_DATA::Output = true

True = print messages to console.

5.6.2.15 Matrix<double> CGS_DATA::x

Current solution to the linear system.

5.6.2.16 Matrix<double> CGS_DATA::bestx

Best found solution to the linear system.

5.6.2.17 Matrix<double> CGS_DATA::r

Residual vector for the linear system.

5.6.2.18 Matrix<double> CGS_DATA::r0

Initial residual vector.

5.6.2.19 Matrix<double> CGS_DATA::u

Search direction for v.

5.6.2.20 Matrix<double> CGS_DATA::w

Updates sigma and u.

5.6.2.21 Matrix<double> CGS_DATA::v

Search direction for x.

5.6.2.22 Matrix<double> CGS_DATA::p

Preconditioning result for w, z, and matvec for Ax.

5.6.2.23 Matrix<double> CGS_DATA::c

Holds the matvec result between A and p.

5.6.2.24 Matrix<double> CGS_DATA::z

Full search direction for x.

The documentation for this struct was generated from the following file:

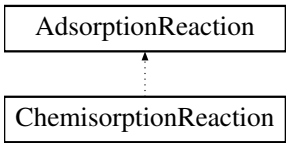
- lark.h

5.7 ChemisorptionReaction Class Reference

Chemisorption Reaction Object.

`#include <shark.h>`

Inheritance diagram for ChemisorptionReaction:



Public Member Functions

- ChemisorptionReaction ()
  Default Constructor.
- ~ChemisorptionReaction ()
  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int n)
  Function to call the initialization of objects sequentially.
- void Display_Info ()
  Display the adsorption reaction information.
- void modifyMBEdeltas (MassBalance &mbo)
  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()
  Find and set the adsorbed species indices for each reaction object.
- int setLigandIndex ()
  Find and set the ligand species index.
- int setDeltas ()
  Find and set all the delta values for the site balance.
- void setActivityModelInfo (int(*act)(const Matrix< double > &logq, Matrix< double > &activity, const void *data), const void *act_data)
  Function to set the surface activity model and data pointer.
- void setActivityEnum (int act)
  Set the surface activity enum value.
- void setDelta (int i, double v)

Set the ith delta factor for the site balance.

- void setVolumeFactor (int i, double v)

  Set the ith volume factor for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

  Set the ith area factor for the species list (m^2/mol)
- void setSpecificArea (double a)

  Set the specific area for the adsorbent (m^2/kg)
- void setSpecificMolality (double a)

  Set the specific molality for the adsorbent (mol/kg)
- void setTotalMass (double m)

  Set the total mass of the adsorbent (kg)
- void setTotalVolume (double v)

  Set the total volume of the system (L)
- void setSurfaceChargeBool (bool opt)

  Set the boolean for inclusion of surface charging.
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent to the given string.
- void setChargeDensityValue (double a)

  Set the value of the charge density parameter to a (C/m^2)
- void setIonicStrengthValue (double a)

  Set the value of the ionic strength parameter to a (mol/L)
- void setActivities (Matrix< double > &x)

  Set the values of activities in the activity matrix.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- double calculateSurfaceChargeDensity (const Matrix< double > &x)

  Function to calculate the surface charge density based on concentrations.
- double calculateElecticPotential (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateAqueousChargeExchange (int i)

  Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int i)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_RxnResidual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int i)

  Calculates the residual for the ith reaction in the system.
- double Eval_SiteBalanceResidual (const Matrix< double > &x)

  Calculates the residual for the overall site balance.
- Reaction & getReaction (int i)

  Return reference to the ith reaction object in the adsorption object.
- double getDelta (int i)

  Get the ith delta factor for the site balance.
- double getVolumeFactor (int i)

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

- double getAreaFactor (int i)

  Get the ith area factor (species not involved return zeros) (m^2/mol)

- double getActivity (int i)

  Get the ith activity factor for the surface species.

- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

- double getSpecificMolality ()

  Get the specific molality of the adsorbent (mol/kg)

- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)

- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)

- double getTotalVolume ()

  Get the total volume of the system (L)

- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)

- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)

- int getNumberRxns ()

  Get the number of reactions involved in the adsorption object.

- int getAdsorbIndex (int i)

  Get the index of the adsorbed species in the ith reaction.

- int getLigandIndex ()

  Get the index of the ligand species.

- int getActivityEnum ()

  Return the enum representing the choosen activity function.

- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.

- std::string getAdsorbentName ()

  Returns the name of the adsorbent as a string.

Protected Attributes

- int ligand_index

  Index of the ligand for all reactions.

- std::vector< double > Delta

  Vector of weights (i.e., deltas) used in the site balance.

Private Attributes

- std::vector< Reaction > ads_rxn

  List of reactions involved with adsorption.

Additional Inherited Members

5.7.1 Detailed Description

Chemisorption Reaction Object.

C++ Object to handle data and functions associated with forumlating adsorption equilibrium reactions in a aqueous mixture based on chemisorption mechanisms. Each unique surface in a system will require an instance of this structure. This is very similar to AdsorptionReaction, however, this will include a site balance residual that will allow us to consider protonation and deprotonation of the ligands.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 ChemisorptionReaction::ChemisorptionReaction ( )

Default Constructor.

5.7.2.2 ChemisorptionReaction::~ChemisorptionReaction ( )

Default Destructor.

5.7.3 Member Function Documentation

5.7.3.1 void ChemisorptionReaction::Initialize_Object (MasterSpeciesList & *List,* int *n*)

Function to call the initialization of objects sequentially.

5.7.3.2 void ChemisorptionReaction::Display_Info ( )

Display the adsorption reaction information.

5.7.3.3 void ChemisorptionReaction::modifyMBEdeltas (MassBalance & *mbo*)

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.7.3.4 int ChemisorptionReaction::setAdsorbIndices ( )

Find and set the adsorbed species indices for each reaction object.

This function searches through the Reaction objects in ChemisorptionReaction to find the adsorbed species and their indices to set that information in the adsorb_index structure. Function will return 0 if successful and -1 on a failure.

5.7.3.5 int ChemisorptionReaction::setLigandIndex ( )

Find and set the ligand species index.

This function searches through the Reaction objects in ChemisorptionReaction to find the ligand species and its index to set that information in the ligand_index structure. Function will return 0 if successful and -1 on a failure.

5.7.3.6 int ChemisorptionReaction::setDeltas ( )

Find and set all the delta values for the site balance.

This function searches through all reaction object instances for the stoicheometry of the ligand in each adsorption reaction. That stoicheometry serves as the basis for determining the site balance. NOTE: the delta for the ligand is set automatically in the setLigandIndex() function, so we can ignore that species. In addition, this function must be called after setLigandIndex() and setAdsorbIndices() are called and after the stoicheometry of each reaction has been determined.

5.7.3.7 void ChemisorptionReaction::setActivityModelInfo (int(∗)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data) *act,* const void ∗ *act_data*)

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.7.3.8 void ChemisorptionReaction::setActivityEnum (int *act*)

Set the surface activity enum value.

5.7.3.9 void ChemisorptionReaction::setDelta (int *i,* double *v*)

Set the ith delta factor for the site balance.

5.7.3.10 void ChemisorptionReaction::setVolumeFactor (int *i,* double *v*)

Set the ith volume factor for the species list (cm^3/mol)

5.7.3.11 void ChemisorptionReaction::setAreaFactor (int *i,* double *a*)

Set the ith area factor for the species list (m^2/mol)

5.7.3.12 void ChemisorptionReaction::setSpecificArea (double *a*)

Set the specific area for the adsorbent (m^2/kg)

5.7.3.13 void ChemisorptionReaction::setSpecificMolality (double *a*)

Set the specific molality for the adsorbent (mol/kg)

5.7.3.14 void ChemisorptionReaction::setTotalMass (double *m*)

Set the total mass of the adsorbent (kg)

5.7.3.15 void ChemisorptionReaction::setTotalVolume (double *v*)

Set the total volume of the system (L)

5.7.3.16 void ChemisorptionReaction::setSurfaceChargeBool (bool *opt*)

Set the boolean for inclusion of surface charging.

5.7.3.17 void ChemisorptionReaction::setAdsorbentName (std::string *name*)

Set the name of the adsorbent to the given string.

5.7.3.18 void ChemisorptionReaction::setChargeDensityValue (double *a*)

Set the value of the charge density parameter to a (C/m^2)

5.7.3.19 void ChemisorptionReaction::setIonicStrengthValue (double *a*)

Set the value of the ionic strength parameter to a (mol/L)

5.7.3.20 void ChemisorptionReaction::setActivities (Matrix< double > & *x*)

Set the values of activities in the activity matrix.

5.7.3.21 void ChemisorptionReaction::calculateAreaFactors ( )

Calculates the area factors used from the van der Waals volumes.

5.7.3.22 void ChemisorptionReaction::calculateEquilibria (double *T*)

Calculates all equilibrium parameters as a function of temperature.

5.7.3.23 void ChemisorptionReaction::setChargeDensity (const Matrix< double > & *x*)

Calculates and sets the current value of charge density.

5.7.3.24 void ChemisorptionReaction::setIonicStrength (const Matrix< double > & *x*)

Calculates and sets the current value of ionic strength.

5.7.3.25 int ChemisorptionReaction::callSurfaceActivity (const Matrix< double > & *x*)

Calls the activity model and returns an int flag for success or failure.

5.7.3.26 double ChemisorptionReaction::calculateSurfaceChargeDensity (const Matrix< double > & *x*)

Function to calculate the surface charge density based on concentrations.

This function is used to calculate the surface charge density of the adsorbed species based on the charges and concentrations of the adsorbed species. The calculation is used to correct the adsorption equilibria constant based on a localized surface charge balance. This requires that you know the molality of the uncomplexed ligand species on the surface, as well as the specific surface area for the adsorbent.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.7.3.27 double ChemisorptionReaction::calculateElecticPotential (double *sigma,* double *T,* double *I,* double *rel_epsilon*)

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.7.3.28 double ChemisorptionReaction::calculateAqueousChargeExchange (int *i*)

Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.

This function will look at all aqueous species involved in the ith adsorption reaction and sum up their stoicheometries and charges to see what the net change in charge is caused by the adsorption of charged species in solution. It is then used to adjust or correct the equilibrium constant for the given adsorption reaction.

Parameters

i	index of the reaction of interest for the adsorption object

5.7.3.29 double ChemisorptionReaction::calculateEquilibriumCorrection (double *sigma,* double *T,* double *I,* double *rel_epsilon,* int *i*)

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
i	index of the reaction of interest for the adsorption object

5.7.3.30 double ChemisorptionReaction::Eval_RxnResidual (const Matrix< double > & *x*, const Matrix< double > & *gama*, double *T*, double *rel_perm*, int *i*)

Calculates the residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.7.3.31 double ChemisorptionReaction::Eval_SiteBalanceResidual (const Matrix< double > & *x*)

Calculates the residual for the overall site balance.

This function will provide a system residual for the site/ligand balance for the Chemisorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.7.3.32 Reaction& ChemisorptionReaction::getReaction (int *i*)

Return reference to the ith reaction object in the adsorption object.

5.7.3.33 double ChemisorptionReaction::getDelta (int *i*)

Get the ith delta factor for the site balance.

5.7.3.34 double ChemisorptionReaction::getVolumeFactor (int *i*)

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

5.7.3.35 double ChemisorptionReaction::getAreaFactor (int *i*)

Get the ith area factor (species not involved return zeros) (m^2/mol)

5.7.3.36 double ChemisorptionReaction::getActivity (int *i*)

Get the ith activity factor for the surface species.

5.7.3.37 double ChemisorptionReaction::getSpecificArea ( )

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.7.3.38 double ChemisorptionReaction::getSpecificMolality ( )

Get the specific molality of the adsorbent (mol/kg)

5.7.3.39 double ChemisorptionReaction::getBulkDensity ( )

Calculate and return bulk density of adsorbent in system (kg/L)

5.7.3.40 double ChemisorptionReaction::getTotalMass ( )

Get the total mass of adsorbent in the system (kg)

5.7.3.41 double ChemisorptionReaction::getTotalVolume ( )

Get the total volume of the system (L)

5.7.3.42 double ChemisorptionReaction::getChargeDensity ( )

Get the value of the surface charge density (C/m^2)

5.7.3.43 double ChemisorptionReaction::getIonicStrength ( )

Get the value of the ionic strength of solution (mol/L)

5.7.3.44 int ChemisorptionReaction::getNumberRxns ( )

Get the number of reactions involved in the adsorption object.

5.7.3.45 int ChemisorptionReaction::getAdsorbIndex (int *i*)

Get the index of the adsorbed species in the ith reaction.

5.7.3.46 int ChemisorptionReaction::getLigandIndex ( )

Get the index of the ligand species.

5.7.3.47 int ChemisorptionReaction::getActivityEnum ( )

Return the enum representing the choosen activity function.

5.7.3.48 bool ChemisorptionReaction::includeSurfaceCharge ( )

Returns true if we are considering surface charging during adsorption.

5.7.3.49 std::string ChemisorptionReaction::getAdsorbentName ( )

Returns the name of the adsorbent as a string.

5.7.4 Member Data Documentation

5.7.4.1 int ChemisorptionReaction::ligand_index [protected]

Index of the ligand for all reactions.

5.7.4.2 std::vector<double> ChemisorptionReaction::Delta [protected]

Vector of weights (i.e., deltas) used in the site balance.

5.7.4.3 std::vector<Reaction> ChemisorptionReaction::ads_rxn [private]

List of reactions involved with adsorption.

The documentation for this class was generated from the following file:

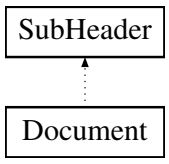
- shark.h

5.8 Document Class Reference

Object for the various documents in the yaml file.

```
#include <yaml_wrapper.h>
```

Inheritance diagram for Document:



Public Member Functions

- Document ()

  Default constructor.
- ~Document ()

  Default destructor.
- Document (const Document &doc)

  Copy constructor.
- Document (std::string name)

  Constructor by name.
- Document (const KeyValueMap &map)

  Constructor by existing map.
- Document (std::string name, const KeyValueMap &map)

  Constructor by name and map.
- Document (std::string key, const Header &head)

  Constructor by single header.
- Document & operator= (const Document &doc)

  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)

  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const

  Return the ValueType at the given key.
- Header & operator() (const std::string key)

  Return the Header reference at the given key.
- Header operator() (const std::string key) const

  Return the Header at the given key.
- std::map< std::string, Header > & getHeadMap ()

  Return the reference to the Header Map.
- KeyValueMap & getDataMap ()

  Return the reference to the KeyValueMap.
- Header & getHeader (std::string key)

  Return reference to the Header in map at the key.
- std::map< std::string, Header >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, Header >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, Header >::const_iterator begin () const

  Returns a const iterator pointing to the begining of the list.
- std::map< std::string, Header >::iterator begin ()

  Returns an iterator pointing to the begining of the list.
- void clear ()

  Clear out info in the Document.
- void resetKeys ()

  Set all keys in the map to match names of the headers.
- void changeKey (std::string oldKey, std::string newKey)

  Change a given oldKey in the header map to the newKey given.
- void revalidateAllKeys ()

  Resets and validates keys in header and subheader maps.
- void addPair (std::string key, std::string val)

  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int t)

Adds a pair object and asserts a type.

- void setName (std::string name)

  Set the name of the Document.
- void setAlias (std::string alias)

  Set the alias of the Document.
- void setNameAliasPair (std::string n, std::string a, int s)

  Set the name, alias, and state of the document.
- void setState (int state)

  Set the state of the Document.
- void DisplayContents ()

  Display the contents of the Document.
- void addHeadKey (std::string key)

  Add a key to the Header without a header object.
- void copyAnchor2Alias (std::string alias, Header &ref)

  Find the anchor in the map, and copy to the Header reference given.
- int size ()

  Return the size of the header map.
- std::string getName ()

  Return the name of the document.
- std::string getAlias ()

  Return the alias of the document.
- int getState ()

  Return the state of the document.
- bool isAlias ()

  Returns true if the document is an alias.
- bool isAnchor ()

  Returns true if the document is an anchor.
- Header & getAnchoredHeader (std::string alias)

  Returns reference to the anchored header, if any.
- Header & getHeadFromSubAlias (std::string alias)

  Returns reference to the Header that contains a Sub with the given alias.

Private Attributes

- std::map< std::string, Header > Head_Map

  Map of headers contained within the document.

Additional Inherited Members

5.8.1 Detailed Description

Object for the various documents in the yaml file.

C++ Object for the documents in a yaml input file as denoted by a Key: followed by — (three dashes) and ending with a ... (three dots). A single yaml file can have multiple document structures and each document structure can have multiple headers (which have sub-headers and key-values) and key-value-pairs. This is the larges single object in the yaml file itself.

Just like Header, this object also inherits from SubHeader and therefore has access to its protected members. You can use access to those members to establish the KeyValuePairs in the Document, name the Document, and give the Document an alias or anchor value.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Document::Document ( )

Default constructor.

5.8.2.2 Document::~Document ( )

Default destructor.

5.8.2.3 Document::Document (const Document & *doc*)

Copy constructor.

5.8.2.4 Document::Document (std::string *name*)

Constructor by name.

5.8.2.5 Document::Document (const KeyValueMap & *map*)

Constructor by existing map.

5.8.2.6 Document::Document (std::string *name,* const KeyValueMap & *map*)

Constructor by name and map.

5.8.2.7 Document::Document (std::string *key,* const Header & *head*)

Constructor by single header.

5.8.3 Member Function Documentation

5.8.3.1 Document& Document::operator= (const Document & *doc*)

Equals overload.

5.8.3.2 ValueTypePair& Document::operator[ ] (const std::string *key*)

Return the ValueType reference at the given key.

5.8.3.3 ValueTypePair Document::operator[ ] (const std::string *key*) const

Return the ValueType at the given key.

5.8.3.4 Header& Document::operator() (const std::string *key*)

Return the Header reference at the given key.

5.8.3.5 Header Document::operator() (const std::string *key*) const

Return the Header at the given key.

5.8.3.6 std::map<std::string, Header>& Document::getHeadMap ( )

Return the reference to the Header Map.

5.8.3.7 KeyValueMap& Document::getDataMap ( )

Return the reference to the KeyValueMap.

5.8.3.8 Header& Document::getHeader (std::string *key*)

Return reference to the Header in map at the key.

5.8.3.9 std::map<std::string, Header>::const_iterator Document::end ( ) const

Returns a const iterator pointing to the end of the list.

5.8.3.10 std::map<std::string, Header>::iterator Document::end ( )

Returns an iterator pointing to the end of the list.

5.8.3.11 std::map<std::string, Header>::const_iterator Document::begin ( ) const

Returns a const iterator pointing to the begining of the list.

5.8.3.12 std::map<std::string, Header>::iterator Document::begin ( )

Returns an iterator pointing to the begining of the list.

5.8.3.13 void Document::clear ( )

Clear out info in the Document.

5.8.3.14 void Document::resetKeys ( )

Set all keys in the map to match names of the headers.

5.8.3.15 void Document::changeKey (std::string *oldKey*, std::string *newKey*)

Change a given oldKey in the header map to the newKey given.

5.8.3.16 void Document::revalidateAllKeys ( )

Resets and validates keys in header and subheader maps.

5.8.3.17 void Document::addPair (std::string *key,* std::string *val*)

Adds a pair object to the map (with only strings)

5.8.3.18 void Document::addPair (std::string *key,* std::string *val,* int *t*)

Adds a pair object and asserts a type.

5.8.3.19 void Document::setName (std::string *name*)

Set the name of the Document.

5.8.3.20 void Document::setAlias (std::string *alias*)

Set the alias of the Document.

5.8.3.21 void Document::setNameAliasPair (std::string *n,* std::string *a,* int *s*)

Set the name, alias, and state of the document.

5.8.3.22 void Document::setState (int *state*)

Set the state of the Document.

5.8.3.23 void Document::DisplayContents ( )

Display the contents of the Document.

5.8.3.24 void Document::addHeadKey (std::string *key*)

Add a key to the Header without a header object.

5.8.3.25 void Document::copyAnchor2Alias (std::string *alias,* Header & *ref*)

Find the anchor in the map, and copy to the Header reference given.

5.8.3.26 int Document::size ( )

Return the size of the header map.

5.8.3.27 std::string Document::getName ( )

Return the name of the document.

5.8.3.28 std::string Document::getAlias ( )

Return the alias of the document.

5.8.3.29 int Document::getState ( )

Return the state of the document.

5.8.3.30 bool Document::isAlias ( )

Returns true if the document is an alias.

5.8.3.31 bool Document::isAnchor ( )

Returns true if the document is an anchor.

5.8.3.32 Header& Document::getAnchoredHeader (std::string *alias*)

Returns reference to the anchored header, if any.

5.8.3.33 Header& Document::getHeadFromSubAlias (std::string *alias*)

Returns reference to the Header that contains a Sub with the given alias.

5.8.4 Member Data Documentation

5.8.4.1 std::map<std::string, Header> Document::Head_Map [private]

Map of headers contained within the document.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.9 DOGFISH_DATA Struct Reference

Primary data structure for running the DOGFISH application.

```
#include <dogfish.h>
```

Public Attributes

- unsigned long int total_steps = 0

  Total number of solver steps taken.
- double time_old = 0.0

  Old value of time (hrs)
- double time = 0.0

  Current value of time (hrs)
- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- bool DirichletBC = false

  False = uses film mass transfer for BC, True = Dirichlet BC.
- bool NonLinear = false

  False = Solve directly, True = Solve iteratively.
- double t_counter = 0.0

  Counter for the time output.
- double t_print

  Print output at every t_print time (hrs)
- int NumComp

  Number of species to track.
- double end_time

  Units: hours.
- double total_sorption_old

  Per mass or volume of single fiber.
- double total_sorption

  Per mass or volume of single fiber.
- double fiber_length

  Units: um.
- double fiber_diameter

  Units: um.
- double fiber_specific_area

  Units: m^2/kg.
- FILE * OutputFile

  Output file pointer to the output file for postprocesses and results.
- double(* eval_R)(int i, int l, const void *data)

  Function pointer to evaluate retardation coefficient.
- double(* eval_DI)(int i, int l, const void *data)

  Function pointer to evaluate intraparticle diffusivity.
- double(* eval_kf)(int i, const void *data)

  Function pointer to evaluate film mass transfer coefficient.
- double(* eval_qs)(int i, const void *data)

  Function pointer to evaluate fiber surface concentration.
- const void * user_data

  Data structure for users info to calculate all parameters.
- std::vector< FINCH_DATA > finch_dat

  Data structure for FINCH_DATA objects.
- std::vector< DOGFISH_PARAM > param_dat

  Data structure for DOGFISH_PARAM objects.

5.9.1 Detailed Description

Primary data structure for running the DOGFISH application.

C-style object to hold information for the adsorption simulations. Contains function pointers and other data structures. This information is passed around to other functions used to simulate the fiber diffusion physics.

5.9.2 Member Data Documentation

5.9.2.1 unsigned long int DOGFISH_DATA::total_steps = 0

Total number of solver steps taken.

5.9.2.2 double DOGFISH_DATA::time_old = 0.0

Old value of time (hrs)

5.9.2.3 double DOGFISH_DATA::time = 0.0

Current value of time (hrs)

5.9.2.4 bool DOGFISH_DATA::Print2File = true

True = results to .txt; False = no printing.

5.9.2.5 bool DOGFISH_DATA::Print2Console = true

True = results to console; False = no printing.

5.9.2.6 bool DOGFISH_DATA::DirichletBC = false

False = uses film mass transfer for BC, True = Dirichlet BC.

5.9.2.7 bool DOGFISH_DATA::NonLinear = false

False = Solve directly, True = Solve iteratively.

5.9.2.8 double DOGFISH_DATA::t_counter = 0.0

Counter for the time output.

5.9.2.9 double DOGFISH_DATA::t_print

Print output at every t_print time (hrs)

5.9.2.10 int DOGFISH_DATA::NumComp

Number of species to track.

5.9.2.11 double DOGFISH_DATA::end_time

Units: hours.

5.9.2.12 double DOGFISH_DATA::total_sorption_old

Per mass or volume of single fiber.

5.9.2.13 double DOGFISH_DATA::total_sorption

Per mass or volume of single fiber.

5.9.2.14 double DOGFISH_DATA::fiber_length

Units: um.

5.9.2.15 double DOGFISH_DATA::fiber_diameter

Units: um.

5.9.2.16 double DOGFISH_DATA::fiber_specific_area

Units: m^2/kg.

5.9.2.17 FILE* DOGFISH_DATA::OutputFile

Output file pointer to the output file for postprocesses and results.

5.9.2.18 double(* DOGFISH_DATA::eval_R) (int i, int l, const void *data)

Function pointer to evaluate retardation coefficient.

5.9.2.19 double(* DOGFISH_DATA::eval_Dl) (int i, int l, const void *data)

Function pointer to evaluate intraparticle diffusivity.

5.9.2.20 double(* DOGFISH_DATA::eval_kf) (int i, const void *data)

Function pointer to evaluate film mass transfer coefficient.

5.9.2.21 double(* DOGFISH_DATA::eval_qs) (int i, const void *data)

Function pointer to evaluate fiber surface concentration.

5.9.2.22 const void* DOGFISH_DATA::user_data

Data structure for users info to calculate all parameters.

5.9.2.23 std::vector<FINCH_DATA> DOGFISH_DATA::finch_dat

Data structure for FINCH_DATA objects.

5.9.2.24 std::vector<DOGFISH_PARAM> DOGFISH_DATA::param_dat

Data structure for DOGFISH_PARAM objects.

The documentation for this struct was generated from the following file:

- dogfish.h

5.10 DOGFISH_PARAM Struct Reference

Data structure for species-specific parameters.

```
#include <dogfish.h>
```

Public Attributes

- double intraparticle_diffusion
  Units: um^2/hr.
- double film_transfer_coeff
  Units: um/hr.
- double surface_concentration
  Units: mol/kg.
- double initial_sorption
  Units: mol/kg.
- double sorbed_molefraction
  Molefraction of sorbed species.
- Molecule species
  Adsorbed species Molecule Object.

5.10.1 Detailed Description

Data structure for species-specific parameters.

C-style object to hold information on all adsorbing species. Parameters are given descriptive names to indicate what each is for.

5.10.2 Member Data Documentation

5.10.2.1 double DOGFISH_PARAM::intraparticle_diffusion

Units: um^2/hr.

5.10.2.2 double DOGFISH_PARAM::film_transfer_coeff

Units: um/hr.

5.10.2.3 double DOGFISH_PARAM::surface_concentration

Units: mol/kg.

5.10.2.4 double DOGFISH_PARAM::initial_sorption

Units: mol/kg.

5.10.2.5 double DOGFISH_PARAM::sorbed_molefraction

Molefraction of sorbed species.

5.10.2.6 Molecule DOGFISH_PARAM::species

Adsorbed species Molecule Object.

The documentation for this struct was generated from the following file:

- dogfish.h

5.11 Dove Class Reference

Dynamic ODE-solver with Various Established methods (DOVE) object.

`#include <dove.h>`

Public Member Functions

- Dove ()
  Default constructor.
- ~Dove ()
  Default destructor.
- void set_numfunc (int i)
  Set the number of functions to solve and reserve necessary space.
- void set_timestep (double d)
  Set the value of the time step.
- void set_timestepmin (double dmin)
  Set the value of the minimum time step.
- void set_timestepmax (double dmax)
  Set the value of the maximum time step.
- void set_endtime (double e)
  Set the value of the end time.
- void set_integrationtype (integrate_subtype type)
  Set the type of integration scheme to use.
- void set_timestepper (timestep_type type)

Set the time stepper scheme type.

- void set_preconditioner (precond_type type)

  Set the type of preconditioner to use.
- void set_outputfile (FILE *file)

  Set the output file for simulation results.
- void set_userdata (const void *data)

  Set the user defined data structure.
- void set_initialcondition (int i, double ic)

  Set the initial condition of variable i to value ic.
- void set_output (bool choice)

  Set the value of DoveOutput (True if you want console messages)
- void set_fileoutput (bool choice)

  Set the value of DoveFileOuput (True if you want results printed to file)
- void set_tolerance (double tol)

  Set the value of residual/error tolerance desired.
- void set_defaultCoeffs ()

  Set all coeff functions to the default.
- void set_defaultJacobis ()

  Set all Jacobians to the default (only does the diagonals!)
- void set_NonlinearAbsTol (double tol)

  Set the value of nonlinear absolute tolerance.
- void set_NonlinearRelTol (double tol)

  Set the value of nonlinear relative tolerance.
- void set_LinearAbsTol (double tol)

  Set the value of linear absolute tolerance.
- void set_LinearRelTol (double tol)

  Set the value of linear relative tolerance.
- void set_NonlinearOutput (bool choice)

  Sets the non-linear output information according to user choice.
- void set_LinearOutput (bool choice)

  Sets the linear output information according to user choice.
- void set_Preconditioning (bool choice)

  Sets the boolean to determine whether or not to include preconditioning.
- void set_LinearMethod (krylov_method choice)

  Sets the linear solver method to user choice.
- void set_LineSearchMethod (linesearch_type choice)

  Sets the line search method to the user choice.
- void set_MaxNonLinearIterations (int it)

  Set the maximum number of non-linear iterations.
- void set_MaxLinearIterations (int it)

  Set the maximum number of linear iterations (or number of restarts)
- void set_RestartLimit (int it)

  Sets the number of iterations before restarting.
- void set_RecursionLevel (int level)

  Sets the maximum level of recursion for the KMS method.
- void set_LinearStatus (bool choice)

  Sets the boolean to determine whether or not to treat as linear (true = Linear)
- void registerFunction (int i, double(*func)(int i, const Matrix< double > &u, double t, const void *data))

  Register the ith user function.
- void registerCoeff (int i, double(*coeff)(int i, const Matrix< double > &u, double t, const void *data))

  Register the ith time coeff function.

- void registerJacobi (int i, int j, double(∗jac)(int i, int j, const Matrix< double > &u, double t, const void ∗data))

  Register the i-jth element of jacobian.
- void print_header ()

  Function to print out a header to output file.
- void print_newresult ()

  Function to print out the new result of n+1 time level.
- void print_result ()

  Function to print out the old result of n time level.
- Matrix< double > & getCurrentU ()

  Return reference to the n level solution.
- Matrix< double > & getOldU ()

  Return reference to the n-1 level solution.
- Matrix< double > & getNewU ()

  Return reference to the n+1 level solution.
- const void ∗ getUserData ()

  Return pointer to user data.
- int getNumFunc ()

  Return the number of functions.
- double getTimeStep ()

  Return the current time step.
- double getTimeStepOld ()

  Return the old time step.
- double getEndTime ()

  Return value of end time.
- double getCurrentTime ()

  Return the value of current time.
- double getOldTime ()

  Return the value of the previous time.
- double getOlderTime ()

  Return the value of the older previous time.
- double getMinTimeStep ()

  Return the value of the minimum time step.
- double getMaxTimeStep ()

  Return the value of the maximum time step.
- bool hasConverged ()

  Returns state of convergence.
- double getNonlinearResidual ()

  Returns the current value of the non-linear residual.
- double getNonlinearRelativeRes ()

  Returns the current value of the non-linear relative residual.
- std::map< int, double(∗)(int i, int j, const Matrix< double > &u, double t, const void ∗data)> & getJacobiMap (int i)

  Function to return a reference to the Jacobian Matrix map at the ith row of the matrix.
- double ComputeTimeStep ()

  Returns a computed value for the next time step.
- double Eval_Func (int i, const Matrix< double > &u, double t)

  Evaluate user function i at given u matrix and time t.
- double Eval_Coeff (int i, const Matrix< double > &u, double t)

  Evaluate user time coefficient function i at given u matrix and time t.
- double Eval_Jacobi (int i, int j, const Matrix< double > &u, double t)

  Evaluate user jacobian function for (i,j) at given u matrix and time t.

- int solve_timestep ()

  Function to solve a single time step.
- void validate_precond ()

  Function to validate and set preconditioning pointer.
- void validate_linearsteps ()

  Function to check and validate the number of linear iterations.
- void update_states ()

  Function to update the stateful information.
- void update_timestep ()

  Function to update the timestep for the simulation.
- void reset_all ()

  Reset all the states.
- int solve_all ()

  Function to solve the system of equations and print results to file (returns 0 on success)
- int solve_FE ()

  Solver function for explicit-FE method.
- int solve_RK4 ()

  Solver function for explicit-RK4 method.
- int solve_RKF ()

  Solver function for explicit-RKF method.

Protected Attributes

- Matrix< double > un

  Matrix for nth level solution vector.
- Matrix< double > unp1

  Matrix for n+1 level solution vector.
- Matrix< double > unm1

  Matrix for n-1 level solution vector.
- double dt

  Time step between n and n+1 time levels.
- double dt_old

  Time step between n and n-1 time levels.
- double time_end

  Time on which to end the ODE simulations.
- double time

  Value of current time.
- double time_old

  Value of previous time.
- double time_older

  Value of older previous time.
- double dtmin

  Minimum allowable time step.
- double dtmax

  Maximum allowable time step.
- double tolerance

  Residual tolerance desired.
- integrate_type int_type

  Type of time integration to use.
- integrate_subtype int_sub

Subtype of time integration scheme to use.

- timestep_type timestepper

  Type of time stepper to be used.

- precond_type preconditioner

  Type of preconditioner to use.

- FILE * Output

  File to where simulation results will be place.

- int num_func

  Number of functions in the system of ODEs.

- bool Converged

  Boolean to hold information on whether or not last step converged.

- bool DoveOutput

  Boolean to determine whether or not to print Dove messages to console.

- bool DoveFileOutput

  Boolean to determine whether or not to print Dove ouput to the file.

- bool Preconditioner

  Boolean to determine whether or not to use a preconditioner.

- bool Linear

  Boolean to determine whether or not to treat problem as linear.

- int linmax

  Users requested maximum number of linear steps.

- Matrix< double(*)(int i, const Matrix< double > &u, double t, const void *data)> user_func

  Matrix object for user defined rate functions.

- Matrix< double(*)(int i, const Matrix< double > &u, double t, const void *data)> user_coeff

  Matrix object for user defined time coefficients (optional)

- std::vector< std::map< int, double(*)(int i, int j, const Matrix< double > &u, double t, const void *data)> > user_jacobi

  A vector of Maps for user defined Jacobian elements (optional)

- const void * user_data

  Pointer for user defined data structure.

- PJFNK_DATA newton_dat

- int(* residual)(const Matrix< double > &x, Matrix< double > &F, const void *res_data)

  Function pointer for the residual function of DOVE.

- int(* precon)(const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

  Function pointer for the preconditioning operation of DOVE.

5.11.1 Detailed Description

Dynamic ODE-solver with Various Established methods (DOVE) object.

This class structure creates a C++ object that can be used to solve coupled systems of Ordinary Differential Equations. A user will interface with this object by creating functions that evaluate the right-hand side of an ODE based on the given variable set. Those functions will collectively create the system to solve numerically using either explicit or implicit methods. The choice of methods can be set by the user, or the object will default to the Backwards-Euler implicit method for stability.

User functions for the right-hand side are written as...

du_i/dt = user_function_i(const Matrix<double> &u, const void *data_struct)

In some cases, there is a need to include a time coefficient on the left-hand side of the rate expression. For those cases, the user may also provide a time coefficient function...

user_time_coeff_i(const Matrix<double> &u, const void *data_struct) * du_i/dt = user_function_i(...)

For most implicit problems, the ODE system must be solved iteratively using a Newton-style method. In these cases, the user may also provide functions for Jacobian matrix elements...

user_jacobi_element_i_j(const Matrix<double> &u, const void *data_struct)

All of these above functions are to be put into Matrices inside of the Dove class object so that Dove will call those functions when it needs to be called. Data structures for all function calls are optional and are to be defined by the user to contain whatever parameter information is needed for their particular problem.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 Dove::Dove ( )

Default constructor.

5.11.2.2 Dove::~Dove ( )

Default destructor.

5.11.3 Member Function Documentation

5.11.3.1 void Dove::set_numfunc (int *i*)

Set the number of functions to solve and reserve necessary space.

5.11.3.2 void Dove::set_timestep (double *d*)

Set the value of the time step.

5.11.3.3 void Dove::set_timestepmin (double *dmin*)

Set the value of the minimum time step.

5.11.3.4 void Dove::set_timestepmax (double *dmax*)

Set the value of the maximum time step.

5.11.3.5 void Dove::set_endtime (double *e*)

Set the value of the end time.

5.11.3.6 void Dove::set_integrationtype (integrate_subtype *type*)

Set the type of integration scheme to use.

5.11.3.7 void Dove::set_timestepper (timestep_type *type*)

Set the time stepper scheme type.

5.11.3.8 void Dove::set_preconditioner (precond_type *type*)

Set the type of preconditioner to use.

5.11.3.9 void Dove::set_outputfile (FILE * *file*)

Set the output file for simulation results.

5.11.3.10 void Dove::set_userdata (const void * *data*)

Set the user defined data structure.

5.11.3.11 void Dove::set_initialcondition (int *i,* double *ic*)

Set the initial condition of variable i to value ic.

5.11.3.12 void Dove::set_output (bool *choice*)

Set the value of DoveOutput (True if you want console messages)

5.11.3.13 void Dove::set_fileoutput (bool *choice*)

Set the value of DoveFileOuput (True if you want results printed to file)

5.11.3.14 void Dove::set_tolerance (double *tol*)

Set the value of residual/error tolerance desired.

5.11.3.15 void Dove::set_defaultCoeffs ( )

Set all coeff functions to the default.

5.11.3.16 void Dove::set_defaultJacobis ( )

Set all Jacobians to the default (only does the diagonals!)

5.11.3.17 void Dove::set_NonlinearAbsTol (double *tol*)

Set the value of nonlinear absolute tolerance.

5.11.3.18 void Dove::set_NonlinearRelTol (double *tol*)

Set the value of nonlinear relative tolerance.

5.11.3.19 void Dove::set_LinearAbsTol (double *tol*)

Set the value of linear absolute tolerance.

5.11.3.20 void Dove::set_LinearRelTol (double *tol*)

Set the value of linear relative tolerance.

5.11.3.21 void Dove::set_NonlinearOutput (bool *choice*)

Sets the non-linear output information according to user choice.

5.11.3.22 void Dove::set_LinearOutput (bool *choice*)

Sets the linear output information according to user choice.

5.11.3.23 void Dove::set_Preconditioning (bool *choice*)

Sets the boolean to determine whether or not to include preconditioning.

5.11.3.24 void Dove::set_LinearMethod (krylov_method *choice*)

Sets the linear solver method to user choice.

5.11.3.25 void Dove::set_LineSearchMethod (linesearch_type *choice*)

Sets the line search method to the user choice.

5.11.3.26 void Dove::set_MaxNonLinearIterations (int *it*)

Set the maximum number of non-linear iterations.

5.11.3.27 void Dove::set_MaxLinearIterations (int *it*)

Set the maximum number of linear iterations (or number of restarts)

5.11.3.28 void Dove::set_RestartLimit (int *it*)

Sets the number of iterations before restarting.

5.11.3.29 void Dove::set_RecursionLevel (int *level*)

Sets the maximum level of recursion for the KMS method.

5.11.3.30 void Dove::set_LinearStatus (bool *choice*)

Sets the boolean to determine whether or not to treat as linear (true = Linear)

5.11.3.31 void Dove::registerFunction (int *i,* double(∗)(int i, const Matrix< double > &u, double t, const void ∗data) *func*)

Register the ith user function.

This function will register the ith user function into the object. That function must accept as arguments the function identifier i, a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifier i can be used to conveniently define coupling between nieghboring elements/variables in the system. In other words, the int i denotes not only the function being registered, but also the primary coupled variable for the function.

i.e., du_i/dt = Func(u_i all other u)

This will allow for this framework to also handle PDEs, whose coupling between ith and jth variables is usually done via nieghboring variables (i.e., u_i in a 1-D PDE couples with u_i-1 and u_i+1). A similar relational scheme is workable with multiple dimensions. Additional information about the coupling between the ith variable and other variables can be passed to the function via the void data pointer.

Note

You are allowed to point to the same user function for all i, but you must make sure that the resulting system is non-singular (i.e., use argument i passed to the function to denote interally which function you are at).

5.11.3.32 void Dove::registerCoeff (int *i,* double(∗)(int i, const Matrix< double > &u, double t, const void ∗data) *coeff*)

Register the ith time coeff function.

This function will register the ith coeff function into the object. That function must accept as arguments the coefficient identifier i, a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifier i can be used to conveniently define identify where the coefficient may be applied spatially. In other words, if solving a PDE, the time coefficient may be a function of location in space, which can be potentially identified by int i.

For example, in 1-D space, the distance x can be computed as x = dx∗i for a regular grid.

5.11.3.33 void Dove::registerJacobi (int *i,* int *j,* double(∗)(int i, int j, const Matrix< double > &u, double t, const void ∗data) *jac*)

Register the i-jth element of jacobian.

This function will register the (i,j) jacobian function into the object. That function must accept as arguments the jacobi identifiers (i and j), a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifiers i and j can be used to determine which Jacobian function this should be, thus allowing a user to potentially reference the same function for all Jacobi elements, but return different results based on matrix location.

Jacobian elements are as follows: J_ij = d(func_i)/d(u_j) derivative of ith function with respect to jth variable.

Note

The jacobian information is used only in preconditioning actions taken by DOVE. The type of preconditioning can be choosen by the user. There are standard types of preconditioning available.

5.11.3.34 void Dove::print_header ( )

Function to print out a header to output file.

5.11.3.35 void Dove::print_newresult ( )

Function to print out the new result of n+1 time level.

5.11.3.36 void Dove::print_result ( )

Function to print out the old result of n time level.

5.11.3.37 Matrix<double>& Dove::getCurrentU ( )

Return reference to the n level solution.

5.11.3.38 Matrix<double>& Dove::getOldU ( )

Return reference to the n-1 level solution.

5.11.3.39 Matrix<double>& Dove::getNewU ( )

Return reference to the n+1 level solution.

5.11.3.40 const void* Dove::getUserData ( )

Return pointer to user data.

5.11.3.41 int Dove::getNumFunc ( )

Return the number of functions.

5.11.3.42 double Dove::getTimeStep ( )

Return the current time step.

5.11.3.43 double Dove::getTimeStepOld ( )

Return the old time step.

5.11.3.44 double Dove::getEndTime ( )

Return value of end time.

5.11.3.45 double Dove::getCurrentTime ( )

Return the value of current time.

5.11.3.46 double Dove::getOldTime ( )

Return the value of the previous time.

5.11.3.47 double Dove::getOlderTime ( )

Return the value of the older previous time.

5.11.3.48 double Dove::getMinTimeStep ( )

Return the value of the minimum time step.

5.11.3.49 double Dove::getMaxTimeStep ( )

Return the value of the maximum time step.

5.11.3.50 bool Dove::hasConverged ( )

Returns state of convergence.

5.11.3.51 double Dove::getNonlinearResidual ( )

Returns the current value of the non-linear residual.

5.11.3.52 double Dove::getNonlinearRelativeRes ( )

Returns the current value of the non-linear relative residual.

5.11.3.53 std::map<int, double (*) (int i, int j, const Matrix<double> &u, double t, const void *data)>& Dove::getJacobiMap (int *i*)

Function to return a reference to the Jacobian Matrix map at the ith row of the matrix.

5.11.3.54 double Dove::ComputeTimeStep ( )

Returns a computed value for the next time step.

5.11.3.55 double Dove::Eval_Func (int *i*, const Matrix< double > & *u*, double *t*)

Evaluate user function i at given u matrix and time t.

5.11.3.56 double Dove::Eval_Coeff (int *i*, const Matrix< double > & *u*, double *t*)

Evaluate user time coefficient function i at given u matrix and time t.

5.11.3.57 double Dove::Eval_Jacobi (int *i*, int *j*, const Matrix< double > & *u*, double *t*)

Evaluate user jacobian function for (i,j) at given u matrix and time t.

5.11.3.58 int Dove::solve_timestep ( )

Function to solve a single time step.

5.11.3.59 void Dove::validate_precond ( )

Function to validate and set preconditioning pointer.

5.11.3.60 void Dove::validate_linearsteps ( )

Function to check and validate the number of linear iterations.

5.11.3.61 void Dove::update_states ( )

Function to update the stateful information.

5.11.3.62 void Dove::update_timestep ( )

Function to update the timestep for the simulation.

5.11.3.63 void Dove::reset_all ( )

Reset all the states.

5.11.3.64 int Dove::solve_all ( )

Function to solve the system of equations and print results to file (returns 0 on success)

This function will iteratively go through and solve the system for all time steps until either failure occurs or the final time has been reached. Output will be placed into the user's output file or a default output file. This function will assume that the initial conditions have already been set for each variable by the user.

5.11.3.65 int Dove::solve_FE ( )

Solver function for explicit-FE method.

This function will solve the Dove system of equations using the standard Forward-Euler method. In this function, DOVE will call the user defined rate functions and use that information at the previous time level to solve for the next time level directly.

unp1[i] = (Rn[i]∗un[i] + dt∗func[i](unp1)) / Rnp1[i]

5.11.3.66 int Dove::solve_RK4 ( )

Solver function for explicit-RK4 method.

This function will solve the Dove system of equations using the Runge-Kutta 4th order method. In this function, D↩ OVE will call user defined rate functions as necessary and use that information at the previous time level to provide an estimate to the solution at the next time level.

5.11.3.67 int Dove::solve_RKF ( )

Solver function for explicit-RKF method.

This function will solve the Dove system of equations using the Runge-Kutta-Fehlberg method. In this function, D↩ OVE will call user defined rate runctions as necessary and use that information at the previous time level to provide an estimate to the solution at the next time level.

5.11.4 Member Data Documentation

5.11.4.1 Matrix<double> Dove::un [protected]

Matrix for nth level solution vector.

5.11.4.2 Matrix<double> Dove::unp1 [protected]

Matrix for n+1 level solution vector.

5.11.4.3 Matrix<double> Dove::unm1 [protected]

Matrix for n-1 level solution vector.

5.11.4.4 double Dove::dt [protected]

Time step between n and n+1 time levels.

5.11.4.5 double Dove::dt_old [protected]

Time step between n and n-1 time levels.

5.11.4.6 double Dove::time_end [protected]

Time on which to end the ODE simulations.

5.11.4.7 double Dove::time [protected]

Value of current time.

5.11.4.8 double Dove::time_old [protected]

Value of previous time.

5.11.4.9 double Dove::time_older [protected]

Value of older previous time.

5.11.4.10 double Dove::dtmin [protected]

Minimum allowable time step.

5.11.4.11 double Dove::dtmax [protected]

Maximum allowable time step.

5.11.4.12 double Dove::tolerance [protected]

Residual tolerance desired.

5.11.4.13 integrate_type Dove::int_type [protected]

Type of time integration to use.

5.11.4.14 integrate_subtype Dove::int_sub [protected]

Subtype of time integration scheme to use.

5.11.4.15 timestep_type Dove::timestepper [protected]

Type of time stepper to be used.

5.11.4.16 precond_type Dove::preconditioner [protected]

Type of preconditioner to use.

5.11.4.17 FILE* Dove::Output [protected]

File to where simulation results will be place.

5.11.4.18 int Dove::num_func [protected]

Number of functions in the system of ODEs.

5.11.4.19 bool Dove::Converged [protected]

Boolean to hold information on whether or not last step converged.

5.11.4.20 bool Dove::DoveOutput [protected]

Boolean to determine whether or not to print Dove messages to console.

5.11.4.21 bool Dove::DoveFileOutput [protected]

Boolean to determine whether or not to print Dove ouput to the file.

5.11.4.22 bool Dove::Preconditioner [protected]

Boolean to determine whether or not to use a preconditioner.

5.11.4.23 bool Dove::Linear [protected]

Boolean to determine whether or not to treat problem as linear.

5.11.4.24 int Dove::linmax [protected]

Users requested maximum number of linear steps.

5.11.4.25 Matrix<double (*) (int i, const Matrix<double> &u, double t, const void *data)> Dove::user_func [protected]

Matrix object for user defined rate functions.

5.11.4.26 Matrix<double (*) (int i, const Matrix<double> &u, double t, const void *data)> Dove::user_coeff [protected]

Matrix object for user defined time coefficients (optional)

5.11.4.27 std::vector< std::map<int, double (*) (int i, int j, const Matrix<double> &u, double t, const void *data)> > Dove::user_jacobi [protected]

A vector of Maps for user defined Jacobian elements (optional)

This structure creates a Sparse Matrix of functions whose sparcity pattern is unknown at creation. Each "vector" index denotes a row in the full matrix. In each row, there is a map of the non-zero elements. Doing the mapping in this way allows for the sparcity of the matrix to easily change while also allowing for relatively fast access to the non-zero elements.

Note

An unordered map would allow for faster access of specific elements, but may be slower when iterating through that map. May consider changing to unordered map in the future.

5.11.4.28 const void* Dove::user_data [protected]

Pointer for user defined data structure.

5.11.4.29 PJFNK_DATA Dove::newton_dat [protected]

Data structure for the PJFNK iterative method

5.11.4.30 int(* Dove::residual) (const Matrix< double > &x, Matrix< double > &F, const void *res_data) [protected]

Function pointer for the residual function of DOVE.

5.11.4.31 int(* Dove::precon) (const Matrix< double > &r, Matrix< double > &p, const void *precon_data) [protected]

Function pointer for the preconditioning operation of DOVE.

The documentation for this class was generated from the following file:

- dove.h

5.12 FINCH_DATA Struct Reference

Data structure for the FINCH object.

`#include <finch.h>`

Public Attributes

- int d = 0

  Dimension of the problem: 0 = cartesian, 1 = cylindrical, 2 = spherical.
- double dt = 0.0125

  Time step.
- double dt_old = 0.0125

  Previous time step.
- double T = 1.0

  Total time.
- double dz = 0.1

  Space step.
- double L = 1.0

  Total space.
- double s = 1.0

  Char quantity (spherical = 1, cylindrical = length, cartesian = area)
- double t = 0.0

  Current Time.
- double t_old = 0.0

  Previous Time.
- double uT = 0.0

  Total amount of conserved quantity in domain.
- double uT_old = 0.0

  Old Total amount of conserved quantity.
- double uAvg = 0.0

  Average amount of conserved quantity in domain.
- double uAvg_old = 0.0

  Old Average amount of conserved quantity.
- double uIC = 0.0

  Initial condition of Conserved Quantity (if constant)
- double vIC = 1.0

  Initial condition of Velocity (if constant)
- double DIC = 1.0

  Initial condition of Dispersion (if constant)
- double kIC = 1.0

  Initial condition of Reaction (if constant)
- double RIC = 1.0

  Initial condition of the Time Coefficient (if constant)
- double uo = 1.0

  Boundary Value of Conserved Quantity.
- double vo = 1.0

  Boundary Value of Velocity.
- double Do = 1.0

  Boundary Value of Dispersion.
- double ko = 1.0

  Boundary Value of Reaction.
- double Ro = 1.0

  Boundary Value of Time Coefficient.
- double kfn = 1.0

  Film mass transfer coefficient Old.
- double kfnp1 = 1.0

Film mass transfer coefficient New.

- double lambda_I

  Boundary Coefficient for Implicit Neumann (Calculated at Runtime)
- double lambda_E

  Boundary Coefficient for Explicit Neumann (Calculated at Runtime)
- int LN = 10

  Number of nodes.
- bool CN = true

  True if Crank-Nicholson, false if Implicit, never use explicit.
- bool Update = false

  Flag to check if the system needs updating.
- bool Dirichlet = false

  Flag to indicate use of Dirichlet or Neumann starting boundary.
- bool CheckMass = false

  Flag to indicate whether or not mass is to be checked.
- bool ExplicitFlux = false

  Flag to indicate whether or not to use fully explicit flux limiters.
- bool Iterative = true

  Flag to indicate whether to solve directly, or iteratively.
- bool SteadyState = false

  Flag to determine whether or not to solve the steady-state problem.
- bool NormTrack = true

  Flag to determine whether or not to track the norms during simulation.
- double beta = 0.5

  Scheme type indicator: 0.5=CN & 1.0=Implicit; all else NULL.
- double tol_rel = 1e-6

  Relative Tolerance for Convergence.
- double tol_abs = 1e-6

  Absolute Tolerance for Convergence.
- int max_iter = 20

  Maximum number of iterations allowed.
- int total_iter = 0

  Total number of iterations made.
- int nl_method = FINCH_Picard

  Non-linear solution method - default = FINCH_Picard.
- std::vector< double > CL_I

  Left side, implicit coefficients (Calculated at Runtime)
- std::vector< double > CL_E

  Left side, explicit coefficients (Calculated at Runtime)
- std::vector< double > CC_I

  Centered, implicit coefficients (Calculated at Runtime)
- std::vector< double > CC_E

  Centered, explicit coefficients (Calculated at Runtime)
- std::vector< double > CR_I

  Right side, implicit coefficients (Calculated at Runtime)
- std::vector< double > CR_E

  Right side, explicit coefficients (Calculated at Runtime)
- std::vector< double > fL_I

  Left side, implicit fluxes (Calculated at Runtime)
- std::vector< double > fL_E

  Left side, explicit fluxes (Calculated at Runtime)

- std::vector< double > fC_I

  Centered, implicit fluxes (Calculated at Runtime)
- std::vector< double > fC_E

  Centered, explicit fluxes (Calculated at Runtime)
- std::vector< double > fR_I

  Right side, implicit fluxes (Calculated at Runtime)
- std::vector< double > fR_E

  Right side, explicit fluxes (Calculated at Runtime)
- std::vector< double > OI

  Implicit upper diagonal matrix elements (Calculated at Runtime)
- std::vector< double > OE

  Explicit upper diagonal matrix elements (Calculated at Runtime)
- std::vector< double > NI

  Implicit diagonal matrix elements (Calculated at Runtime)
- std::vector< double > NE

  Explicit diagonal matrix elements (Calculated at Runtime)
- std::vector< double > MI

  Implicit lower diagonal matrix elements (Calculated at Runtime)
- std::vector< double > ME

  Explicit lower diagonal matrix elements (Calculated at Runtime)
- std::vector< double > uz_l_I
- std::vector< double > uz_lm1_I
- std::vector< double > uz_lp1_I

  Implicit local slopes (Calculated at Runtime)
- std::vector< double > uz_l_E
- std::vector< double > uz_lm1_E
- std::vector< double > uz_lp1_E

  Explicit local slopes (Calculated at Runtime)
- Matrix< double > unm1

  Conserved Quantity Older.
- Matrix< double > un

  Conserved Quantity Old.
- Matrix< double > unp1

  Conserved Quantity New.
- Matrix< double > u_star

  Conserved Quantity Projected New.
- Matrix< double > ubest

  Best found solution if solving iteratively.
- Matrix< double > vn

  Velocity Old.
- Matrix< double > vnp1

  Velocity New.
- Matrix< double > Dn

  Dispersion Old.
- Matrix< double > Dnp1

  Dispersion New.
- Matrix< double > kn

  Reaction Old.
- Matrix< double > knp1

  Reaction New.
- Matrix< double > Sn

Forcing Function Old.

- Matrix< double > Snp1

  Forcing Function New.

- Matrix< double > Rn

  Time Coeff Old.

- Matrix< double > Rnp1

  Time Coeff New.

- Matrix< double > Fn

  Flux Limiter Old.

- Matrix< double > Fnp1

  Flux Limiter New.

- Matrix< double > gI

  Implicit Side Boundary Conditions.

- Matrix< double > gE

  Explicit Side Boundary Conditions.

- Matrix< double > res

  Current residual.

- Matrix< double > pres

  Current search direction.

- int(* callroutine)(const void *user_data)

  Function pointer to executioner (DEFAULT = default_execution)

- int(* setic)(const void *user_data)

  Function pointer to initial conditions (DEFAULT = default_ic)

- int(* settime)(const void *user_data)

  Function pointer to set time step (DEFAULT = default_timestep)

- int(* setpreprocess)(const void *user_data)

  Function pointer to preprocesses (DEFAULT = default_preprocess)

- int(* solve)(const void *user_data)

  Function pointer to the solver (DEFAULT = default_solve)

- int(* setparams)(const void *user_data)

  Function pointer to set parameters (DEFAULT = default_params)

- int(* discretize)(const void *user_data)

  Function pointer to discretization (DEFAULT = ospre_discretization)

- int(* setbcs)(const void *user_data)
- int(* evalres)(const Matrix< double > &x, Matrix< double > &res, const void *user_data)

  Function pointer to the residual function (DEFAULT = default_res)

- int(* evalprecon)(const Matrix< double > &b, Matrix< double > &p, const void *user_data)

  Function pointer to the preconditioning function (DEFAULT = default_precon)

- int(* setpostprocess)(const void *user_data)

  Function pointer to the postprocesses (DEFAULT = default_postprocess)

- int(* resettime)(const void *user_data)

  Function pointer to reset time (DEFAULT = default_reset)

- PICARD_DATA picard_dat

  Data structure for PICARD method (no need to use this)

- PJFNK_DATA pjfnk_dat

  Data structure for PJFNK method (more rigours method)

- const void * param_data

  User's data structure used to evaluate the parameter function (Must override if setparams is overriden)

5.12.1 Detailed Description

Data structure for the FINCH object.

C-style object that holds data, functions, and other structures necessary to discretize and solve a FINCH problem. All of this information must be overriden or initialized prior to running a FINCH simulation. Many, many default functions are provided to make it easier to incorporate FINCH into other problems. The main function to override will be the setparams function. This will be a function that the user provides to tell the FINCH simulation how the parameters of the problem vary in time and space and whether or not they are coupled the the variable u. All functions are overridable and several can be skipped entirely, or called directly at different times in the execution of a particular routine. This make FINCH extremely flexible to the user.

Note

All parameters and dimensions do not carry any units with them. The user is required to keep track of all their own units in their particular problem and ensure that units will cancel and be consistent in their own physical model.

5.12.2 Member Data Documentation

5.12.2.1 int FINCH_DATA::d = 0

Dimension of the problem: 0 = cartesian, 1 = cylindrical, 2 = spherical.

5.12.2.2 double FINCH_DATA::dt = 0.0125

Time step.

5.12.2.3 double FINCH_DATA::dt_old = 0.0125

Previous time step.

5.12.2.4 double FINCH_DATA::T = 1.0

Total time.

5.12.2.5 double FINCH_DATA::dz = 0.1

Space step.

5.12.2.6 double FINCH_DATA::L = 1.0

Total space.

5.12.2.7 double FINCH_DATA::s = 1.0

Char quantity (spherical = 1, cylindrical = length, cartesian = area)

5.12.2.8 double FINCH_DATA::t = 0.0

Current Time.

5.12.2.9 double FINCH_DATA::t_old = 0.0

Previous Time.

5.12.2.10 double FINCH_DATA::uT = 0.0

Total amount of conserved quantity in domain.

5.12.2.11 double FINCH_DATA::uT_old = 0.0

Old Total amount of conserved quantity.

5.12.2.12 double FINCH_DATA::uAvg = 0.0

Average amount of conserved quantity in domain.

5.12.2.13 double FINCH_DATA::uAvg_old = 0.0

Old Average amount of conserved quantity.

5.12.2.14 double FINCH_DATA::uIC = 0.0

Initial condition of Conserved Quantity (if constant)

5.12.2.15 double FINCH_DATA::vIC = 1.0

Initial condition of Velocity (if constant)

5.12.2.16 double FINCH_DATA::DIC = 1.0

Initial condition of Dispersion (if constant)

5.12.2.17 double FINCH_DATA::kIC = 1.0

Initial condition of Reaction (if constant)

5.12.2.18 double FINCH_DATA::RIC = 1.0

Initial condition of the Time Coefficient (if constant)

5.12.2.19 double FINCH_DATA::uo = 1.0

Boundary Value of Conserved Quantity.

5.12.2.20 double FINCH_DATA::vo = 1.0

Boundary Value of Velocity.

5.12.2.21 double FINCH_DATA::Do = 1.0

Boundary Value of Dispersion.

5.12.2.22 double FINCH_DATA::ko = 1.0

Boundary Value of Reaction.

5.12.2.23 double FINCH_DATA::Ro = 1.0

Boundary Value of Time Coefficient.

5.12.2.24 double FINCH_DATA::kfn = 1.0

Film mass transfer coefficient Old.

5.12.2.25 double FINCH_DATA::kfnp1 = 1.0

Film mass transfer coefficient New.

5.12.2.26 double FINCH_DATA::lambda_I

Boundary Coefficient for Implicit Neumann (Calculated at Runtime)

5.12.2.27 double FINCH_DATA::lambda_E

Boundary Coefficient for Explicit Neumann (Calculated at Runtime)

5.12.2.28 int FINCH_DATA::LN = 10

Number of nodes.

5.12.2.29 bool FINCH_DATA::CN = true

True if Crank-Nicholson, false if Implicit, never use explicit.

5.12.2.30 bool FINCH_DATA::Update = false

Flag to check if the system needs updating.

5.12.2.31 bool FINCH_DATA::Dirichlet = false

Flag to indicate use of Dirichlet or Neumann starting boundary.

5.12.2.32 bool FINCH_DATA::CheckMass = false

Flag to indicate whether or not mass is to be checked.

5.12.2.33 bool FINCH_DATA::ExplicitFlux = false

Flag to indicate whether or not to use fully explicit flux limiters.

5.12.2.34 bool FINCH_DATA::Iterative = true

Flag to indicate whether to solve directly, or iteratively.

5.12.2.35 bool FINCH_DATA::SteadyState = false

Flag to determine whether or not to solve the steady-state problem.

5.12.2.36 bool FINCH_DATA::NormTrack = true

Flag to determine whether or not to track the norms during simulation.

5.12.2.37 double FINCH_DATA::beta = 0.5

Scheme type indicator: 0.5=CN & 1.0=Implicit; all else NULL.

5.12.2.38 double FINCH_DATA::tol_rel = 1e-6

Relative Tolerance for Convergence.

5.12.2.39 double FINCH_DATA::tol_abs = 1e-6

Absolute Tolerance for Convergence.

5.12.2.40 int FINCH_DATA::max_iter = 20

Maximum number of iterations allowed.

5.12.2.41 int FINCH_DATA::total_iter = 0

Total number of iterations made.

5.12.2.42 int FINCH_DATA::nl_method = FINCH_Picard

Non-linear solution method - default = FINCH_Picard.

5.12.2.43 std::vector<double> FINCH_DATA::CL_I

Left side, implicit coefficients (Calculated at Runtime)

5.12.2.44 std::vector<double> FINCH_DATA::CL_E

Left side, explicit coefficients (Calculated at Runtime)

5.12.2.45 std::vector<double> FINCH_DATA::CC_I

Centered, implicit coefficients (Calculated at Runtime)

5.12.2.46 std::vector<double> FINCH_DATA::CC_E

Centered, explicit coefficients (Calculated at Runtime)

5.12.2.47 std::vector<double> FINCH_DATA::CR_I

Right side, implicit coefficients (Calculated at Runtime)

5.12.2.48 std::vector<double> FINCH_DATA::CR_E

Right side, explicit coefficients (Calculated at Runtime)

5.12.2.49 std::vector<double> FINCH_DATA::fL_I

Left side, implicit fluxes (Calculated at Runtime)

5.12.2.50 std::vector<double> FINCH_DATA::fL_E

Left side, explicit fluxes (Calculated at Runtime)

5.12.2.51 std::vector<double> FINCH_DATA::fC_I

Centered, implicit fluxes (Calculated at Runtime)

5.12.2.52 std::vector<double> FINCH_DATA::fC_E

Centered, explicit fluxes (Calculated at Runtime)

5.12.2.53 std::vector<double> FINCH_DATA::fR_I

Right side, implicit fluxes (Calculated at Runtime)

5.12.2.54 std::vector<double> FINCH_DATA::fR_E

Right side, explicit fluxes (Calculated at Runtime)

5.12.2.55 std::vector<double> FINCH_DATA::OI

Implicit upper diagonal matrix elements (Calculated at Runtime)

5.12.2.56 std::vector<double> FINCH_DATA::OE

Explicit upper diagonal matrix elements (Calculated at Runtime)

5.12.2.57 std::vector<double> FINCH_DATA::NI

Implicit diagonal matrix elements (Calculated at Runtime)

5.12.2.58 std::vector<double> FINCH_DATA::NE

Explicit diagonal matrix elements (Calculated at Runtime)

5.12.2.59 std::vector<double> FINCH_DATA::MI

Implicit lower diagonal matrix elements (Calculated at Runtime)

5.12.2.60 std::vector<double> FINCH_DATA::ME

Explicit lower diagonal matrix elements (Calculated at Runtime)

5.12.2.61 std::vector<double> FINCH_DATA::uz_l_I

5.12.2.62 std::vector<double> FINCH_DATA::uz_lm1_I

5.12.2.63 std::vector<double> FINCH_DATA::uz_lp1_I

Implicit local slopes (Calculated at Runtime)

5.12.2.64 std::vector<double> FINCH_DATA::uz_l_E

5.12.2.65 std::vector<double> FINCH_DATA::uz_lm1_E

5.12.2.66 std::vector<double> FINCH_DATA::uz_lp1_E

Explicit local slopes (Calculated at Runtime)

5.12.2.67 Matrix<double> FINCH_DATA::unm1

Conserved Quantity Older.

5.12.2.68 Matrix<double> FINCH_DATA::un

Conserved Quantity Old.

5.12.2.69 Matrix<double> FINCH_DATA::unp1

Conserved Quantity New.

5.12.2.70 Matrix<double> FINCH_DATA::u_star

Conserved Quantity Projected New.

5.12.2.71 Matrix<double> FINCH_DATA::ubest

Best found solution if solving iteratively.

5.12.2.72 Matrix<double> FINCH_DATA::vn

Velocity Old.

5.12.2.73 Matrix<double> FINCH_DATA::vnp1

Velocity New.

5.12.2.74 Matrix<double> FINCH_DATA::Dn

Dispersion Old.

5.12.2.75 Matrix<double> FINCH_DATA::Dnp1

Dispersion New.

5.12.2.76 Matrix<double> FINCH_DATA::kn

Reaction Old.

5.12.2.77 Matrix<double> FINCH_DATA::knp1

Reaction New.

5.12.2.78 Matrix<double> FINCH_DATA::Sn

Forcing Function Old.

5.12.2.79 Matrix<double> FINCH_DATA::Snp1

Forcing Function New.

5.12.2.80 Matrix<double> FINCH_DATA::Rn

Time Coeff Old.

5.12.2.81 Matrix<double> FINCH_DATA::Rnp1

Time Coeff New.

5.12.2.82 Matrix<double> FINCH_DATA::Fn

Flux Limiter Old.

5.12.2.83 Matrix<double> FINCH_DATA::Fnp1

Flux Limiter New.

5.12.2.84 Matrix<double> FINCH_DATA::gI

Implicit Side Boundary Conditions.

5.12.2.85 Matrix<double> FINCH_DATA::gE

Explicit Side Boundary Conditions.

5.12.2.86 Matrix<double> FINCH_DATA::res

Current residual.

5.12.2.87 Matrix<double> FINCH_DATA::pres

Current search direction.

5.12.2.88 int(* FINCH_DATA::callroutine) (const void *user_data)

Function pointer to executioner (DEFAULT = default_execution)

5.12.2.89 int(* FINCH_DATA::setic) (const void *user_data)

Function pointer to initial conditions (DEFAULT = default_ic)

5.12.2.90 int(* FINCH_DATA::settime) (const void *user_data)

Function pointer to set time step (DEFAULT = default_timestep)

5.12.2.91 int(* FINCH_DATA::setpreprocess) (const void *user_data)

Function pointer to preprocesses (DEFAULT = default_preprocess)

5.12.2.92 int(* FINCH_DATA::solve) (const void *user_data)

Function pointer to the solver (DEFAULT = default_solve)

5.12.2.93 int(* FINCH_DATA::setparams) (const void *user_data)

Function pointer to set parameters (DEFAULT = default_params)

5.12.2.94 int(* FINCH_DATA::discretize) (const void *user_data)

Function pointer to discretization (DEFAULT = ospre_discretization)

5.12.2.95 int(* FINCH_DATA::setbcs) (const void *user_data)

Function pointer to set boundary conditions (DEFAULT = default_bcs)

5.12.2.96 int(* FINCH_DATA::evalres) (const Matrix< double > &x, Matrix< double > &res, const void *user_data)

Function pointer to the residual function (DEFAULT = default_res)

5.12.2.97 int(* FINCH_DATA::evalprecon) (const Matrix< double > &b, Matrix< double > &p, const void *user_data)

Function pointer to the preconditioning function (DEFAULT = default_precon)

5.12.2.98 int(* FINCH_DATA::setpostprocess) (const void *user_data)

Function pointer to the postprocesses (DEFAULT = default_postprocess)

5.12.2.99 int(* FINCH_DATA::resettime) (const void *user_data)

Function pointer to reset time (DEFAULT = default_reset)

5.12.2.100 PICARD_DATA FINCH_DATA::picard_dat

Data structure for PICARD method (no need to use this)

5.12.2.101 PJFNK_DATA FINCH_DATA::pjfnk_dat

Data structure for PJFNK method (more rigours method)

5.12.2.102 const void* FINCH_DATA::param_data

User's data structure used to evaluate the parameter function (Must override if setparams is overriden)

The documentation for this struct was generated from the following file:

- finch.h

5.13 GCR_DATA Struct Reference

Data structure for the implementation of the GCR algorithm for non-symmetric linear systems.

```
#include <lark.h>
```

Public Attributes

- int restart = -1

  Restart parameter for outer iterations - default = 20.
- int maxit = 0

  Maximum allowable outer iterations.
- int iter_outer = 0

  Number of outer iterations taken.
- int iter_inner = 0

  Number of inner iterations taken.
- int total_iter = 0

  Total number of iterations taken.
- bool breakdown = false

  Boolean to determine if a step has failed.
- double alpha

  Inner iteration step size.
- double beta

  Outer iteration step size.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolute tolerance for convergence - default = 1e-6.
- double res

  Absolute residual norm for linear system.
- double relres

  Relative residual norm for linear system.
- double relres_base

  Initial residual norm of the linear system.
- double bestres

  Best found residual norm of the linear system.
- bool Output = true

  True = print messages to the console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual Vector.
- Matrix< double > c_temp

  Temporary c vector to be updated.
- Matrix< double > u_temp

  Temporary u vector to be updated.
- std::vector< Matrix< double > > u

  Vector span for updating x.
- std::vector< Matrix< double > > c

  Vector span for updating r.
- OPTRANS_DATA transpose_dat

  Data structure for Operator Transposition.

5.13.1 Detailed Description

Data structure for the implementation of the GCR algorithm for non-symmetric linear systems.

C-style object used in conjunction with the Generalized Conjugate Residual (GCR) algorithm for solving a non-symmetric linear system of equations. When the linear system in question has a positive-definite-symmetric component to it, then this algorithm is equivalent to GMRESRP. However, it is generally less efficient than GMRESRP and can suffer breakdowns.

5.13.2 Member Data Documentation

5.13.2.1 int GCR_DATA::restart = -1

Restart parameter for outer iterations - default = 20.

5.13.2.2 int GCR_DATA::maxit = 0

Maximum allowable outer iterations.

5.13.2.3 int GCR_DATA::iter_outer = 0

Number of outer iterations taken.

5.13.2.4 int GCR_DATA::iter_inner = 0

Number of inner iterations taken.

5.13.2.5 int GCR_DATA::total_iter = 0

Total number of iterations taken.

5.13.2.6 bool GCR_DATA::breakdown = false

Boolean to determine if a step has failed.

5.13.2.7 double GCR_DATA::alpha

Inner iteration step size.

5.13.2.8 double GCR_DATA::beta

Outer iteration step size.

5.13.2.9 double GCR_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.13.2.10 double GCR_DATA::tol_abs = 1e-6

Absolute tolerance for convergence - default = 1e-6.

5.13.2.11 double GCR_DATA::res

Absolute residual norm for linear system.

5.13.2.12 double GCR_DATA::relres

Relative residual norm for linear system.

5.13.2.13 double GCR_DATA::relres_base

Initial residual norm of the linear system.

5.13.2.14 double GCR_DATA::bestres

Best found residual norm of the linear system.

5.13.2.15 bool GCR_DATA::Output = true

True = print messages to the console.

5.13.2.16 Matrix<double> GCR_DATA::x

Current solution to the linear system.

5.13.2.17 Matrix<double> GCR_DATA::bestx

Best found solution to the linear system.

5.13.2.18 Matrix<double> GCR_DATA::r

Residual Vector.

5.13.2.19 Matrix<double> GCR_DATA::c_temp

Temporary c vector to be updated.

5.13.2.20 Matrix<double> GCR_DATA::u_temp

Temporary u vector to be updated.

5.13.2.21 std::vector<Matrix<double> > GCR_DATA::u

Vector span for updating x.

5.13.2.22 std::vector<Matrix<double> > GCR_DATA::c

Vector span for updating r.

5.13.2.23 OPTRANS_DATA GCR_DATA::transpose_dat

Data structure for Operator Transposition.

The documentation for this struct was generated from the following file:

- lark.h

5.14 GMRESLP_DATA Struct Reference

Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning.

`#include <lark.h>`

Public Attributes

- int restart = -1
  Restart parameter - default = min(vector_size,20)
- int maxit = 0
  Maximum allowable iterations - default = min(vector_size,1000)
- int iter = 0
  Number of iterations needed for convergence.
- int steps = 0
  Total number of gmres iterations and krylov iterations.
- double tol_rel = 1e-6
  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6
  Absolution tolerance for convergence - default = 1e-6.
- double res
  Absolution redisual norm of the linear system.
- double relres
  Relative residual norm of the linear system.
- double relres_base
  Initial residual norm of the linear system.
- double bestres
  Best found residual norm of the linear system.
- bool Output = true
  True = print messages to console.
- Matrix< double > x
  Current solution to the linear system.
- Matrix< double > bestx
  Best found solution to the linear system.
- Matrix< double > r
  Residual vector for the linear system.
- ARNOLDI_DATA arnoldi_dat
  Data structure for the kyrlov subspace.

5.14.1 Detailed Description

Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning.

C-style object used in conjunction with Generalized Minimum RESidual Left-Precondtioned (GMRESLP) and Full Orthogonalization Method (FOM) algorithms to iteratively or directly solve a linear system of equations. When using with GMRESLP, you can only check/observe the linear residuals before a restart or after the Arnoldi space is constructed. This is because this object uses Left-side Preconditioning. A faster routine may be GMRESRP, which is able to construct residuals after each Arnoldi iteration.

5.14.2 Member Data Documentation

5.14.2.1 int GMRESLP_DATA::restart = -1

Restart parameter - default = min(vector_size,20)

5.14.2.2 int GMRESLP_DATA::maxit = 0

Maximum allowable iterations - default = min(vector_size,1000)

5.14.2.3 int GMRESLP_DATA::iter = 0

Number of iterations needed for convergence.

5.14.2.4 int GMRESLP_DATA::steps = 0

Total number of gmres iterations and krylov iterations.

5.14.2.5 double GMRESLP_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.14.2.6 double GMRESLP_DATA::tol_abs = 1e-6

Absolution tolerance for convergence - default = 1e-6.

5.14.2.7 double GMRESLP_DATA::res

Absolution redisual norm of the linear system.

5.14.2.8 double GMRESLP_DATA::relres

Relative residual norm of the linear system.

5.14.2.9 double GMRESLP_DATA::relres_base

Initial residual norm of the linear system.

5.14.2.10 double GMRESLP_DATA::bestres

Best found residual norm of the linear system.

5.14.2.11 bool GMRESLP_DATA::Output = true

True = print messages to console.

5.14.2.12 Matrix<double> GMRESLP_DATA::x

Current solution to the linear system.

5.14.2.13 Matrix<double> GMRESLP_DATA::bestx

Best found solution to the linear system.

5.14.2.14 Matrix<double> GMRESLP_DATA::r

Residual vector for the linear system.

5.14.2.15 ARNOLDI_DATA GMRESLP_DATA::arnoldi_dat

Data structure for the kyrlov subspace.

The documentation for this struct was generated from the following file:

- lark.h

5.15 GMRESR_DATA Struct Reference

Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GMRESR)

```
#include <lark.h>
```

Public Attributes

- int gcr_restart = -1
  Number of GCR restarts (default = 20, max = N)
- int gcr_maxit = 0
  Number of GCR iterations.
- int gmres_restart = -1
  Number of GMRES restarts (max = 20)
- int gmres_maxit = 1
  Number of GMRES iterations (max = 5, default = 1)
- int N
  Dimension of the linear system.
- int total_iter
  Total GMRES and GCR iterations.
- int iter_outer

Total GCR iterations.

- int iter_inner

  Total GMRES iterations.

- bool GCR_Output = true

  True = print GCR messages.

- bool GMRES_Output = false

  True = print GMRES messages.

- double gmres_tol = 0.1

  Tolerance relative to GCR iterations.

- double gcr_rel_tol = 1e-6

  Relative outer residual tolerance.

- double gcr_abs_tol = 1e-6

  Absolute outer residual tolerance.

- Matrix< double > arg

  Argument matrix passed between preconditioner and iterator.

- GCR_DATA gcr_dat

  Data structure for the outer GCR steps.

- GMRESRP_DATA gmres_dat

  Data structure for the inner GMRES steps.

- int(* matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *matvec_data)

  User supplied matrix-vector product function.

- int(* terminal_precon)(const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

  Optional user supplied terminal preconditioner.

- const void * matvec_data

  Data structure for the user's matvec function.

- const void * term_precon

  Data structure for the user's terminal preconditioner.

5.15.1 Detailed Description

Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GMRESR)

C-style object to be used in conjunction with the Generalized Minimum RESidual Recurive (GMRESR) algorithm. Although the name suggests that this method used GMRES recursively, what it is actually doing is nesting GMRE↩ SRP iterations inside the GCR method to form a preconditioner for GCR. The name GMRESR came from literature (Vorst and Vuik, "GMRESR: A family of nested GMRES methods", 1991).

5.15.2 Member Data Documentation

5.15.2.1 int GMRESR_DATA::gcr_restart = -1

Number of GCR restarts (default = 20, max = N)

5.15.2.2 int GMRESR_DATA::gcr_maxit = 0

Number of GCR iterations.

5.15.2.3 int GMRESR_DATA::gmres_restart = -1

Number of GMRES restarts (max = 20)

5.15.2.4 int GMRESR_DATA::gmres_maxit = 1

Number of GMRES iterations (max = 5, default = 1)

5.15.2.5 int GMRESR_DATA::N

Dimension of the linear system.

5.15.2.6 int GMRESR_DATA::total_iter

Total GMRES and GCR iterations.

5.15.2.7 int GMRESR_DATA::iter_outer

Total GCR iterations.

5.15.2.8 int GMRESR_DATA::iter_inner

Total GMRES iterations.

5.15.2.9 bool GMRESR_DATA::GCR_Output = true

True = print GCR messages.

5.15.2.10 bool GMRESR_DATA::GMRES_Output = false

True = print GMRES messages.

5.15.2.11 double GMRESR_DATA::gmres_tol = 0.1

Tolerance relative to GCR iterations.

5.15.2.12 double GMRESR_DATA::gcr_rel_tol = 1e-6

Relative outer residual tolerance.

5.15.2.13 double GMRESR_DATA::gcr_abs_tol = 1e-6

Absolute outer residual tolerance.

5.15.2.14 Matrix<double> GMRESR_DATA::arg

Argument matrix passed between preconditioner and iterator.

5.15.2.15 GCR_DATA GMRESR_DATA::gcr_dat

Data structure for the outer GCR steps.

5.15.2.16 GMRESRP_DATA GMRESR_DATA::gmres_dat

Data structure for the inner GMRES steps.

5.15.2.17 int(* GMRESR_DATA::matvec) (const Matrix< double > &x, Matrix< double > &Ax, const void *matvec_data)

User supplied matrix-vector product function.

5.15.2.18 int(* GMRESR_DATA::terminal_precon) (const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

Optional user supplied terminal preconditioner.

5.15.2.19 const void* GMRESR_DATA::matvec_data

Data structure for the user's matvec function.

5.15.2.20 const void* GMRESR_DATA::term_precon

Data structure for the user's terminal preconditioner.

The documentation for this struct was generated from the following file:

- lark.h

5.16 GMRESRP_DATA Struct Reference

Data structure for the Restarted GMRES algorithm with Right Preconditioning.

```
#include <lark.h>
```

Public Attributes

- int restart = -1
  Restart parameter - default = min(20,vector_size)
- int maxit = 0
  Maximum allowable outer iterations.
- int iter_outer = 0
  Total number of outer iterations.
- int iter_inner = 0
  Total number of inner iterations.
- int iter_total = 0
  Total number of overall iterations.
- double tol_rel = 1e-6
  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6
  Absolute tolerance for convergence - default = 1e-6.
- double res
  Absolute residual norm for linear system.

- double relres

  Relative residual norm for linear system.
- double relres_base

  Initial residual norm of the linear system.
- double bestres

  Best found residual norm of the linear system.
- bool Output = true

  True = print messages to console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual vector for the linear system.
- std::vector< Matrix< double > > Vk

  (N x k) orthonormal vector basis
- std::vector< Matrix< double > > Zk

  (N x k) preconditioned vector set
- std::vector< std::vector< double > > H

  (k+1 x k) upper Hessenberg storage matrix
- std::vector< std::vector< double > > H_bar

  (k+1 x k) Factorized matrix
- std::vector< double > y

  (k x 1) Vector search direction
- std::vector< double > e0

  (k+1 x 1) Normalized vector with residual info
- std::vector< double > e0_bar

  (k+1 x 1) Factorized normal vector
- Matrix< double > w

  (N) x (1) interim result of the matrix_vector multiplication
- Matrix< double > v

  (N) x (1) holding cell for the column entries of Vk and other interims
- Matrix< double > sum

  (N) x (1) running sum of subspace vectors for use in altering w

5.16.1 Detailed Description

Data structure for the Restarted GMRES algorithm with Right Preconditioning.

C-style object used in conjunction with Generalized Minimum RESidual Right Preconditioned (GMRESRP) algorithm to iteratively solve a linear system of equations. Unlike GMRESLP, the GMRESRP method is capable of checking linear residuals at both the inner and outer steps. As a result, this algorithm may terminate earlier than GMRESLP if it has found a suitable solution during one of the inner steps.

5.16.2 Member Data Documentation

5.16.2.1 int GMRESRP_DATA::restart = -1

Restart parameter - default = min(20,vector_size)

5.16.2.2 int GMRESRP_DATA::maxit = 0

Maximum allowable outer iterations.

5.16.2.3 int GMRESRP_DATA::iter_outer = 0

Total number of outer iterations.

5.16.2.4 int GMRESRP_DATA::iter_inner = 0

Total number of inner iterations.

5.16.2.5 int GMRESRP_DATA::iter_total = 0

Total number of overall iterations.

5.16.2.6 double GMRESRP_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.16.2.7 double GMRESRP_DATA::tol_abs = 1e-6

Absolute tolerance for convergence - default = 1e-6.

5.16.2.8 double GMRESRP_DATA::res

Absolute residual norm for linear system.

5.16.2.9 double GMRESRP_DATA::relres

Relative residual norm for linear system.

5.16.2.10 double GMRESRP_DATA::relres_base

Initial residual norm of the linear system.

5.16.2.11 double GMRESRP_DATA::bestres

Best found residual norm of the linear system.

5.16.2.12 bool GMRESRP_DATA::Output = true

True = print messages to console.

5.16.2.13 Matrix<double> GMRESRP_DATA::x

Current solution to the linear system.

5.16.2.14 Matrix<double> GMRESRP_DATA::bestx

Best found solution to the linear system.

5.16.2.15 Matrix<double> GMRESRP_DATA::r

Residual vector for the linear system.

5.16.2.16 std::vector< Matrix<double> > GMRESRP_DATA::Vk

(N x k) orthonormal vector basis

5.16.2.17 std::vector< Matrix<double> > GMRESRP_DATA::Zk

(N x k) preconditioned vector set

5.16.2.18 std::vector< std::vector< double > > GMRESRP_DATA::H

(k+1 x k) upper Hessenberg storage matrix

5.16.2.19 std::vector< std::vector< double > > GMRESRP_DATA::H_bar

(k+1 x k) Factorized matrix

5.16.2.20 std::vector< double > GMRESRP_DATA::y

(k x 1) Vector search direction

5.16.2.21 std::vector< double > GMRESRP_DATA::e0

(k+1 x 1) Normalized vector with residual info

5.16.2.22 std::vector< double > GMRESRP_DATA::e0_bar

(k+1 x 1) Factorized normal vector

5.16.2.23 Matrix<double> GMRESRP_DATA::w

(N) x (1) interim result of the matrix_vector multiplication

5.16.2.24 Matrix<double> GMRESRP_DATA::v

(N) x (1) holding cell for the column entries of Vk and other interims

5.16.2.25 Matrix<double> GMRESRP_DATA::sum

(N) x (1) running sum of subspace vectors for use in altering w

The documentation for this struct was generated from the following file:

- lark.h

5.17 GPAST_DATA Struct Reference

GPAST Data Structure.

`#include <magpie.h>`

Public Attributes

- double x
  Adsorbed mole fraction.
- double y
  Gas phase mole fraction.
- double He
  Henry's Coefficient (mol/kg/kPa)
- double q
  Amount adsorbed for each component (mol/kg)
- std::vector< double > gama_inf
  Infinite dilution activities.
- double qo
  Pure component capacities (mol/kg)
- double PIo
  Pure component spreading pressures (mol/kg)
- std::vector< double > po
  Pure component reference state pressures (kPa)
- double poi
  Reference state pressures solved for using Recover eval GPAST.
- bool present
  If true, then the component is present; if false, then the component is not present.

5.17.1 Detailed Description

GPAST Data Structure.

C-style object holding all parameter information associated with the Generalized Predictive Adsorbed Solution Theory (GPAST) system of equations. Each species in the gas phase will have one of these objects.

5.17.2 Member Data Documentation

5.17.2.1 double GPAST_DATA::x

Adsorbed mole fraction.

5.17.2.2 double GPAST_DATA::y

Gas phase mole fraction.

5.17.2.3 double GPAST_DATA::He

Henry's Coefficient (mol/kg/kPa)

5.17.2.4 double GPAST_DATA::q

Amount adsorbed for each component (mol/kg)

5.17.2.5 std::vector<double> GPAST_DATA::gama_inf

Infinite dilution activities.

5.17.2.6 double GPAST_DATA::qo

Pure component capacities (mol/kg)

5.17.2.7 double GPAST_DATA::Plo

Pure component spreading pressures (mol/kg)

5.17.2.8 std::vector<double> GPAST_DATA::po

Pure component reference state pressures (kPa)

5.17.2.9 double GPAST_DATA::poi

Reference state pressures solved for using Recover eval GPAST.

5.17.2.10 bool GPAST_DATA::present

If true, then the component is present; if false, then the component is not present.

The documentation for this struct was generated from the following file:

- magpie.h

5.18 GSTA_DATA Struct Reference

GSTA Data Structure.

`#include <magpie.h>`

Public Attributes

- double qmax
  Theoretical maximum capacity of adsorbate-adsorbent pair (mol/kg)
- int m
  Number of parameters in the GSTA isotherm.
- std::vector< double > dHo
  Enthalpies for each site (J/mol)
- std::vector< double > dSo
  Entropies for each site (J/(K∗mol))

5.18.1 Detailed Description

GSTA Data Structure.

C-style object holding all parameter information associated with the Generalized Statistical Thermodynamic Adsorption (GSTA) isotherm model. Each species in the gas phase will have one of these objects.

5.18.2 Member Data Documentation

5.18.2.1 double GSTA_DATA::qmax

Theoretical maximum capacity of adsorbate-adsorbent pair (mol/kg)

5.18.2.2 int GSTA_DATA::m

Number of parameters in the GSTA isotherm.

5.18.2.3 std::vector<double> GSTA_DATA::dHo

Enthalpies for each site (J/mol)

5.18.2.4 std::vector<double> GSTA_DATA::dSo

Entropies for each site (J/(K∗mol))

The documentation for this struct was generated from the following file:

- magpie.h

5.19 GSTA_OPT_DATA Struct Reference

Data structure used in the GSTA optimization routines.

`#include <gsta_opt.h>`

Public Attributes

- int total_eval

  Keeps track of the total number of function evaluations.
- int n_par

  Number of parameters being optimized for.
- double qmax

  Maximum theoretical adsorption capacity (M/M) (0 if unknown)
- int iso

  Keeps isotherm that is currently being optimized.
- std::vector< std::vector< double > > Fobj

  Creates a dynamic array to store all Fobj values.
- std::vector< std::vector< double > > q
- std::vector< std::vector< double > > P

  Creates a dynamic array for q and P data pairs.
- std::vector< std::vector< double > > best_par

  Used to store the values of the parameters of best fit.
- std::vector< std::vector< double > > Kno

  Dimensionless parameters determined from best_par.
- std::vector< std::vector< std::vector< double > > > all_pars

  Used to create a ragged array of all parameters.
- std::vector< std::vector< double > > norms

  Used to store the values of all the calculated norms.
- std::vector< double > opt_qmax

  If qmax is unknown, this vector holds it's optimized values.

5.19.1 Detailed Description

Data structure used in the GSTA optimization routines.

C-style structure that keeps track of all infomation during the optimization routine. All solutions and parameters to the GSTA isotherm are held in order to find the best solution with the fewest parameters.

5.19.2 Member Data Documentation

5.19.2.1 int GSTA_OPT_DATA::total_eval

Keeps track of the total number of function evaluations.

5.19.2.2 int GSTA_OPT_DATA::n_par

Number of parameters being optimized for.

5.19.2.3 double GSTA_OPT_DATA::qmax

Maximum theoretical adsorption capacity (M/M) (0 if unknown)

5.19.2.4 int GSTA_OPT_DATA::iso

Keeps isotherm that is currently being optimized.

5.19.2.5 std::vector<std::vector<double> > GSTA_OPT_DATA::Fobj

Creates a dynamic array to store all Fobj values.

5.19.2.6 std::vector<std::vector<double> > GSTA_OPT_DATA::q

5.19.2.7 std::vector<std::vector<double> > GSTA_OPT_DATA::P

Creates a dynamic array for q and P data pairs.

5.19.2.8 std::vector<std::vector<double> > GSTA_OPT_DATA::best_par

Used to store the values of the parameters of best fit.

5.19.2.9 std::vector<std::vector<double> > GSTA_OPT_DATA::Kno

Dimensionless parameters determined from best_par.

5.19.2.10 std::vector<std::vector<std::vector<double> > > GSTA_OPT_DATA::all_pars

Used to create a ragged array of all parameters.

5.19.2.11 std::vector<std::vector<double> > GSTA_OPT_DATA::norms

Used to store the values of all the calculated norms.

5.19.2.12 std::vector<double> GSTA_OPT_DATA::opt_qmax

If qmax is unknown, this vector holds it's optimized values.

The documentation for this struct was generated from the following file:

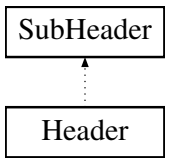
- gsta_opt.h

5.20 Header Class Reference

Object for headers in a yaml document (inherits from SubHeader)

`#include <yaml_wrapper.h>`

Inheritance diagram for Header:



Public Member Functions

- Header ()

  Default Constructor.
- ~Header ()

  Default Destructor.
- Header (const Header &head)

  Copy constructor.
- Header (std::string name)

  Constructor by header name.
- Header (const KeyValueMap &map)

  Constructor by existing map.
- Header (std::string name, const KeyValueMap &map)

  Constructor by name and map.
- Header (std::string key, const SubHeader &sub)

  Constructor by single subheader object.
- Header & operator= (const Header &head)

  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)

  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const

  Return the ValueType at the given key.
- SubHeader & operator() (const std::string key)

  Return the SubHeader reference at the given key.
- SubHeader operator() (const std::string key) const

  Return the SubHeader at the given key.
- std::map< std::string, SubHeader > & getSubMap ()

  Return the reference to the SubHeader Map.
- KeyValueMap & getDataMap ()

  Return the reference to the KeyValueMap.
- SubHeader & getSubHeader (std::string key)

  Return the subheader at the given key.
- std::map< std::string, SubHeader >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, SubHeader >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, SubHeader >::const_iterator begin () const

  Returns a const iterator pointing to the begining of the list.
- std::map< std::string, SubHeader >::iterator begin ()

  Returns an iterator pointing to the begining of the list.
- void clear ()

  Clear out the SubMap, KeyValueMap, and other info.
- void resetKeys ()

  Reset the keys of the SubMap to the names of each SubHeader.
- void changeKey (std::string oldKey, std::string newKey)

  Change one of the keys in the map.
- void addPair (std::string key, std::string val)

  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int t)

  Adds a pair object and asserts a type.
- void setName (std::string name)

Set the name of the Header.

- void setAlias (std::string alias)

  Set the alias of the header, if any.
- void setNameAliasPair (std::string n, std::string a, int s)

  Set the name, alias, and state for the header.
- void setState (int state)

  Set the state of the header, if any.
- void DisplayContents ()

  Display the contents of the header object.
- void addSubKey (std::string key)

  Adds a key to the SubHeader Map.
- void copyAnchor2Alias (std::string alias, SubHeader &ref)

  Find the anchor in the map, and copy to the Header reference given.
- int size ()

  Return the size of the Sub_Map.
- std::string getName ()

  Return the name of the header.
- std::string getAlias ()

  Return the alias of the header.
- int getState ()

  Return the state of the header.
- bool isAlias ()

  Returns true if the header is an alias.
- bool isAnchor ()

  Returns true if the header is an anchor.
- SubHeader & getAnchoredSub (std::string alias)

  Returns reference to the anchored subheader, if any.

Private Attributes

- std::map< std::string, SubHeader > Sub_Map

  Map of the contained subheaders in the main header.

Additional Inherited Members

5.20.1 Detailed Description

Object for headers in a yaml document (inherits from SubHeader)

C++ Object for headers in a yaml document that is built from the SubHeader object already created. The chain of inheritance works in this direction because a Header can have both a map of SubHeaders and a map of KeyValue↩ Pairs. Therefore, the SubHeader object is actually the more generic form of a header.

Since this object inherits from SubHeader, it has access to all it's protected members, including the alias, state, name, and KeyValueMap. Operator overloads and other functions are provided to allow the user to query both the KeyValueMap and SubHeader for specific information. The names of the SubHeaders are also used as it's keys. Make sure all SubHeader keys are unique to this header.

5.20.2 Constructor & Destructor Documentation

5.20.2.1 Header::Header ( )

Default Constructor.

5.20.2.2 Header::~Header ( )

Default Destructor.

5.20.2.3 Header::Header (const Header & *head*)

Copy constructor.

5.20.2.4 Header::Header (std::string *name*)

Constructor by header name.

5.20.2.5 Header::Header (const KeyValueMap & *map*)

Constructor by existing map.

5.20.2.6 Header::Header (std::string *name,* const KeyValueMap & *map*)

Constructor by name and map.

5.20.2.7 Header::Header (std::string *key,* const SubHeader & *sub*)

Constructor by single subheader object.

5.20.3 Member Function Documentation

5.20.3.1 Header& Header::operator= (const Header & *head*)

Equals overload.

5.20.3.2 ValueTypePair& Header::operator[ ] (const std::string *key*)

Return the ValueType reference at the given key.

5.20.3.3 ValueTypePair Header::operator[ ] (const std::string *key*) const

Return the ValueType at the given key.

5.20.3.4 SubHeader& Header::operator() (const std::string *key*)

Return the SubHeader reference at the given key.

5.20.3.5 SubHeader Header::operator() (const std::string *key*) const

Return the SubHeader at the given key.

5.20.3.6 std::map<std::string, SubHeader>& Header::getSubMap ( )

Return the reference to the SubHeader Map.

5.20.3.7 KeyValueMap& Header::getDataMap ( )

Return the reference to the KeyValueMap.

5.20.3.8 SubHeader& Header::getSubHeader (std::string *key*)

Return the subheader at the given key.

5.20.3.9 std::map<std::string, SubHeader>::const_iterator Header::end ( ) const

Returns a const iterator pointing to the end of the list.

5.20.3.10 std::map<std::string, SubHeader>::iterator Header::end ( )

Returns an iterator pointing to the end of the list.

5.20.3.11 std::map<std::string, SubHeader>::const_iterator Header::begin ( ) const

Returns a const iterator pointing to the begining of the list.

5.20.3.12 std::map<std::string, SubHeader>::iterator Header::begin ( )

Returns an iterator pointing to the begining of the list.

5.20.3.13 void Header::clear ( )

Clear out the SubMap, KeyValueMap, and other info.

5.20.3.14 void Header::resetKeys ( )

Reset the keys of the SubMap to the names of each SubHeader.

5.20.3.15 void Header::changeKey (std::string *oldKey*, std::string *newKey*)

Change one of the keys in the map.

5.20.3.16 void Header::addPair (std::string *key*, std::string *val*)

Adds a pair object to the map (with only strings)

5.20.3.17 void Header::addPair (std::string *key,* std::string *val,* int *t*)

Adds a pair object and asserts a type.

5.20.3.18 void Header::setName (std::string *name*)

Set the name of the Header.

5.20.3.19 void Header::setAlias (std::string *alias*)

Set the alias of the header, if any.

5.20.3.20 void Header::setNameAliasPair (std::string *n,* std::string *a,* int *s*)

Set the name, alias, and state for the header.

5.20.3.21 void Header::setState (int *state*)

Set the state of the header, if any.

5.20.3.22 void Header::DisplayContents ( )

Display the contents of the header object.

5.20.3.23 void Header::addSubKey (std::string *key*)

Adds a key to the SubHeader Map.

5.20.3.24 void Header::copyAnchor2Alias (std::string *alias,* SubHeader & *ref*)

Find the anchor in the map, and copy to the Header reference given.

5.20.3.25 int Header::size ( )

Return the size of the Sub_Map.

5.20.3.26 std::string Header::getName ( )

Return the name of the header.

5.20.3.27 std::string Header::getAlias ( )

Return the alias of the header.

5.20.3.28 int Header::getState ( )

Return the state of the header.

5.20.3.29 bool Header::isAlias ( )

Returns true if the header is an alias.

5.20.3.30 bool Header::isAnchor ( )

Returns true if the header is an anchor.

5.20.3.31 SubHeader& Header::getAnchoredSub (std::string *alias*)

Returns reference to the anchored subheader, if any.

5.20.4 Member Data Documentation

5.20.4.1 std::map<std::string, SubHeader> Header::Sub_Map [private]

Map of the contained subheaders in the main header.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.21 KeyValueMap Class Reference

Key-Value-Type Map object creating a map of the KeyValuePair objects.

`#include <yaml_wrapper.h>`

Public Member Functions

- KeyValueMap ()
  Default constructor.
- ~KeyValueMap ()
  Default destructor.
- KeyValueMap (const std::map< std::string, std::string > &map)
  Construct from a map of strings.
- KeyValueMap (std::string key, std::string value)
  Construct one element in the map.
- KeyValueMap (const KeyValueMap &map)
  Copy constructor.
- KeyValueMap & operator= (const KeyValueMap &map)
  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)
  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const
  Return the ValueType at the give key.
- std::map< std::string, ValueTypePair > & getMap ()
  Return a reference to the Key_Value map object.

- std::map< std::string, ValueTypePair >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, ValueTypePair >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, ValueTypePair >::const_iterator begin () const

  Returns a const iterator pointing to the beginning of the list.
- std::map< std::string, ValueTypePair >::iterator begin ()

  Returns an iterator pointing to the beginning of the list.
- void clear ()

  Clears the map.
- void addKey (std::string key)

  Adds a key to the object with a default value.
- void editValue4Key (std::string val, std::string key)

  Edits a given value for a pre-existing key.
- void editValue4Key (std::string val, int type, std::string key)

  Edits a value for a pre-existing key and asserts type.
- void addPair (std::string key, ValueTypePair val)

  Adds a pair object to the map.
- void addPair (std::string key, std::string val)

  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int type)

  Adds a pair object and asserts a type.
- void findType (std::string key)

  Find what data type the value at the key is.
- void assertType (std::string key, int type)

  Assert the given type at the given key.
- void findAllTypes ()

  Find all types for all data in map.
- void DisplayMap ()

  Print out the map to console.
- int size ()

  Returns the size of the map.
- std::string getString (std::string key)

  Retrieve the string at the key.
- bool getBool (std::string key)

  Retrieve the boolean at the key.
- double getDouble (std::string key)

  Retrieve the double at the key.
- int getInt (std::string key)

  Retrieve the int at the key.
- std::string getValue (std::string key)

  Retrieve the value at the key.
- int getType (std::string key)

  Retrieve the type at the key.
- ValueTypePair & getPair (std::string key)

  Retrieve the pair at the key.

Private Attributes

- std::map< std::string, ValueTypePair > Key_Value

  Map of Keys and Values paired with types.

5.21.1 Detailed Description

Key-Value-Type Map object creating a map of the KeyValuePair objects.

C++ Object that creates a map of the KeyValuePair objects. Functions defined here allow the user to iterate through this map, access specific keys in the map, edit values associated with those keys, find the data types for the values in those keys, ect. The keys are used as an access operator for their corresponding value. As such, each key in the map is required to be unique, but the values are allowed to be duplicated.

5.21.2 Constructor & Destructor Documentation

5.21.2.1 KeyValueMap::KeyValueMap ( )

Default constructor.

5.21.2.2 KeyValueMap::~KeyValueMap ( )

Default destructor.

5.21.2.3 KeyValueMap::KeyValueMap (const std::map< std::string, std::string > & *map*)

Construct from a map of strings.

5.21.2.4 KeyValueMap::KeyValueMap (std::string *key,* std::string *value*)

Construct one element in the map.

5.21.2.5 KeyValueMap::KeyValueMap (const **KeyValueMap** & *map*)

Copy constructor.

5.21.3 Member Function Documentation

5.21.3.1 **KeyValueMap**& KeyValueMap::operator= (const **KeyValueMap** & *map*)

Equals overload.

5.21.3.2 **ValueTypePair**& KeyValueMap::operator[ ] (const std::string *key*)

Return the ValueType reference at the given key.

5.21.3.3 **ValueTypePair** KeyValueMap::operator[ ] (const std::string *key*) const

Return the ValueType at the give key.

5.21.3.4 std::map<std::string, **ValueTypePair** >& KeyValueMap::getMap ( )

Return a reference to the Key_Value map object.

5.21.3.5 std::map<std::string, ValueTypePair>::const_iterator KeyValueMap::end ( ) const

Returns a const iterator pointing to the end of the list.

5.21.3.6 std::map<std::string, ValueTypePair>::iterator KeyValueMap::end ( )

Returns an iterator pointing to the end of the list.

5.21.3.7 std::map<std::string, ValueTypePair>::const_iterator KeyValueMap::begin ( ) const

Returns a const iterator pointing to the beginning of the list.

5.21.3.8 std::map<std::string, ValueTypePair>::iterator KeyValueMap::begin ( )

Returns an iterator pointing to the beginning of the list.

5.21.3.9 void KeyValueMap::clear ( )

Clears the map.

5.21.3.10 void KeyValueMap::addKey (std::string *key*)

Adds a key to the object with a default value.

5.21.3.11 void KeyValueMap::editValue4Key (std::string *val,* std::string *key*)

Edits a given value for a pre-existing key.

5.21.3.12 void KeyValueMap::editValue4Key (std::string *val,* int *type,* std::string *key*)

Edits a value for a pre-existing key and asserts type.

5.21.3.13 void KeyValueMap::addPair (std::string *key,* ValueTypePair *val*)

Adds a pair object to the map.

5.21.3.14 void KeyValueMap::addPair (std::string *key,* std::string *val*)

Adds a pair object to the map (with only strings)

5.21.3.15 void KeyValueMap::addPair (std::string *key,* std::string *val,* int *type*)

Adds a pair object and asserts a type.

5.21.3.16 void KeyValueMap::findType (std::string *key*)

Find what data type the value at the key is.

5.21.3.17 void KeyValueMap::assertType (std::string *key,* int *type*)

Assert the given type at the given key.

5.21.3.18 void KeyValueMap::findAllTypes ( )

Find all types for all data in map.

5.21.3.19 void KeyValueMap::DisplayMap ( )

Print out the map to console.

5.21.3.20 int KeyValueMap::size ( )

Returns the size of the map.

5.21.3.21 std::string KeyValueMap::getString (std::string *key*)

Retrieve the string at the key.

5.21.3.22 bool KeyValueMap::getBool (std::string *key*)

Retrieve the boolean at the key.

5.21.3.23 double KeyValueMap::getDouble (std::string *key*)

Retrieve the double at the key.

5.21.3.24 int KeyValueMap::getInt (std::string *key*)

Retrieve the int at the key.

5.21.3.25 std::string KeyValueMap::getValue (std::string *key*)

Retrieve the value at the key.

5.21.3.26 int KeyValueMap::getType (std::string *key*)

Retrieve the type at the key.

5.21.3.27 ValueTypePair& KeyValueMap::getPair (std::string *key*)

Retrieve the pair at the key.

5.21.4 Member Data Documentation

5.21.4.1 std::map<std::string, ValueTypePair > KeyValueMap::Key_Value [private]

Map of Keys and Values paired with types.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.22 KMS_DATA Struct Reference

Data structure for the implemenation of the Krylov Multi-Space (KMS) Method.

`#include <lark.h>`

Public Attributes

- int level = 0

  Current level in the recursion.
- int max_level = 0

  Maximum allowable recursion levels (Default = 0 -> GMRES, Max = 5)
- int restart = -1

  Restart parameter for the outer iterates (Default = 20, Max = N)
- int maxit = 0

  Maximum allowable iterations for the outer steps.
- int inner_iter = 0

  Number of inner steps taken.
- int outer_iter = 0

  Number of outer steps taken.
- int total_iter = 0

  Total number of iterations in all steps.
- double outer_reltol = 1e-6

  Relative residual tolerance for outer steps (Default = 1e-6)
- double outer_abstol = 1e-6

  Absolute residual tolerance for outer steps (Default = 1e-6)
- double inner_reltol = 0.1

  Residual tolerance for inner steps made relative to outer steps (Default = 0.1)
- bool Output_outer = true

  True = Print the outer steps residuals.
- bool Output_inner = false

  True = Print the inner steps residuals.
- GMRESRP_DATA gmres_out

  Data structure for the outer steps.
- std::vector< GMRESRP_DATA > gmres_in

  Data structures for each recursion level.
- int(* matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *matvec_data)

  User supplied matrix-vector product function.
- int(* terminal_precon)(const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

  Optional user supplied terminal preconditioner.
- const void * matvec_data

  Data structure for the user's matvec function.
- const void * term_precon

  Data structure for the user's terminal preconditioner.

5.22.1 Detailed Description

Data structure for the implemenation of the Krylov Multi-Space (KMS) Method.

C-style object to be used in conjunction with the Krylov Multi-Space (KMS) Algorithm to iteratively solve non-symmetric, indefinite linear systems. This method was inspired by the Flexible GMRES (FGMRES) and Recursive GMRES (GMRESR) methods proposed by Saad (1993) and Vorst and Vuik (1991), respectively. The idea behind this method is to recursively call FGMRES to solve a linear system with pregressively smaller Krylov Subspaces built by a Right-Preconditioned GMRES algorithm. Thus creating a "V-cycle" of iteration similar to that seen in Multi-Grid algorithms.

5.22.2 Member Data Documentation

5.22.2.1 int KMS_DATA::level = 0

Current level in the recursion.

5.22.2.2 int KMS_DATA::max_level = 0

Maximum allowable recursion levels (Default = 0 -> GMRES, Max = 5)

5.22.2.3 int KMS_DATA::restart = -1

Restart parameter for the outer iterates (Default = 20, Max = N)

5.22.2.4 int KMS_DATA::maxit = 0

Maximum allowable iterations for the outer steps.

5.22.2.5 int KMS_DATA::inner_iter = 0

Number of inner steps taken.

5.22.2.6 int KMS_DATA::outer_iter = 0

Number of outer steps taken.

5.22.2.7 int KMS_DATA::total_iter = 0

Total number of iterations in all steps.

5.22.2.8 double KMS_DATA::outer_reltol = 1e-6

Relative residual tolerance for outer steps (Default = 1e-6)

5.22.2.9 double KMS_DATA::outer_abstol = 1e-6

Absolute residual tolerance for outer steps (Default = 1e-6)

5.22.2.10 double KMS_DATA::inner_reltol = 0.1

Residual tolerance for inner steps made relative to outer steps (Default = 0.1)

5.22.2.11 bool KMS_DATA::Output_outer = true

True = Print the outer steps residuals.

5.22.2.12 bool KMS_DATA::Output_inner = false

True = Print the inner steps residuals.

5.22.2.13 GMRESRP_DATA KMS_DATA::gmres_out

Data structure for the outer steps.

5.22.2.14 std::vector<GMRESRP_DATA> KMS_DATA::gmres_in

Data structures for each recursion level.

5.22.2.15 int(* KMS_DATA::matvec) (const Matrix< double > &x, Matrix< double > &Ax, const void *matvec_data)

User supplied matrix-vector product function.

5.22.2.16 int(* KMS_DATA::terminal_precon) (const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

Optional user supplied terminal preconditioner.

5.22.2.17 const void* KMS_DATA::matvec_data

Data structure for the user's matvec function.

5.22.2.18 const void* KMS_DATA::term_precon

Data structure for the user's terminal preconditioner.

The documentation for this struct was generated from the following file:

- lark.h

5.23 MAGPIE_DATA Struct Reference

MAGPIE Data Structure.

`#include <magpie.h>`

Public Attributes

- std::vector< GSTA_DATA > gsta_dat
- std::vector< mSPD_DATA > mspd_dat
- std::vector< GPAST_DATA > gpast_dat
- SYSTEM_DATA sys_dat

5.23.1 Detailed Description

MAGPIE Data Structure.

C-style object holding all information necessary to run a MAGPIE simulation. This is the data structure that will be used in other sub-routines when a mixed gas adsorption simulation needs to be run.

5.23.2 Member Data Documentation

5.23.2.1 std::vector<GSTA_DATA> MAGPIE_DATA::gsta_dat

5.23.2.2 std::vector<mSPD_DATA> MAGPIE_DATA::mspd_dat

5.23.2.3 std::vector<GPAST_DATA> MAGPIE_DATA::gpast_dat

5.23.2.4 SYSTEM_DATA MAGPIE_DATA::sys_dat

The documentation for this struct was generated from the following file:

- magpie.h

5.24 MassBalance Class Reference

Mass Balance Object.

```
#include <shark.h>
```

Public Member Functions

- MassBalance ()

  Default Constructor.
- ~MassBalance ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List)

  Function to initialize the MassBalance object from the MasterSpeciesList.
- void Display_Info ()

  Display the mass balance information.
- void Set_Delta (int i, double v)

  Function to set the ith weight (delta) for the mass balance.
- void Set_TotalConcentration (double v)

  Set the total concentration of the mass balance to v (mol/L)
- void Set_Type (int type)

  Set the Mass Balance type to BATCH, CSTR, or PFR.
- void Set_Volume (double v)

  Set the volume of the reactor.
- void Set_FlowRate (double v)

  Set the flow rate for the CSTR or PFR.
- void Set_Area (double v)

  Set the cross sectional area for the PFR.
- void Set_TimeStep (double v)

  Set the time step for the CSTR or PFR.
- void Set_InitialConcentration (double v)

  Set the initial concentration for the mass balance.
- void Set_InletConcentration (double v)

  Set the inlet concentration for the CSTR or PFR.
- void Set_SteadyState (bool ss)

  Set the boolean for Steady-State simulation.
- void Set_ZeroInitialSolids (bool solids)

  Set the boolean for initial solids in solution.
- void Set_Name (std::string name)

  Set the name of the mass balance (i.e., Uranium, Carbonate, etc.)
- double Get_Delta (int i)

  Fetch the ith weight (i.e., delta) value.
- double Sum_Delta ()

  Sums up the delta values and returns the total (should never be zero)
- double Get_TotalConcentration ()

  Fetch the total concentration (mol/L)
- int Get_Type ()

  Fetch the reactor type.
- double Get_Volume ()

  Fetch the reactor volume.
- double Get_FlowRate ()

  Fetch the reactor flow rate.
- double Get_Area ()

  Fetch the reactor cross section area.
- double Get_TimeStep ()

  Fetch the time step.
- double Get_InitialConcentration ()

Fetch the initial concentration.

- double Get_InletConcentration ()

  Fetch the inlet concentration.

- bool isSteadyState ()

  Fetch the steady-state condition.

- bool isZeroInitialSolids ()

  Fetch the initial solids condition.

- std::string Get_Name ()

  Return name of mass balance object.

- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old)

  Evaluate the residual for the mass balance object given the log(C) concentrations.

- double Eval_IC_Residual (const Matrix< double > &x)

  Evaluate the initial residual for the unsteady mass balance object given the log(C) concentrations.

Protected Attributes

- MasterSpeciesList * List

  Pointer to a master species object.

- std::vector< double > Delta

  Vector of weights (i.e., deltas) used in the mass balance.

- double TotalConcentration

  Total concentration of specific object (mol/L)

- int Type

  Type of mass balance object (default = BATCH)

- double volume

  Volume of the reactor (L)

- double flow_rate

  Volumetric flow rate in reactor (L/hr)

- double xsec_area

  Cross sectional area in PFR configuration (m^2)

- double dt

  Time step for non-batch case (hrs)

- double InitialConcentration

  Concentration initially in the domain (mol/L)

- double InletConcentration

  Concentration in the inlet of the domain (mol/L)

- bool SteadyState

  True if running steady-state simulation.

- bool ZeroInitialSolids

  True if zero solids present for initial condition.

Private Attributes

- std::string Name

  Name designation used in mass balance.

5.24.1 Detailed Description

Mass Balance Object.

C++ style object that holds data and functions associated with mass balances of primary species in a system. The mass balances involve a total concentration (in mol/L) and a vector of weighted contributions to that total concentration from each species in the MasterSpeciesList. This object only considers mass balances in a batch type of system (i.e., not input or output of mass). However, one could inherit from this object to create mass balances for flow systems as well.

5.24.2 Constructor & Destructor Documentation

5.24.2.1 MassBalance::MassBalance ( )

Default Constructor.

5.24.2.2 MassBalance::~MassBalance ( )

Default Destructor.

5.24.3 Member Function Documentation

5.24.3.1 void MassBalance::Initialize_Object (MasterSpeciesList & *List*)

Function to initialize the MassBalance object from the MasterSpeciesList.

5.24.3.2 void MassBalance::Display_Info ( )

Display the mass balance information.

5.24.3.3 void MassBalance::Set_Delta (int *i*, double *v*)

Function to set the ith weight (delta) for the mass balance.

This function sets the weight (i.e., delta value) of the ith species in the list to the value of v. That value represents the weighting of that species in the determination of the total mass for the primary species set.

Parameters

i	index of the species in the MasterSpeciesList
v	value of the weigth (or delta) applied to the mass balance

5.24.3.4 void MassBalance::Set_TotalConcentration (double *v*)

Set the total concentration of the mass balance to v (mol/L)

5.24.3.5 void MassBalance::Set_Type (int *type*)

Set the Mass Balance type to BATCH, CSTR, or PFR.

5.24.3.6 void MassBalance::Set_Volume (double *v*)

Set the volume of the reactor.

5.24.3.7 void MassBalance::Set_FlowRate (double *v*)

Set the flow rate for the CSTR or PFR.

5.24.3.8 void MassBalance::Set_Area (double *v*)

Set the cross sectional area for the PFR.

5.24.3.9 void MassBalance::Set_TimeStep (double *v*)

Set the time step for the CSTR or PFR.

5.24.3.10 void MassBalance::Set_InitialConcentration (double *v*)

Set the initial concentration for the mass balance.

5.24.3.11 void MassBalance::Set_InletConcentration (double *v*)

Set the inlet concentration for the CSTR or PFR.

5.24.3.12 void MassBalance::Set_SteadyState (bool *ss*)

Set the boolean for Steady-State simulation.

5.24.3.13 void MassBalance::Set_ZeroInitialSolids (bool *solids*)

Set the boolean for initial solids in solution.

5.24.3.14 void MassBalance::Set_Name (std::string *name*)

Set the name of the mass balance (i.e., Uranium, Carbonate, etc.)

5.24.3.15 double MassBalance::Get_Delta (int *i*)

Fetch the ith weight (i.e., delta) value.

5.24.3.16 double MassBalance::Sum_Delta ( )

Sums up the delta values and returns the total (should never be zero)

5.24.3.17 double MassBalance::Get_TotalConcentration ( )

Fetch the total concentration (mol/L)

5.24.3.18 int MassBalance::Get_Type ( )

Fetch the reactor type.

5.24.3.19 double MassBalance::Get_Volume ( )

Fetch the reactor volume.

5.24.3.20 double MassBalance::Get_FlowRate ( )

Fetch the reactor flow rate.

5.24.3.21 double MassBalance::Get_Area ( )

Fetch the reactor cross section area.

5.24.3.22 double MassBalance::Get_TimeStep ( )

Fetch the time step.

5.24.3.23 double MassBalance::Get_InitialConcentration ( )

Fetch the initial concentration.

5.24.3.24 double MassBalance::Get_InletConcentration ( )

Fetch the inlet concentration.

5.24.3.25 bool MassBalance::isSteadyState ( )

Fetch the steady-state condition.

5.24.3.26 bool MassBalance::isZeroInitialSolids ( )

Fetch the initial solids condition.

5.24.3.27 std::string MassBalance::Get_Name ( )

Return name of mass balance object.

5.24.3.28 double MassBalance::Eval_Residual (const Matrix< double > & *x_new,* const Matrix< double > & *x_old*)

Evaluate the residual for the mass balance object given the log(C) concentrations.

This function calculates and provides the residual for this mass balance object based on the total concentration in the system and the weighted contributions from each species. Concentrations are given as the log(C) values.

Parameters

x_new	matrix of the log(C) concentration values at the current non-linear step
x_old	matrix of the old log(C) concentration values for transient simulations

5.24.3.29 double MassBalance::Eval_IC_Residual (const Matrix< double > & *x*)

Evaluate the initial residual for the unsteady mass balance object given the log(C) concentrations.

This function calculates and provides the initial residual for this mass balance object based on the initial concentration in the system and the weighted contributions from each species. Concentrations are given as the log(C) values.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.24.4 Member Data Documentation

5.24.4.1 MasterSpeciesList* MassBalance::List [protected]

Pointer to a master species object.

5.24.4.2 std::vector<double> MassBalance::Delta [protected]

Vector of weights (i.e., deltas) used in the mass balance.

5.24.4.3 double MassBalance::TotalConcentration [protected]

Total concentration of specific object (mol/L)

5.24.4.4 int MassBalance::Type [protected]

Type of mass balance object (default = BATCH)

5.24.4.5 double MassBalance::volume [protected]

Volume of the reactor (L)

5.24.4.6 double MassBalance::flow_rate [protected]

Volumetric flow rate in reactor (L/hr)

5.24.4.7 double MassBalance::xsec_area [protected]

Cross sectional area in PFR configuration (m^2)

5.24.4.8 double MassBalance::dt [protected]

Time step for non-batch case (hrs)

5.24.4.9 double MassBalance::InitialConcentration [protected]

Concentration initially in the domain (mol/L)

5.24.4.10 double MassBalance::InletConcentration [protected]

Concentration in the inlet of the domain (mol/L)

5.24.4.11 bool MassBalance::SteadyState [protected]

True if running steady-state simulation.

5.24.4.12 bool MassBalance::ZeroInitialSolids [protected]

True if zero solids present for initial condition.

5.24.4.13 std::string MassBalance::Name [private]

Name designation used in mass balance.

The documentation for this class was generated from the following file:

- shark.h

5.25 MasterSpeciesList Class Reference

Master Species List Object.

`#include <shark.h>`

Public Member Functions

- MasterSpeciesList ()
  Default constructor.
- ~MasterSpeciesList ()
  Default destructor.
- MasterSpeciesList (const MasterSpeciesList &msl)
  Copy Constructor.
- MasterSpeciesList & operator= (const MasterSpeciesList &msl)
  Equals operator.
- void set_list_size (int i)
  Function to initialize the size of the list.
- void set_species (int i, std::string formula)
  Function to register the ith species in the list based on a registered molecular formula (see mola.h)

- void set_species (int i, int charge, double enthalpy, double entropy, double energy, bool HS, bool G, std::string Phase, std::string Name, std::string Formula, std::string lin_formula)

  Function to register the ith species in the list based on custom molecule information (see mola.h)
- void DisplayInfo (int i)

  Function to display information of ith object.
- void DisplayAll ()

  Function to display all information of list.
- void DisplayConcentrations (Matrix< double > &C)

  Function to display the concentrations of species in list.
- void set_alkalinity (double alk)

  Set the alkalinity of the solution (Default = 0 M)
- int list_size ()

  Returns size of list.
- Molecule & get_species (int i)

  Returns a reference to the ith species in master list.
- int get_index (std::string name)

  Returns an integer representing location of the named species in the list.
- double charge (int i)

  Fetch and return charge of ith species in list.
- double alkalinity ()

  Fetch the value of alkalinity of the solution (mol/L)
- std::string speciesName (int i)

  Function to return the name of the ith species.
- double Eval_ChargeResidual (const Matrix< double > &x)

  Calculate charge balance residual for the electroneutrality constraint.

Protected Attributes

- int size

  Size of the list.
- std::vector< Molecule > species

  List of Molecule Objects.
- double residual_alkalinity

  Conc of strong base - conc of strong acid in solution (mol/L)

5.25.1 Detailed Description

Master Species List Object.

C++ style object that holds data and function associated with solving multi-species problems. This object contains a vector of Molecule objects from mola.h and uses those objects to help setup speciation problems that need to be solved. One of the primary functions in this object is the contribution of electroneutrality (Eval_ChargeResidual). However, we only need this constraint if the pH of our aqueous system is unknown.

5.25.2 Constructor & Destructor Documentation

5.25.2.1 MasterSpeciesList::MasterSpeciesList ( )

Default constructor.

5.25.2.2 MasterSpeciesList::~MasterSpeciesList ( )

Default destructor.

5.25.2.3 MasterSpeciesList::MasterSpeciesList (const MasterSpeciesList & *msl*)

Copy Constructor.

5.25.3 Member Function Documentation

5.25.3.1 MasterSpeciesList& MasterSpeciesList::operator= (const MasterSpeciesList & *msl*)

Equals operator.

5.25.3.2 void MasterSpeciesList::set_list_size (int *i*)

Function to initialize the size of the list.

5.25.3.3 void MasterSpeciesList::set_species (int *i,* std::string *formula*)

Function to register the ith species in the list based on a registered molecular formula (see mola.h)

5.25.3.4 void MasterSpeciesList::set_species (int *i,* int *charge,* double *enthalpy,* double *entropy,* double *energy,* bool *HS,* bool *G,* std::string *Phase,* std::string *Name,* std::string *Formula,* std::string *lin_formula*)

Function to register the ith species in the list based on custom molecule information (see mola.h)

5.25.3.5 void MasterSpeciesList::DisplayInfo (int *i*)

Function to display information of ith object.

5.25.3.6 void MasterSpeciesList::DisplayAll ( )

Function to display all information of list.

5.25.3.7 void MasterSpeciesList::DisplayConcentrations (Matrix< double > & *C*)

Function to display the concentrations of species in list.

This function will print to the console the species list in order with each species associated concentration from the matrix C. The concentrations and species list MUST be in the same order and the units of C are assumed to be mol/L.

Parameters

C	matrix of concentrations of species in the list in mol/L

5.25.3.8 void MasterSpeciesList::set_alkalinity (double *alk*)

Set the alkalinity of the solution (Default = 0 M)

This function is used to set the value of residual alkalinity used in the electroneutrality calculations. Typically, this value will be 0 M (mol/L) if all species in the system are present as variables. However, occasionally, one may want to set the alkalinity of the solution to a constant in order to restrict the pH of the solution.

Parameters

alk	Residual alkalinity in M (mol/L)

5.25.3.9 int MasterSpeciesList::list_size ( )

Returns size of list.

5.25.3.10 Molecule& MasterSpeciesList::get_species (int *i*)

Returns a reference to the ith species in master list.

This function will return a Molecule object for the ith species in the list of molecules. Once returned, the user then can operate on that molecule using the functions define in mola.h.

5.25.3.11 int MasterSpeciesList::get_index (std::string *name*)

Returns an integer representing location of the named species in the list.

5.25.3.12 double MasterSpeciesList::charge (int *i*)

Fetch and return charge of ith species in list.

5.25.3.13 double MasterSpeciesList::alkalinity ( )

Fetch the value of alkalinity of the solution (mol/L)

5.25.3.14 std::string MasterSpeciesList::speciesName (int *i*)

Function to return the name of the ith species.

5.25.3.15 double MasterSpeciesList::Eval_ChargeResidual (const Matrix< double > & *x*)

Calculate charge balance residual for the electroneutrality constraint.

This function returns the value of the residual for the electroneutrality equation in the system. Electroneutrality is based on the concentrations and charges of each species in the system so the charges of each molecule must be appropriately set. Concentrations of those species are fed into this function via the argument x, but come in as the log(C) values (i.e., x = log(C)).

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.25.4 Member Data Documentation

5.25.4.1 int MasterSpeciesList::size [protected]

Size of the list.

5.25.4.2 std::vector<Molecule> MasterSpeciesList::species [protected]

List of Molecule Objects.

5.25.4.3 double MasterSpeciesList::residual_alkalinity [protected]

Conc of strong base - conc of strong acid in solution (mol/L)

The documentation for this class was generated from the following file:

- shark.h

5.26 Matrix< T > Class Template Reference

Templated C++ Matrix Class Object (click Matrix to go to function definitions)

```
#include <macaw.h>
```

Public Member Functions

- Matrix (int rows, int columns)
  Constructor for matrix with given number of rows and columns.
- T & operator() (int i, int j)
  Access operator for the matrix element at row i and column j (e.g., aij = A(i,j))
- T operator() (int i, int j) const
  Constant access operator for the the matrix element at row i and column j.
- Matrix (const Matrix &M)
  Copy constructor for constructing a matrix as a copy of another matrix.
- Matrix & operator= (const Matrix &M)
  Equals operator for setting one matrix equal to another matrix.
- Matrix ()
  Default constructor for creating an empty matrix.
- ~Matrix ()
  Default destructor for clearing out memory.
- void set_size (int i, int j)
  Function to set/change the size of a matrix to i rows and j columns.
- void zeros ()
  Function to set/change all values in a matrix to zeros.
- void edit (int i, int j, T value)
  Function to set/change the element of a matrix at row i and column j to given value.
- int rows ()
  Function to return the number of rows in a given matrix.
- int columns ()

Function to return the number of columns in a matrix.

- T determinate ()

  Function to compute the determinate of a matrix and return that value.

- T norm ()

  Function to compute the L2-norm of a matrix and return that value.

- T sum ()

  Function to compute the sum of all elements in a matrix and return that value.

- T inner_product (const Matrix &x)

  Function to compute the inner product between this matrix and matrix x.

- Matrix & cofactor (const Matrix &M)

  Function to convert this matrix to a cofactor matrix of the given matrix M.

- Matrix operator+ (const Matrix &M)

  Operator to add this matrix and matrix M and return the new matrix result.

- Matrix operator- (const Matrix &M)

  Operator to subtract this matrix and matrix M and return the new matrix result.

- Matrix operator∗ (const T)

  Operator to multiply this matrix by a scalar T return the new matrix result.

- Matrix operator/ (const T)

  Operator to divide this matrix by a scalar T and return the new matrix result.

- Matrix operator+ (const T)

  Operator to add this matrix to a scalar T and return the new matrix result.

- Matrix operator- (const T)

  Operator to subtract this matrix to a scalar T and return the new matrix result.

- Matrix operator∗ (const Matrix &M)

  Operator to multiply this matrix and matrix M and return the new matrix result.

- Matrix outer_product (const Matrix &M)

  Operator to perform an outer product between this and M and return result.

- Matrix & transpose (const Matrix &M)
- Matrix & transpose_multiply (const Matrix &MT, const Matrix &v)

  Function to convert this matrix into the result of the given matrix M transposed and multiplied by the other given matrix v.

- Matrix & adjoint (const Matrix &M)

  Function to convert this matrix to the adjoint of the given matrix.

- Matrix & inverse (const Matrix &M)

  Function to convert this matrix to the inverse of the given matrix.

- void Display (const std::string Name)

  Function to display the contents of this matrix given a Name for the matrix.

- Matrix & tridiagonalSolve (const Matrix &A, const Matrix &b)

  Function to solve Ax=b for x if A is symmetric, tridiagonal (this->x)

- Matrix & ladshawSolve (const Matrix &A, const Matrix &d)

  Function to solve Ax=d for x if A is non-symmetric, tridiagonal (this->x)

- Matrix & tridiagonalFill (const T A, const T B, const T C, bool Spherical)

  Function to fill in this matrix with coefficients A, B, and C to form a tridiagonal matrix.

- Matrix & naturalLaplacian3D (int m)

  Function to fill out this matrix with coefficients from a 3D Laplacian function.

- Matrix & sphericalBCFill (int node, const T coeff, T variable)

  Function to fill out a column matrix with spherical specific boundary conditions.

- Matrix & ConstantICFill (const T IC)

  Function to set all values in a column matrix to a given constant.

- Matrix & SolnTransform (const Matrix &A, bool Forward)

  Function to transform the values in a column matrix from cartesian to spherical coordinates.

- T sphericalAvg (double radius, double dr, double bound, bool Dirichlet)

  Function to compute a spatial average of this column matrix in spherical coordinates.
- T IntegralAvg (double radius, double dr, double bound, bool Dirichlet)

  Function to compute a spatial average of this column matrix in spherical coordinates.
- T IntegralTotal (double dr, double bound, bool Dirichlet)

  Function to compute a spatial total of this column matrix in spherical coordinates.
- Matrix & tridiagonalVectorFill (const std::vector< T > &A, const std::vector< T > &B, const std::vector< T > &C)

  Function to fill in this matrix, in tridiagonal fashion, using the vectors of coefficients.
- Matrix & columnVectorFill (const std::vector< T > &A)

  Function to fill in a column matrix with the values of the given vector object.
- Matrix & columnProjection (const Matrix &b, const Matrix &b_old, const double dt, const double dt_old)

  Function to project a column matrix solution in time based on older state vectors.
- Matrix & dirichletBCFill (int node, const T coeff, T variable)

  Function to fill in a column matrix with all zeros except at the given node.
- Matrix & diagonalSolve (const Matrix &D, const Matrix &v)

  Function to solve the system Dx=v for x given that D is diagonal (this->x)
- Matrix & upperTriangularSolve (const Matrix &U, const Matrix &v)

  Function to solve the system Ux=v for x given that U is upper Triagular (this->x)
- Matrix & lowerTriangularSolve (const Matrix &L, const Matrix &v)

  Function to solve the system Lx=v for x given that L is lower Triagular (this->x)
- Matrix & upperHessenberg2Triangular (Matrix &b)

  Function to convert this square matrix to upper Triangular (assuming this is upper Hessenberg)
- Matrix & lowerHessenberg2Triangular (Matrix &b)

  Function to convert this square matrix to lower Triangular (assuming this is lower Hessenberg)
- Matrix & upperHessenbergSolve (const Matrix &H, const Matrix &v)

  Function to solve the system Hx=v for x given that H is upper Hessenberg (this->x)
- Matrix & lowerHessenbergSolve (const Matrix &H, const Matrix &v)

  Function to solve the system Hx=v for x given that H is lower Hessenberg (this->x)
- Matrix & qrSolve (const Matrix &M, const Matrix &b)

  Function to solve the system Mx=b using QR factorization for x given that M is invertable.
- Matrix & columnExtract (int j, const Matrix &M)

  Function to set this column matrix to the jth column of the given matrix M.
- Matrix & rowExtract (int i, const Matrix &M)

  Function to set this row matrix to the ith row of the given matrix M.
- Matrix & columnReplace (int j, const Matrix &v)

  Function to this matrices' jth column with the given column matrix v.
- Matrix & rowReplace (int i, const Matrix &v)

  Function to this matrices' ith row with the given row matrix v.
- void rowShrink ()

  Function to delete the last row of this matrix.
- void columnShrink ()

  Function to delete the last column of this matrix.
- void rowExtend (const Matrix &v)

  Function to add the row matrix v to the end of this matrix.
- void columnExtend (const Matrix &v)

  Function to add the column matrix v to the end of this matrix.

Protected Attributes

- int num_rows

  Number of rows of the matrix.
- int num_cols

  Number of columns of the matrix.
- std::vector< T > Data

  Storage vector for the elements of the matrix.

5.26.1 Detailed Description

template<class T>
class Matrix< T >

Templated C++ Matrix Class Object (click Matrix to go to function definitions)

C++ templated class object containing many different functions, actions, and solver routines associated with Dense Matrices. Operator overloads are also provided to give the user a more natural way of operating matrices on other matrices or scalars. These operator overloads are especially useful for reducing the amount of code needed to be written when working with matrix-based problems.

5.26.2 Constructor & Destructor Documentation

5.26.2.1 template<class T > Matrix< T >::Matrix (int *rows*, int *columns*)

Constructor for matrix with given number of rows and columns.

5.26.2.2 template<class T > Matrix< T >::Matrix (const Matrix< T > & *M*)

Copy constructor for constructing a matrix as a copy of another matrix.

5.26.2.3 template<class T > Matrix< T >::Matrix ( )

Default constructor for creating an empty matrix.

5.26.2.4 template<class T > Matrix< T >::~Matrix ( )

Default destructor for clearing out memory.

5.26.3 Member Function Documentation

5.26.3.1 template<class T > T & Matrix< T >::operator() (int *i*, int *j*)

Access operator for the matrix element at row i and column j (e.g., aij = A(i,j))

5.26.3.2 template<class T > T Matrix< T >::operator() (int *i*, int *j*) const

Constant access operator for the the matrix element at row i and column j.

5.26.3.3 template<class T > Matrix< T > & Matrix< T >::operator= (const Matrix< T > & *M*)

Equals operator for setting one matrix equal to another matrix.

5.26.3.4 template<class T > void Matrix< T >::set_size (int *i*, int *j*)

Function to set/change the size of a matrix to i rows and j columns.

5.26.3.5 template<class T > void Matrix< T >::zeros ( )

Function to set/change all values in a matrix to zeros.

5.26.3.6 template<class T> void Matrix< T >::edit (int *i*, int *j*, T *value*)

Function to set/change the element of a matrix at row i and column j to given value.

5.26.3.7 template<class T > int Matrix< T >::rows ( )

Function to return the number of rows in a given matrix.

5.26.3.8 template<class T > int Matrix< T >::columns ( )

Function to return the number of columns in a matrix.

5.26.3.9 template<class T > T Matrix< T >::determinate ( )

Function to compute the determinate of a matrix and return that value.

5.26.3.10 template<class T > T Matrix< T >::norm ( )

Function to compute the L2-norm of a matrix and return that value.

5.26.3.11 template<class T > T Matrix< T >::sum ( )

Function to compute the sum of all elements in a matrix and return that value.

5.26.3.12 template<class T > T Matrix< T >::inner_product (const Matrix< T > & *x*)

Function to compute the inner product between this matrix and matrix x.

5.26.3.13 template<class T > Matrix< T > & Matrix< T >::cofactor (const Matrix< T > & *M*)

Function to convert this matrix to a cofactor matrix of the given matrix M.

5.26.3.14 template<class T > Matrix< T > Matrix< T >::operator+ (const Matrix< T > & *M*)

Operator to add this matrix and matrix M and return the new matrix result.

5.26.3.15 template<class T> Matrix<T> Matrix<T>::operator- (const Matrix<T> & *M*)

Operator to subtract this matrix and matrix M and return the new matrix result.

5.26.3.16 template<class T> Matrix<T> Matrix<T>::operator* (const T *a*)

Operator to multiply this matrix by a scalar T return the new matrix result.

5.26.3.17 template<class T> Matrix<T> Matrix<T>::operator/ (const T *a*)

Operator to divide this matrix by a scalar T and return the new matrix result.

5.26.3.18 template<class T> Matrix<T> Matrix<T>::operator+ (const T *a*)

Operator to add this matrix to a scalar T and return the new matrix result.

5.26.3.19 template<class T> Matrix<T> Matrix<T>::operator- (const T *a*)

Operator to subtract this matrix to a scalar T and return the new matrix result.

5.26.3.20 template<class T> Matrix<T> Matrix<T>::operator* (const Matrix<T> & *M*)

Operator to multiply this matrix and matrix M and return the new matrix result.

5.26.3.21 template<class T> Matrix<T> Matrix<T>::outer_product (const Matrix<T> & *M*)

Operator to perform an outer product between this and M and return result.

5.26.3.22 template<class T> Matrix<T> & Matrix<T>::transpose (const Matrix<T> & *M*)

Function to convert this matrix to the transpose of the given matrix M

5.26.3.23 template<class T> Matrix<T> & Matrix<T>::transpose_multiply (const Matrix<T> & *MT,* const Matrix<T> & *v*)

Function to convert this matrix into the result of the given matrix M transposed and multiplied by the other given matrix v.

5.26.3.24 template<class T> Matrix<T> & Matrix<T>::adjoint (const Matrix<T> & *M*)

Function to convert this matrix to the adjoint of the given matrix.

5.26.3.25 template<class T> Matrix<T> & Matrix<T>::inverse (const Matrix<T> & *M*)

Function to convert this matrix to the inverse of the given matrix.

5.26.3.26 template<class T> void Matrix<T>::Display (const std::string *Name*)

Function to display the contents of this matrix given a Name for the matrix.

5.26.3.27 template<class T > Matrix< T > & Matrix< T >::tridiagonalSolve (const Matrix< T > & *A,* const Matrix< T > & *b*)

Function to solve Ax=b for x if A is symmetric, tridiagonal (this->x)

5.26.3.28 template<class T > Matrix< T > & Matrix< T >::ladshawSolve (const Matrix< T > & *A,* const Matrix< T > & *d*)

Function to solve Ax=d for x if A is non-symmetric, tridiagonal (this->x)

5.26.3.29 template<class T> Matrix< T > & Matrix< T >::tridiagonalFill (const T *A,* const T *B,* const T *C,* bool *Spherical*)

Function to fill in this matrix with coefficients A, B, and C to form a tridiagonal matrix.

This function fills in the diagonal elements of a square matrix with coefficient B, upper diagonal with C, and lower diagonal with A. The boolean will apply a transformation to those coefficients, if the problem happens to stem from 1-D diffusion in spherical coordinates.

5.26.3.30 template<class T > Matrix< T > & Matrix< T >::naturalLaplacian3D (int *m*)

Function to fill out this matrix with coefficients from a 3D Laplacian function.

This function will fill out the coefficients of the matrix with the coefficients that stem from discretizing a 3D Laplacian on a natural grid with 2nd order finite differences.

5.26.3.31 template<class T> Matrix< T > & Matrix< T >::sphericalBCFill (int *node,* const T *coeff,* T *variable*)

Function to fill out a column matrix with spherical specific boundary conditions.

This function will fille out a column matrix with zeros at all nodes expect for the node indicated. That node's value will be the product of the node id with the coeff and variable values given.

5.26.3.32 template<class T> Matrix< T > & Matrix< T >::ConstantICFill (const T *IC*)

Function to set all values in a column matrix to a given constant.

5.26.3.33 template<class T > Matrix< T > & Matrix< T >::SolnTransform (const Matrix< T > & *A,* bool *Forward*)

Function to transform the values in a column matrix from cartesian to spherical coordinates.

5.26.3.34 template<class T > T Matrix< T >::sphericalAvg (double *radius,* double *dr,* double *bound,* bool *Dirichlet*)

Function to compute a spatial average of this column matrix in spherical coordinates.

This function is used to compute an average value of a variable, represented in this column matrix, by integrating over the domain of the sphere. (Assumes you have variable value at center node)

Parameters

radius	radius of the sphere
dr	space between each node
bound	value of the variable at the boundary
Dirichlet	True if problem has a Dirichlet BC, False if Neumann

5.26.3.35 template<class T > T Matrix< T >::IntegralAvg (double *radius,* double *dr,* double *bound,* bool *Dirichlet*)

Function to compute a spatial average of this column matrix in spherical coordinates.

This function is used to compute an average value of a variable, represented in this column matrix, by integrating over the domain of the sphere. (Assumes you DO NOT have variable value at center node)

Parameters

radius	radius of the sphere
dr	space between each node
bound	value of the variable at the boundary
Dirichlet	True if problem has a Dirichlet BC, False if Neumann

5.26.3.36 template<class T > T Matrix< T >::IntegralTotal (double *dr,* double *bound,* bool *Dirichlet*)

Function to compute a spatial total of this column matrix in spherical coordinates.

This function is used to compute an average value of a variable, represented in this column matrix, by integrating over the domain of the sphere. (Assumes you DO NOT have variable value at center node)

Parameters

dr	space between each node
bound	value of the variable at the boundary
Dirichlet	True if problem has a Dirichlet BC, False if Neumann

5.26.3.37 template<class T> Matrix< T > & Matrix< T >::tridiagonalVectorFill (const std::vector< T > & *A,* const std::vector< T > & *B,* const std::vector< T > & *C*)

Function to fill in this matrix, in tridiagonal fashion, using the vectors of coefficients.

5.26.3.38 template<class T> Matrix< T > & Matrix< T >::columnVectorFill (const std::vector< T > & *A*)

Function to fill in a column matrix with the values of the given vector object.

5.26.3.39 template<class T > Matrix< T > & Matrix< T >::columnProjection (const Matrix< T > & *b,* const Matrix< T > & *b_old,* const double *dt,* const double *dt_old*)

Function to project a column matrix solution in time based on older state vectors.

This function is used in finch.h to form Matrix u_star. It uses the size of the current step and old step, dt and dt_old respectively, to form an approximation for the next state. The current state and olde state of the variables are passed as b and b_old respectively.

5.26.3.40 template<class T> Matrix< T > & Matrix< T >::dirichletBCFill (int *node,* const T *coeff,* T *variable*)

Function to fill in a column matrix with all zeros except at the given node.

Similar to sphericalBCFill, this function will set the values of all elements in the column matrix to zero except at the given node, where the value is set to the product of coeff and variable. This is often used to set BCs in finch.h or other related files/simulations.

5.26.3.41 template<class T > Matrix< T > & Matrix< T >::diagonalSolve (const Matrix< T > & *D,* const Matrix< T > & *v*)

Function to solve the system Dx=v for x given that D is diagonal (this->x)

5.26.3.42 template<class T > Matrix< T > & Matrix< T >::upperTriangularSolve (const Matrix< T > & *U,* const Matrix< T > & *v*)

Function to solve the system Ux=v for x given that U is upper Triagular (this->x)

5.26.3.43 template<class T > Matrix< T > & Matrix< T >::lowerTriangularSolve (const Matrix< T > & *L,* const Matrix< T > & *v*)

Function to solve the system Lx=v for x given that L is lower Triagular (this->x)

5.26.3.44 template<class T > Matrix< T > & Matrix< T >::upperHessenberg2Triangular (Matrix< T > & *b*)

Function to convert this square matrix to upper Triangular (assuming this is upper Hessenberg)

During this transformation, a column vector (b) is also being transformed to represent the BCs in a linear system. This algorithm uses Givens Rotations to efficiently convert the upper Hessenberg matrix to an upper triangular matrix.

5.26.3.45 template<class T > Matrix< T > & Matrix< T >::lowerHessenberg2Triangular (Matrix< T > & *b*)

Function to convert this square matrix to lower Triangular (assuming this is lower Hessenberg)

During this transformation, a column vector (b) is also being transformed to represent the BCs in a linear system. This algorithm uses Givens Rotations to efficiently convert the lower Hessenberg matrix to an lower triangular matrix.

5.26.3.46 template<class T > Matrix< T > & Matrix< T >::upperHessenbergSolve (const Matrix< T > & *H,* const Matrix< T > & *v*)

Function to solve the system Hx=v for x given that H is upper Hessenberg (this->x)

5.26.3.47 template<class T > Matrix< T > & Matrix< T >::lowerHessenbergSolve (const Matrix< T > & *H,* const Matrix< T > & *v*)

Function to solve the system Hx=v for x given that H is lower Hessenberg (this->x)

5.26.3.48 template<class T > Matrix< T > & Matrix< T >::qrSolve (const Matrix< T > & *M,* const Matrix< T > & *b*)

Function to solve the system Mx=b using QR factorization for x given that M is invertable.

5.26.3.49 template<class T > Matrix< T > & Matrix< T >::columnExtract (int *j,* const Matrix< T > & *M*)

Function to set this column matrix to the jth column of the given matrix M.

5.26.3.50 template<class T > Matrix< T > & Matrix< T >::rowExtract (int *i,* const Matrix< T > & *M*)

Function to set this row matrix to the ith row of the given matrix M.

5.26.3.51 template<class T > Matrix< T > & Matrix< T >::columnReplace (int *j*, const Matrix< T > & *v*)

Function to this matrices' jth column with the given column matrix v.

5.26.3.52 template<class T > Matrix< T > & Matrix< T >::rowReplace (int *i*, const Matrix< T > & *v*)

Function to this matrices' ith row with the given row matrix v.

5.26.3.53 template<class T > void Matrix< T >::rowShrink ( )

Function to delete the last row of this matrix.

5.26.3.54 template<class T > void Matrix< T >::columnShrink ( )

Function to delete the last column of this matrix.

5.26.3.55 template<class T > void Matrix< T >::rowExtend (const Matrix< T > & *v*)

Function to add the row matrix v to the end of this matrix.

5.26.3.56 template<class T > void Matrix< T >::columnExtend (const Matrix< T > & *v*)

Function to add the column matrix v to the end of this matrix.

5.26.4 Member Data Documentation

5.26.4.1 template<class T> int Matrix< T >::num_rows [protected]

Number of rows of the matrix.

5.26.4.2 template<class T> int Matrix< T >::num_cols [protected]

Number of columns of the matrix.

5.26.4.3 template<class T> std::vector<T> Matrix< T >::Data [protected]

Storage vector for the elements of the matrix.

The documentation for this class was generated from the following file:

- macaw.h

5.27 MIXED_GAS Struct Reference

Data structure holding information necessary for computing mixed gas properties.

`#include <egret.h>`

Public Attributes

- int N

  Given: Total number of gas species.
- bool CheckMolefractions = true

  Given: True = Check Molefractions for errors.
- double total_pressure

  Given: Total gas pressure (kPa)
- double gas_temperature

  Given: Gas temperature (K)
- double velocity

  Given: Gas phase velocity (cm/s)
- double char_length

  Given: Characteristic Length (cm)
- std::vector< double > molefraction

  Given: Gas molefractions of each species (-)
- double total_density

  Calculated: Total gas density (g/cm^3) {use RE3}.
- double total_dyn_vis

  Calculated: Total dynamic viscosity (g/cm/s)
- double kinematic_viscosity

  Calculated: Kinematic viscosity (cm^2/s)
- double total_molecular_weight

  Calculated: Total molecular weight (g/mol)
- double total_specific_heat

  Calculated: Total specific heat (J/g/K)
- double Reynolds

  Calculated: Value of the Reynold's number (-)
- Matrix< double > binary_diffusion

  Calculated: Tensor matrix of binary gas diffusivities (cm^2/s)
- std::vector< PURE_GAS > species_dat

  Vector of the pure gas info of all species.

5.27.1 Detailed Description

Data structure holding information necessary for computing mixed gas properties.

C-style object holding the mixed gas information necessary for performing gas dynamic simulations. This object works in conjunction with the calculate_variables function and uses the kinetic theory of gases to estimate mixed gas properties.

5.27.2 Member Data Documentation

5.27.2.1 int MIXED_GAS::N

Given: Total number of gas species.

5.27.2.2 bool MIXED_GAS::CheckMolefractions = true

Given: True = Check Molefractions for errors.

5.27.2.3 double MIXED_GAS::total_pressure

Given: Total gas pressure (kPa)

5.27.2.4 double MIXED_GAS::gas_temperature

Given: Gas temperature (K)

5.27.2.5 double MIXED_GAS::velocity

Given: Gas phase velocity (cm/s)

5.27.2.6 double MIXED_GAS::char_length

Given: Characteristic Length (cm)

5.27.2.7 std::vector<double> MIXED_GAS::molefraction

Given: Gas molefractions of each species (-)

5.27.2.8 double MIXED_GAS::total_density

Calculated: Total gas density (g/cm^3) {use RE3}.

5.27.2.9 double MIXED_GAS::total_dyn_vis

Calculated: Total dynamic viscosity (g/cm/s)

5.27.2.10 double MIXED_GAS::kinematic_viscosity

Calculated: Kinematic viscosity (cm^2/s)

5.27.2.11 double MIXED_GAS::total_molecular_weight

Calculated: Total molecular weight (g/mol)

5.27.2.12 double MIXED_GAS::total_specific_heat

Calculated: Total specific heat (J/g/K)

5.27.2.13 double MIXED_GAS::Reynolds

Calculated: Value of the Reynold's number (-)

5.27.2.14 Matrix<double> MIXED_GAS::binary_diffusion

Calculated: Tensor matrix of binary gas diffusivities (cm^2/s)

5.27.2.15 std::vector<PURE_GAS> MIXED_GAS::species_dat

Vector of the pure gas info of all species.

The documentation for this struct was generated from the following file:

- egret.h

5.28 Molecule Class Reference

C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

```
#include <mola.h>
```

Public Member Functions

- Molecule ()

  Default Constructor (builds an empty molecule object)
- ~Molecule ()

  Default Destructor (clears out memory)
- Molecule (int charge, double enthalpy, double entropy, double energy, bool HS, bool G, std::string Phase, std::string Name, std::string Formula, std::string lin_formula)

  Construct any molecule from the available information.
- void Register (int charge, double enthalpy, double entropy, double energy, bool HS, bool G, std::string Phase, std::string Name, std::string Formula, std::string lin_formula)

  Function to register this molecule from the available information.
- void Register (std::string formula)

  Function to register this molecule based on the given formula (if formula is in library)
- void setFormula (std::string form)

  Sets the formula for a molecule.
- void calculateMolarWeight ()

  Forces molecule to calculate its molar weight.
- void calculateMolarVolume ()

  Force molecule to calculate van der Waals volume.
- void calculateMolarArea ()

  Force molecule to calculate van der Waals area.
- void setMolarWeigth (double mw)

  Set the molar weight of species to a constant.
- void setMolarVolume (double v)

  Set the van der Waals volume of the species to a constant.
- void setMolarArea (double a)

  Set the van der Waals area of the species to a constant.
- void editCharge (int c)

  Change the ionic charge of a molecule.
- void editOneOxidationState (int state, std::string Symbol)

  Change oxidation state of one of the given atoms (always first match found)
- void editAllOxidationStates (int state, std::string Symbol)

  Change oxidation state of all of the given atoms.
- void calculateAvgOxiState (std::string Symbol)

  Function to calculate the average oxidation state of the atoms.

- void editEnthalpy (double enthalpy)

  Edit the molecules formation enthalpy (J/mol)
- void editEntropy (double entropy)

  Edit the molecules formation entropy (J/K/mol)
- void editHS (double H, double S)

  Edit both formation enthalpy and entropy.
- void editEnergy (double energy)

  Edit Gibb's formation energy.
- void removeOneAtom (std::string Symbol)

  Removes one atom of the symbol given (always the first atom found)
- void removeAllAtoms (std::string Symbol)

  Removes all atoms of the symbol given.
- int Charge ()

  Return the charge of the molecule.
- double MolarWeight ()

  Return the molar weight of the molecule.
- double MolarVolume ()

  Return the van der Waals volume of the molecule.
- double MolarArea ()

  Return the van der Waals area of the molecule.
- bool HaveHS ()

  Returns true if enthalpy and entropy are known.
- bool HaveEnergy ()

  Returns true if the Gibb's energy is known.
- bool isRegistered ()

  Returns true if the molecule has been registered.
- double Enthalpy ()

  Return the formation enthalpy of the molecule.
- double Entropy ()

  Return the formation entropy of the molecule.
- double Energy ()

  Return the Gibb's formation energy of the molecule.
- std::string MoleculeName ()

  Return the common name of the molecule.
- std::string MolecularFormula ()

  Return the molecular formula of the molecule.
- std::string MoleculePhase ()

  Return the phase of the molecule.
- int MoleculePhaseID ()

  Return the enum phase ID of the molecule.
- void DisplayInfo ()

  Function to display molecule information.

Protected Attributes

- int charge

  Ionic charge of the molecule - specified.
- double molar_weight

  Molar weight of the molecule (g/mol) - determined from atoms or specified.
- double molar_volume

van der Waals Volume of the molecule (cubic angstroms) - determined from atoms or specified

- double molar_area

  van der Waals Area of the molecule (square angstroms) - determined from atoms or specified
- double formation_enthalpy

  Enthalpy of formation of the molecule (J/mol) - constant.
- double formation_entropy

  Entropy of formation of the molecule (J/K/mol) - constant.
- double formation_energy

  Gibb's energy of formation (J/mol) - given.
- std::string Phase

  Phase of the molecule (i.e. Solid, Liquid, Aqueous, Gas...)
- int PhaseID

  Phase ID of the molecule (from the enum)
- std::vector< Atom > atoms

  Atoms which make up the molecule - based on Formula.

Private Attributes

- std::string Name

  Name of the Molecule - Common Name (i.e. H2O = Water)
- std::string Formula

  Formula for the molecule - specified (i.e. H2O)
- bool haveG

  True = given Gibb's energy of formation.
- bool haveHS

  True = give enthalpy and entropy of formation.
- bool registered

  True = the object was registered.

5.28.1 Detailed Description

C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

C++ Class Object that stores information and certain operations associated with molecules. Registered molecules are built up from their respective atoms so that the molecule can keep track of information such as molecular weigth and oxidation states. Primarily, this object is used in conjunction with shark.h to formulate the system of equations necessary for solving speciation type problems in aqueous systems. However, this object is generalized enough to be of use in RedOx calculations, reaction formulation, and molecular transformations.

All information for a molecule should be initialized prior to performing operations with or on the object. There are several molecules already defined for construction by the formulas listed at the top of this section.

5.28.2 Constructor & Destructor Documentation

5.28.2.1 Molecule::Molecule ( )

Default Constructor (builds an empty molecule object)

5.28.2.2 Molecule::~Molecule ( )

Default Destructor (clears out memory)

5.28.2.3 Molecule::Molecule (int *charge*, double *enthalpy*, double *entropy*, double *energy*, bool *HS*, bool *G*, std::string *Phase*, std::string *Name*, std::string *Formula*, std::string *lin_formula*)

Construct any molecule from the available information.

This constructor will build a user defined custom molecule.

Parameters

charge	the ionic charge of the molecule
enthalpy	the standard formation enthalpy of the molecule (J/mol)
entropy	the standard formation entropy of the molecule (J/K/mol)
energy	the standard Gibb's Free Energy of formation of the molecule (J/mol)
HS	boolean to be set to true if enthalpy and entropy were given
G	boolean to be set to true if the energy was given
Phase	string denoting molecule's phase (i.e., Liquid, Aqueous, Gas, Solid)
Name	string denoting the common name of the molecule (i.e., H2O -> Water)
Formula	string denoting the formula by which the molecule is referened (i.e., Cl - (aq))
lin_formula	string denoting all the atoms in the molecule (i.e., UO2(OH)2 -> UO4H2)

5.28.3 Member Function Documentation

5.28.3.1 void Molecule::Register (int *charge,* double *enthalpy,* double *entropy,* double *energy,* bool *HS,* bool *G,* std::string *Phase,* std::string *Name,* std::string *Formula,* std::string *lin_formula*)

Function to register this molecule from the available information.

This function will build a user defined custom molecule.

Parameters

charge	the ionic charge of the molecule
enthalpy	the standard formation enthalpy of the molecule (J/mol)
entropy	the standard formation entropy of the molecule (J/K/mol)
energy	the standard Gibb's Free Energy of formation of the molecule (J/mol)
HS	boolean to be set to true if enthalpy and entropy were given
G	boolean to be set to true if the energy was given
Phase	string denoting molecule's phase (i.e., Liquid, Aqueous, Gas, Solid)
Name	string denoting the common name of the molecule (i.e., H2O -> Water)
Formula	string denoting the formula by which the molecule is referened (i.e., Cl - (aq))
lin_formula	string denoting all the atoms in the molecule (i.e., UO2(OH)2 -> UO4H2)

5.28.3.2 void Molecule::Register (std::string *formula*)

Function to register this molecule based on the given formula (if formula is in library)

This function will create this molecule object from the given formula, but only if that formula is already registered in the library. See the top of this class section for a list of all currently registered formulas.

Note

The formula is checked against a known set of molecules inside of the registration function If the formula is unknown, an error will print to the screen. Unknown molecules should be registered using the full registration function from above. The library can only be added to by a going in and editing the source code of the mola.cpp file. However, this is a relatively simple task.

5.28.3.3 void Molecule::setFormula (std::string *form*)

Sets the formula for a molecule.

5.28.3.4 void Molecule::calculateMolarWeight ( )

Forces molecule to calculate its molar weight.

5.28.3.5 void Molecule::calculateMolarVolume ( )

Force molecule to calculate van der Waals volume.

5.28.3.6 void Molecule::calculateMolarArea ( )

Force molecule to calculate van der Waals area.

5.28.3.7 void Molecule::setMolarWeigth (double *mw*)

Set the molar weight of species to a constant.

5.28.3.8 void Molecule::setMolarVolume (double *v*)

Set the van der Waals volume of the species to a constant.

5.28.3.9 void Molecule::setMolarArea (double *a*)

Set the van der Waals area of the species to a constant.

5.28.3.10 void Molecule::editCharge (int *c*)

Change the ionic charge of a molecule.

5.28.3.11 void Molecule::editOneOxidationState (int *state*, std::string *Symbol*)

Change oxidation state of one of the given atoms (always first match found)

This function will search the list of Atoms that make up the Molecule for the given atomic Symbol. It will change the oxidation state of the first found matching atom with the given state.

5.28.3.12 void Molecule::editAllOxidationStates (int *state*, std::string *Symbol*)

Change oxidation state of all of the given atoms.

This function will search the list of Atoms that make up the Molecule for the given atomic Symbol. It will change the oxidation state of all found matching atoms with the given state.

5.28.3.13 void Molecule::calculateAvgOxiState (std::string *Symbol*)

Function to calculate the average oxidation state of the atoms.

This function search the atoms in the molecule for the matching atomic Symbol. It then looks at all oxidation states of that atom in the molecule and then sets all the oxidation states of that atom to the average value calculated.

5.28.3.14 void Molecule::editEnthalpy (double *enthalpy*)

Edit the molecules formation enthalpy (J/mol)

5.28.3.15 void Molecule::editEntropy (double *entropy*)

Edit the molecules formation entropy (J/K/mol)

5.28.3.16 void Molecule::editHS (double *H*, double *S*)

Edit both formation enthalpy and entropy.

This function will change or set the values for formation enthalpy (J/mol) and formation entropy (J/K/mol) based on the given values.

Parameters

H	formation enthalpy (J/mol)
S	formation entropy (J/K/mol)

5.28.3.17 void Molecule::editEnergy (double *energy*)

Edit Gibb's formation energy.

5.28.3.18 void Molecule::removeOneAtom (std::string *Symbol*)

Removes one atom of the symbol given (always the first atom found)

5.28.3.19 void Molecule::removeAllAtoms (std::string *Symbol*)

Removes all atoms of the symbol given.

5.28.3.20 int Molecule::Charge ( )

Return the charge of the molecule.

5.28.3.21 double Molecule::MolarWeight ( )

Return the molar weight of the molecule.

5.28.3.22 double Molecule::MolarVolume ( )

Return the van der Waals volume of the molecule.

5.28.3.23 double Molecule::MolarArea ( )

Return the van der Waals area of the molecule.

5.28.3.24 bool Molecule::HaveHS ( )

Returns true if enthalpy and entropy are known.

5.28.3.25 bool Molecule::HaveEnergy ( )

Returns true if the Gibb's energy is known.

5.28.3.26 bool Molecule::isRegistered ( )

Returns true if the molecule has been registered.

5.28.3.27 double Molecule::Enthalpy ( )

Return the formation enthalpy of the molecule.

5.28.3.28 double Molecule::Entropy ( )

Return the formation entropy of the molecule.

5.28.3.29 double Molecule::Energy ( )

Return the Gibb's formation energy of the molecule.

5.28.3.30 std::string Molecule::MoleculeName ( )

Return the common name of the molecule.

5.28.3.31 std::string Molecule::MolecularFormula ( )

Return the molecular formula of the molecule.

5.28.3.32 std::string Molecule::MoleculePhase ( )

Return the phase of the molecule.

5.28.3.33 int Molecule::MoleculePhaseID ( )

Return the enum phase ID of the molecule.

5.28.3.34 void Molecule::DisplayInfo ( )

Function to display molecule information.

5.28.4 Member Data Documentation

5.28.4.1 int Molecule::charge [protected]

Ionic charge of the molecule - specified.

5.28.4.2 double Molecule::molar_weight [protected]

Molar weight of the molecule (g/mol) - determined from atoms or specified.

5.28.4.3 double Molecule::molar_volume [protected]

van der Waals Volume of the molecule (cubic angstroms) - determined from atoms or specified

5.28.4.4 double Molecule::molar_area [protected]

van der Waals Area of the molecule (square angstroms) - determined from atoms or specified

5.28.4.5 double Molecule::formation_enthalpy [protected]

Enthalpy of formation of the molecule (J/mol) - constant.

5.28.4.6 double Molecule::formation_entropy [protected]

Entropy of formation of the molecule (J/K/mol) - constant.

5.28.4.7 double Molecule::formation_energy [protected]

Gibb's energy of formation (J/mol) - given.

5.28.4.8 std::string Molecule::Phase [protected]

Phase of the molecule (i.e. Solid, Liquid, Aqueous, Gas...)

5.28.4.9 int Molecule::PhaseID [protected]

Phase ID of the molecule (from the enum)

5.28.4.10 std::vector<Atom> Molecule::atoms [protected]

Atoms which make up the molecule - based on Formula.

5.28.4.11 std::string Molecule::Name [private]

Name of the Molecule - Common Name (i.e. H2O = Water)

5.28.4.12 std::string Molecule::Formula [private]

Formula for the molecule - specified (i.e. H2O)

5.28.4.13 bool Molecule::haveG [private]

True = given Gibb's energy of formation.

5.28.4.14 bool Molecule::haveHS [private]

True = give enthalpy and entropy of formation.

5.28.4.15 bool Molecule::registered [private]

True = the object was registered.

The documentation for this class was generated from the following file:

- mola.h

5.29 MONKFISH_DATA Struct Reference

Primary data structure for running MONKFISH.

`#include <monkfish.h>`

Public Attributes

- unsigned long int total_steps = 0

  Total number of steps taken by the algorithm (iterations and time steps)
- double time_old = 0.0

  Old value of time in the simulation (hrs)
- double time = 0.0

  Current value of time in the simulation (hrs)
- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- bool DirichletBC = true

  False = uses film mass transfer for BC, True = Dirichlet BC.
- bool NonLinear = false

  False = Solve directly, True = Solve iteratively.
- bool haveMinMax = false

  True = know min and max fiber density, False = only know avg density (Used in ICs)
- bool MultiScale = true

  True = solve single fiber model at nodes, False = solve equilibrium at nodes.
- int level = 2

  Level of coupling between multiple scales (default = 2)
- double t_counter = 0.0

  Counter for the time output.
- double t_print

  Print output at every t_print time (hrs)
- int NumComp

  Number of species to track.
- double end_time

  Units: hours.
- double total_sorption_old

  Old total adsorption per mass of woven nest (mg/g)
- double total_sorption

  Current total adsorption per mass woven nest (mg/g)
- double single_fiber_density

  Units: g/L.
- double avg_fiber_density

  Units: g/L (Used in ICs)
- double max_fiber_density

  Units: g/L (Used in ICs)
- double min_fiber_density

  Units: g/L (Used in ICs)
- double max_porosity

  Units: -.
- double min_porosity

  Units: -.
- double domain_diameter

  Nominal diameter of the woven fiber ball - Units: cm.
- FILE * Output

  Output file pointer for printing to text file.
- double(* eval_eps)(int i, int l, const void *user_data)

Function pointer to evaluate the porosity of the woven bundle of fibers.

- double(∗ eval_rho)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the fiber density in the domain.
- double(∗ eval_Dex)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the interparticle diffusivity.
- double(∗ eval_ads)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the adsorption strength for the macro-scale.
- double(∗ eval_Ret)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the retardation coefficient for the macro-scale.
- double(∗ eval_Cex)(int i, const void ∗user_data)

  Function pointer to evaluate the exterior concentration for the domain.
- double(∗ eval_kf)(int i, const void ∗user_data)

  Function pointer to evalutate the film mass transfer coefficient for the macro-scale.
- const void ∗ user_data

  User supplied data function to evaluate the function pointers (Default = MONKFISH_DATA)
- std::vector< FINCH_DATA > finch_dat

  FINCH data structures to solve each species interparticle diffusion equation.
- std::vector< MONKFISH_PARAM > param_dat

  MONKFISH parameter data structure for each species adsorbing.
- std::vector< DOGFISH_DATA > dog_dat

  DOGFISH data structures for each node in the macro-scale problem.

5.29.1 Detailed Description

Primary data structure for running MONKFISH.

C-style object holding simulation information for MONKFISH as well as common system parameters like fiber density, fiber diameter, fiber length, etc. This object also contains function pointers to different parameter evaluation functions that can be changed to suit a particular problem. Default functions will be given, so not every user needs to override these functions. This structure also contains vectors of other objects including FINCH and DOGFISH objects to resolve the diffusion physics at both the macro- and micro-scale.

5.29.2 Member Data Documentation

5.29.2.1 unsigned long int MONKFISH_DATA::total_steps = 0

Total number of steps taken by the algorithm (iterations and time steps)

5.29.2.2 double MONKFISH_DATA::time_old = 0.0

Old value of time in the simulation (hrs)

5.29.2.3 double MONKFISH_DATA::time = 0.0

Current value of time in the simulation (hrs)

5.29.2.4 bool MONKFISH_DATA::Print2File = true

True = results to .txt; False = no printing.

5.29.2.5 bool MONKFISH_DATA::Print2Console = true

True = results to console; False = no printing.

5.29.2.6 bool MONKFISH_DATA::DirichletBC = true

False = uses film mass transfer for BC, True = Dirichlet BC.

5.29.2.7 bool MONKFISH_DATA::NonLinear = false

False = Solve directly, True = Solve iteratively.

5.29.2.8 bool MONKFISH_DATA::haveMinMax = false

True = know min and max fiber density, False = only know avg density (Used in ICs)

5.29.2.9 bool MONKFISH_DATA::MultiScale = true

True = solve single fiber model at nodes, False = solve equilibrium at nodes.

5.29.2.10 int MONKFISH_DATA::level = 2

Level of coupling between multiple scales (default = 2)

5.29.2.11 double MONKFISH_DATA::t_counter = 0.0

Counter for the time output.

5.29.2.12 double MONKFISH_DATA::t_print

Print output at every t_print time (hrs)

5.29.2.13 int MONKFISH_DATA::NumComp

Number of species to track.

5.29.2.14 double MONKFISH_DATA::end_time

Units: hours.

5.29.2.15 double MONKFISH_DATA::total_sorption_old

Old total adsorption per mass of woven nest (mg/g)

5.29.2.16 double MONKFISH_DATA::total_sorption

Current total adsorption per mass woven nest (mg/g)

5.29.2.17 double MONKFISH_DATA::single_fiber_density

Units: g/L.

5.29.2.18 double MONKFISH_DATA::avg_fiber_density

Units: g/L (Used in ICs)

5.29.2.19 double MONKFISH_DATA::max_fiber_density

Units: g/L (Used in ICs)

5.29.2.20 double MONKFISH_DATA::min_fiber_density

Units: g/L (Used in ICs)

5.29.2.21 double MONKFISH_DATA::max_porosity

Units: -.

5.29.2.22 double MONKFISH_DATA::min_porosity

Units: -.

5.29.2.23 double MONKFISH_DATA::domain_diameter

Nominal diameter of the woven fiber ball - Units: cm.

5.29.2.24 FILE* MONKFISH_DATA::Output

Output file pointer for printing to text file.

5.29.2.25 double(* MONKFISH_DATA::eval_eps) (int i, int l, const void *user_data)

Function pointer to evaluate the porosity of the woven bundle of fibers.

5.29.2.26 double(* MONKFISH_DATA::eval_rho) (int i, int l, const void *user_data)

Function pointer to evaluate the fiber density in the domain.

5.29.2.27 double(* MONKFISH_DATA::eval_Dex) (int i, int l, const void *user_data)

Function pointer to evaluate the interparticle diffusivity.

5.29.2.28 double(* MONKFISH_DATA::eval_ads) (int i, int l, const void *user_data)

Function pointer to evaluate the adsorption strength for the macro-scale.

5.29.2.29 double(∗ MONKFISH_DATA::eval_Ret) (int i, int l, const void ∗user_data)

Function pointer to evaluate the retardation coefficient for the macro-scale.

5.29.2.30 double(∗ MONKFISH_DATA::eval_Cex) (int i, const void ∗user_data)

Function pointer to evaluate the exterior concentration for the domain.

5.29.2.31 double(∗ MONKFISH_DATA::eval_kf) (int i, const void ∗user_data)

Function pointer to evalutate the film mass transfer coefficient for the macro-scale.

5.29.2.32 const void∗ MONKFISH_DATA::user_data

User supplied data function to evaluate the function pointers (Default = MONKFISH_DATA)

5.29.2.33 std::vector<FINCH_DATA> MONKFISH_DATA::finch_dat

FINCH data structures to solve each species interparticle diffusion equation.

5.29.2.34 std::vector<MONKFISH_PARAM> MONKFISH_DATA::param_dat

MONKFISH parameter data structure for each species adsorbing.

5.29.2.35 std::vector<DOGFISH_DATA> MONKFISH_DATA::dog_dat

DOGFISH data structures for each node in the macro-scale problem.

The documentation for this struct was generated from the following file:

- monkfish.h

5.30 MONKFISH_PARAM Struct Reference

Data structure for species specific information and parameters.

`#include <monkfish.h>`

Public Attributes

- double interparticle_diffusion

  Units: cm^2/hr.
- double exterior_concentration

  Units: mol/L.
- double exterior_transfer_coeff

  Units: cm/hr.
- double sorbed_molefraction

  Units: -.
- double initial_sorption

  Units: mg/g.
- double sorption_bc

  Units: mg/g.
- double intraparticle_diffusion

  Units: um^2/hr.
- double film_transfer_coeff

  Units: um/hr.
- Matrix< double > avg_sorption

  Units: mg/g.
- Matrix< double > avg_sorption_old

  Units: mg/g.
- Molecule species

  Species in the liquid phase.

5.30.1 Detailed Description

Data structure for species specific information and parameters.

C-style object to hold information associated with the different species present in the interparticle diffusion problem. Each species may have different diffusivities, mass transfer coefficients, etc. Average adsorption for each species will be held in matrix objects.

5.30.2 Member Data Documentation

5.30.2.1 double MONKFISH_PARAM::interparticle_diffusion

Units: cm^2/hr.

5.30.2.2 double MONKFISH_PARAM::exterior_concentration

Units: mol/L.

5.30.2.3 double MONKFISH_PARAM::exterior_transfer_coeff

Units: cm/hr.

5.30.2.4 double MONKFISH_PARAM::sorbed_molefraction

Units: -.

5.30.2.5 double MONKFISH_PARAM::initial_sorption

Units: mg/g.

5.30.2.6 double MONKFISH_PARAM::sorption_bc

Units: mg/g.

5.30.2.7 double MONKFISH_PARAM::intraparticle_diffusion

Units: um^2/hr.

5.30.2.8 double MONKFISH_PARAM::film_transfer_coeff

Units: um/hr.

5.30.2.9 Matrix<double> MONKFISH_PARAM::avg_sorption

Units: mg/g.

5.30.2.10 Matrix<double> MONKFISH_PARAM::avg_sorption_old

Units: mg/g.

5.30.2.11 Molecule MONKFISH_PARAM::species

Species in the liquid phase.

The documentation for this struct was generated from the following file:

- monkfish.h

5.31 mSPD_DATA Struct Reference

MSPD Data Structure.

```
#include <magpie.h>
```

Public Attributes

- double s
  Area shape factor.
- double v
  van der Waals Volume (cm^3/mol)
- double eMax
  Maximum lateral interaction energy (J/mol)
- std::vector< double > eta
  Binary interaction parameter matrix (i,j)
- double gama
  Activity coefficient calculated from mSPD.

5.31.1 Detailed Description

MSPD Data Structure.

C-Style object holding all parameter information associated with the Modified Spreading Pressure Dependent (SPD) activity model. Each species in the gas phase will have one of these objects.

5.31.2 Member Data Documentation

5.31.2.1 double mSPD_DATA::s

Area shape factor.

5.31.2.2 double mSPD_DATA::v

van der Waals Volume (cm^3/mol)

5.31.2.3 double mSPD_DATA::eMax

Maximum lateral interaction energy (J/mol)

5.31.2.4 std::vector<double> mSPD_DATA::eta

Binary interaction parameter matrix (i,j)

5.31.2.5 double mSPD_DATA::gama

Activity coefficient calculated from mSPD.

The documentation for this struct was generated from the following file:

- magpie.h

5.32 MultiligandAdsorption Class Reference

Multi-ligand Adsorption Reaction Object.

```
#include <shark.h>
```

Public Member Functions

- MultiligandAdsorption ()

  Default Constructor.
- ~MultiligandAdsorption ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int l, std::vector< int > n)

  Function to call the initialization of objects sequentially.
- void modifyDeltas (MassBalance &mbo)

  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

  Find and set the adsorbed species indices for each reaction object in each ligand object.
- int checkAqueousIndices ()

  Function to check and report errors in the aqueous species indices.
- void setActivityModelInfo (int(*act)(const Matrix< double > &logq, Matrix< double > &activity, const void *data), const void *act_data)

  Function to set the surface activity model and data pointer.
- int setAqueousIndexAuto ()

  Automatically sets the primary aqueous species index based on reactions for each ligand.
- void setActivityEnum (int act)

  Set the activity enum to the value of act.
- void setMolarFactor (int ligand, int rxn, double m)

  Set the molar factor for the rxn reaction of the ligand ligand to a value of m.
- void setVolumeFactor (int i, double v)

  Set all ith volume factors for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

  Set all ith area factors for the species list (m^2/mol)
- void setSpecificMolality (int ligand, double a)

  Set the specific molality for the ligand (mol/kg)
- void setSurfaceCharge (int ligand, double c)
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent material or particle.
- void setLigandName (int i, std::string name)

  Set the name of the ith ligand.
- void setSpecificArea (double area)

  Set the specific area of the adsorbent.
- void setTotalMass (double mass)

  Set the mass of the adsorbent.
- void setTotalVolume (double volume)

  Set the total volume of the system.
- void setSurfaceChargeBool (bool opt)

  Set the surface charge boolean.
- void setElectricPotential (double a)

  Set the surface electric potential.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

Calculates and sets the current value of ionic strength.

- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- void calculateElecticPotential (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int rxn, int ligand)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_↩ perm, int rxn, int ligand)

  Calculates the residual for the ith reaction and lth ligand in the system.
- AdsorptionReaction & getAdsorptionObject (int i)

  Return reference to the adsortpion object corresponding to ligand i.
- int getNumberLigands ()

  Get the number of ligands involved with the surface.
- int getActivityEnum ()

  Get the value of the activity enum set by user.
- double getActivity (int i)

  Get the ith activity coefficient from the matrix object.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- double getElectricPotential ()

  Get the value of the electric surface potential (V)
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getLigandName (int i)

  Get the name of the ligand object indexed by i.
- std::string getAdsorbentName ()

  Get the name of the adsorbent.

Protected Attributes

- MasterSpeciesList * List

  Pointer to the MasterSpeciesList object.
- int num_ligands

  Number of different ligands to consider.
- std::string adsorbent_name

  Name of the adsorbent.
- int(* surface_activity)(const Matrix< double > &logq, Matrix< double > &activity, const void *data)

  Pointer to a surface activity model.
- const void * activity_data

Pointer to the data structure needed for surface activities.

- int act_fun

  Enumeration to represent the choosen surface activity function.
- Matrix< double > activities

  List of the activities calculated by the activity model.
- double specific_area

  Specific surface area of the adsorbent (m^2/kg)
- double total_mass

  Total mass of the adsorbent in the system (kg)
- double total_volume

  Total volume of the system (L)
- double ionic_strength

  Ionic Strength of the system used to adjust equilibria constants (mol/L)
- double charge_density

  Surface charge density of the adsorbent used to adjust equilbria (C/m^2)
- double electric_potential

  Electric surface potential of the adsorbent used to adjust equilibria (V)
- bool IncludeSurfCharge

  True = Includes surface charging corrections, False = Does not consider surface charge.

Private Attributes

- std::vector< AdsorptionReaction > ligand_obj

  List of the ligands and reactions they have on the surface.

5.32.1 Detailed Description

Multi-ligand Adsorption Reaction Object.

C++ Object to handle data and functions associated with forumlating multi-ligand adsorption reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure. This object is made from a vector of AdsorptionReaction objects, but differentiate between different ligands that exist on the surface.

5.32.2 Constructor & Destructor Documentation

5.32.2.1 MultiligandAdsorption::MultiligandAdsorption ( )

Default Constructor.

5.32.2.2 MultiligandAdsorption::~MultiligandAdsorption ( )

Default Destructor.

5.32.3 Member Function Documentation

5.32.3.1 void MultiligandAdsorption::Initialize_Object (MasterSpeciesList & *List*, int *l*, std::vector< int > *n*)

Function to call the initialization of objects sequentially.

Function will initialize each ligand adsorption object.

Parameters

List	reference to MasterSpeciesList object
l	number of ligands on the surface
n	number of reactions for each ligand (ligands must be correctly indexed)

5.32.3.2 void MultiligandAdsorption::modifyDeltas (MassBalance & *mbo*)

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.32.3.3 int MultiligandAdsorption::setAdsorbIndices ( )

Find and set the adsorbed species indices for each reaction object in each ligand object.

This function searches through the Reaction objects in AdsorptionReaction to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.32.3.4 int MultiligandAdsorption::checkAqueousIndices ( )

Function to check and report errors in the aqueous species indices.

5.32.3.5 void MultiligandAdsorption::setActivityModelInfo (int(∗)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data) *act*, const void ∗ *act_data*)

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.32.3.6 int MultiligandAdsorption::setAqueousIndexAuto ( )

Automatically sets the primary aqueous species index based on reactions for each ligand.

This function will go through all species and all reactions in each adsorption object and automatically set the primary aqueous species index based on the stoicheometry of the reaction. It will also check and make sure that the primary aqueous index species appears opposite of the adsorbed species in the reactions. Note: This function assumes that the adsorbed indices have already been set.

5.32.3.7 void MultiligandAdsorption::setActivityEnum (int *act*)

Set the activity enum to the value of act.

5.32.3.8 void MultiligandAdsorption::setMolarFactor (int *ligand,* int *rxn,* double *m*)

Set the molar factor for the rxn reaction of the ligand ligand to a value of m.

5.32.3.9 void MultiligandAdsorption::setVolumeFactor (int *i,* double *v*)

Set all ith volume factors for the species list (cm^3/mol)

5.32.3.10 void MultiligandAdsorption::setAreaFactor (int *i,* double *a*)

Set all ith area factors for the species list (m^2/mol)

5.32.3.11 void MultiligandAdsorption::setSpecificMolality (int *ligand,* double *a*)

Set the specific molality for the ligand (mol/kg)

5.32.3.12 void MultiligandAdsorption::setSurfaceCharge (int *ligand,* double *c*)

5.32.3.13 void MultiligandAdsorption::setAdsorbentName (std::string *name*)

Set the name of the adsorbent material or particle.

5.32.3.14 void MultiligandAdsorption::setLigandName (int *i,* std::string *name*)

Set the name of the ith ligand.

5.32.3.15 void MultiligandAdsorption::setSpecificArea (double *area*)

Set the specific area of the adsorbent.

5.32.3.16 void MultiligandAdsorption::setTotalMass (double *mass*)

Set the mass of the adsorbent.

5.32.3.17 void MultiligandAdsorption::setTotalVolume (double *volume*)

Set the total volume of the system.

5.32.3.18 void MultiligandAdsorption::setSurfaceChargeBool (bool *opt*)

Set the surface charge boolean.

5.32.3.19 void MultiligandAdsorption::setElectricPotential (double *a*)

Set the surface electric potential.

5.32.3.20 void MultiligandAdsorption::calculateAreaFactors ( )

Calculates the area factors used from the van der Waals volumes.

5.32.3.21 void MultiligandAdsorption::calculateEquilibria (double *T*)

Calculates all equilibrium parameters as a function of temperature.

5.32.3.22 void MultiligandAdsorption::setChargeDensity (const Matrix< double > & *x*)

Calculates and sets the current value of charge density.

5.32.3.23 void MultiligandAdsorption::setIonicStrength (const Matrix< double > & *x*)

Calculates and sets the current value of ionic strength.

5.32.3.24 int MultiligandAdsorption::callSurfaceActivity (const Matrix< double > & *x*)

Calls the activity model and returns an int flag for success or failure.

5.32.3.25 void MultiligandAdsorption::calculateElecticPotential (double *sigma,* double *T,* double *I,* double *rel_epsilon*)

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.32.3.26 double MultiligandAdsorption::calculateEquilibriumCorrection (double *sigma,* double *T,* double *I,* double *rel_epsilon,* int *rxn,* int *ligand*)

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.32.3.27 double MultiligandAdsorption::Eval_Residual (const Matrix< double > & *x*, const Matrix< double > & *gama*, double *T*, double *rel_perm*, int *rxn*, int *ligand*)

Calculates the residual for the ith reaction and lth ligand in the system.

This function will provide a system residual for the ith reaction object involved in the lth ligand's Adsorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.32.3.28 AdsorptionReaction& MultiligandAdsorption::getAdsorptionObject (int *i*)

Return reference to the adsortpion object corresponding to ligand i.

5.32.3.29 int MultiligandAdsorption::getNumberLigands ( )

Get the number of ligands involved with the surface.

5.32.3.30 int MultiligandAdsorption::getActivityEnum ( )

Get the value of the activity enum set by user.

5.32.3.31 double MultiligandAdsorption::getActivity (int *i*)

Get the ith activity coefficient from the matrix object.

5.32.3.32 double MultiligandAdsorption::getSpecificArea ( )

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.32.3.33 double MultiligandAdsorption::getBulkDensity ( )

Calculate and return bulk density of adsorbent in system (kg/L)

5.32.3.34 double MultiligandAdsorption::getTotalMass ( )

Get the total mass of adsorbent in the system (kg)

5.32.3.35 double MultiligandAdsorption::getTotalVolume ( )

Get the total volume of the system (L)

5.32.3.36 double MultiligandAdsorption::getChargeDensity ( )

Get the value of the surface charge density (C/m^2)

5.32.3.37 double MultiligandAdsorption::getIonicStrength ( )

Get the value of the ionic strength of solution (mol/L)

5.32.3.38 double MultiligandAdsorption::getElectricPotential ( )

Get the value of the electric surface potential (V)

5.32.3.39 bool MultiligandAdsorption::includeSurfaceCharge ( )

Returns true if we are considering surface charging during adsorption.

5.32.3.40 std::string MultiligandAdsorption::getLigandName (int *i*)

Get the name of the ligand object indexed by i.

5.32.3.41 std::string MultiligandAdsorption::getAdsorbentName ( )

Get the name of the adsorbent.

5.32.4 Member Data Documentation

5.32.4.1 MasterSpeciesList* MultiligandAdsorption::List [protected]

Pointer to the MasterSpeciesList object.

5.32.4.2 int MultiligandAdsorption::num_ligands [protected]

Number of different ligands to consider.

5.32.4.3 std::string MultiligandAdsorption::adsorbent_name [protected]

Name of the adsorbent.

5.32.4.4 int(* MultiligandAdsorption::surface_activity) (const Matrix< double > &logq, Matrix< double > &activity, const void *data) [protected]

Pointer to a surface activity model.

This is a function pointer for a surface activity model. The function must accept the log of the surface concentrations as an argument (logq) and provide the activities for each species (activity). The pointer data is used to pass any additional arguments needed.

Parameters

logq	matrix of the log (base 10) of surface concentrations of all species
activity	matrix of activity coefficients for all surface species (must be overriden)
data	pointer to a data structure needed to calculate activities

5.32.4.5 const void* MultiligandAdsorption::activity_data [protected]

Pointer to the data structure needed for surface activities.

5.32.4.6 int MultiligandAdsorption::act_fun [protected]

Enumeration to represent the choosen surface activity function.

5.32.4.7 Matrix<double> MultiligandAdsorption::activities [protected]

List of the activities calculated by the activity model.

5.32.4.8 double MultiligandAdsorption::specific_area [protected]

Specific surface area of the adsorbent (m^2/kg)

5.32.4.9 double MultiligandAdsorption::total_mass [protected]

Total mass of the adsorbent in the system (kg)

5.32.4.10 double MultiligandAdsorption::total_volume [protected]

Total volume of the system (L)

5.32.4.11 double MultiligandAdsorption::ionic_strength [protected]

Ionic Strength of the system used to adjust equilibria constants (mol/L)

5.32.4.12 double MultiligandAdsorption::charge_density [protected]

Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

5.32.4.13 double MultiligandAdsorption::electric_potential [protected]

Electric surface potential of the adsorbent used to adjust equilibria (V)

5.32.4.14 bool MultiligandAdsorption::IncludeSurfCharge [protected]

True = Includes surface charging corrections, False = Does not consider surface charge.

5.32.4.15 std::vector<AdsorptionReaction> MultiligandAdsorption::ligand_obj [private]

List of the ligands and reactions they have on the surface.

The documentation for this class was generated from the following file:

- shark.h

5.33 MultiligandChemisorption Class Reference

Multi-ligand Chemisorption Reaction Object.

`#include <shark.h>`

Public Member Functions

- MultiligandChemisorption ()
  Default Constructor.
- ~MultiligandChemisorption ()
  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int l, std::vector< int > n)
  Function to call the initialization of objects sequentially.
- void Display_Info ()
  Display the adsorption reaction information.
- void modifyMBEdeltas (MassBalance &mbo)
  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()
  Find and set the adsorbed species indices for each reaction object in each ligand object.
- int setLigandIndices ()
  Find and set the ligand species index.
- int setDeltas ()
  Find and set all the delta values for the site balance.
- void setActivityModelInfo (int(*act)(const Matrix< double > &logq, Matrix< double > &activity, const void *data), const void *act_data)
  Function to set the surface activity model and data pointer.
- void setActivityEnum (int act)
  Set the activity enum to the value of act.
- void setVolumeFactor (int i, double v)
  Set all ith volume factors for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)
  Set all ith area factors for the species list (m^2/mol)
- void setSpecificMolality (int ligand, double a)
  Set the specific molality for the ligand (mol/kg)
- void setAdsorbentName (std::string name)
  Set the name of the adsorbent material or particle.
- void setLigandName (int ligand, std::string name)
  Set the name of the ith ligand.
- void setSpecificArea (double area)
  Set the specific area of the adsorbent.

- void setTotalMass (double mass)

  Set the mass of the adsorbent.
- void setTotalVolume (double volume)

  Set the total volume of the system.
- void setSurfaceChargeBool (bool opt)

  Set the surface charge boolean.
- void setElectricPotential (double a)

  Set the surface electric potential.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- void calculateElecticPotential (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int rxn, int ligand)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_RxnResidual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int rxn, int ligand)

  Calculates the residual for the ith reaction and lth ligand in the system.
- double Eval_SiteBalanceResidual (const Matrix< double > &x, int ligand)

  Calculates the residual for the overall site balance for a given ligand.
- ChemisorptionReaction & getChemisorptionObject (int ligand)

  Return reference to the adsortpion object corresponding to the ligand.
- int getNumberLigands ()

  Get the number of ligands involved with the surface.
- int getActivityEnum ()

  Get the value of the activity enum set by user.
- double getActivity (int i)

  Get the ith activity coefficient from the matrix object.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- double getElectricPotential ()

  Get the value of the electric surface potential (V)
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.

- std::string getLigandName (int ligand)

  Get the name of the ligand object indexed by ligand.
- std::string getAdsorbentName ()

  Get the name of the adsorbent.

Protected Attributes

- MasterSpeciesList * List

  Pointer to the MasterSpeciesList object.
- int num_ligands

  Number of different ligands to consider.
- std::string adsorbent_name

  Name of the adsorbent.
- int(* surface_activity)(const Matrix< double > &logq, Matrix< double > &activity, const void *data)

  Pointer to a surface activity model.
- const void * activity_data

  Pointer to the data structure needed for surface activities.
- int act_fun

  Enumeration to represent the choosen surface activity function.
- Matrix< double > activities

  List of the activities calculated by the activity model.
- double specific_area

  Specific surface area of the adsorbent (m^2/kg)
- double total_mass

  Total mass of the adsorbent in the system (kg)
- double total_volume

  Total volume of the system (L)
- double ionic_strength

  Ionic Strength of the system used to adjust equilibria constants (mol/L)
- double charge_density

  Surface charge density of the adsorbent used to adjust equilbria (C/m^2)
- double electric_potential

  Electric surface potential of the adsorbent used to adjust equilibria (V)
- bool IncludeSurfCharge

  True = Includes surface charging corrections, False = Does not consider surface charge.

Private Attributes

- std::vector< ChemisorptionReaction > ligand_obj

  List of the ligands and reactions they have on the surface.

5.33.1 Detailed Description

Multi-ligand Chemisorption Reaction Object.

C++ Object to handle data and functions associated with forumlating multi-ligand chemisorption reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure. This object is made from a vector of ChemisorptionReaction objects, but differentiate between different ligands that exist on the surface. It is based largely off of the original Multiligand Adsorption object, but will include an explict way to handle the site balances associated with each ligand.

5.33.2 Constructor & Destructor Documentation

5.33.2.1 MultiligandChemisorption::MultiligandChemisorption ( )

Default Constructor.

5.33.2.2 MultiligandChemisorption::~MultiligandChemisorption ( )

Default Destructor.

5.33.3 Member Function Documentation

5.33.3.1 void MultiligandChemisorption::Initialize_Object (MasterSpeciesList & *List*, int *l*, std::vector< int > *n*)

Function to call the initialization of objects sequentially.

Function will initialize each ligand adsorption object.

Parameters

List	reference to MasterSpeciesList object
l	number of ligands on the surface
n	number of reactions for each ligand (ligands must be correctly indexed)

5.33.3.2 void MultiligandChemisorption::Display_Info ( )

Display the adsorption reaction information.

5.33.3.3 void MultiligandChemisorption::modifyMBEdeltas (MassBalance & *mbo*)

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.33.3.4 int MultiligandChemisorption::setAdsorbIndices ( )

Find and set the adsorbed species indices for each reaction object in each ligand object.

This function searches through the Reaction objects in ChemisorptionReaction to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.33.3.5 int MultiligandChemisorption::setLigandIndices ( )

Find and set the ligand species index.

This function searches through the Reaction objects in ChemisorptionReaction to find the ligand species and its index to set that information in the ligand_index structure. Function will return 0 if successful and -1 on a failure.

5.33.3.6 int MultiligandChemisorption::setDeltas ( )

Find and set all the delta values for the site balance.

This function searches through all reaction object instances for the stoicheometry of the ligand in each adsorption reaction. That stoicheometry serves as the basis for determining the site balance. NOTE: the delta for the ligand is set automatically in the setLigandIndex() function, so we can ignore that species. In addition, this function must be called after setLigandIndex() and setAdsorbIndices() are called and after the stoicheometry of each reaction has been determined.

5.33.3.7 void MultiligandChemisorption::setActivityModelInfo (int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void *data) *act*, const void * *act_data*)

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.33.3.8 void MultiligandChemisorption::setActivityEnum (int *act*)

Set the activity enum to the value of act.

5.33.3.9 void MultiligandChemisorption::setVolumeFactor (int *i*, double *v*)

Set all ith volume factors for the species list (cm^3/mol)

5.33.3.10 void MultiligandChemisorption::setAreaFactor (int *i*, double *a*)

Set all ith area factors for the species list (m^2/mol)

5.33.3.11 void MultiligandChemisorption::setSpecificMolality (int *ligand*, double *a*)

Set the specific molality for the ligand (mol/kg)

5.33.3.12 void MultiligandChemisorption::setAdsorbentName (std::string *name*)

Set the name of the adsorbent material or particle.

5.33.3.13 void MultiligandChemisorption::setLigandName (int *ligand*, std::string *name*)

Set the name of the ith ligand.

5.33.3.14 void MultiligandChemisorption::setSpecificArea (double *area*)

Set the specific area of the adsorbent.

5.33.3.15 void MultiligandChemisorption::setTotalMass (double *mass*)

Set the mass of the adsorbent.

5.33.3.16 void MultiligandChemisorption::setTotalVolume (double *volume*)

Set the total volume of the system.

5.33.3.17 void MultiligandChemisorption::setSurfaceChargeBool (bool *opt*)

Set the surface charge boolean.

5.33.3.18 void MultiligandChemisorption::setElectricPotential (double *a*)

Set the surface electric potential.

5.33.3.19 void MultiligandChemisorption::calculateAreaFactors ( )

Calculates the area factors used from the van der Waals volumes.

5.33.3.20 void MultiligandChemisorption::calculateEquilibria (double *T*)

Calculates all equilibrium parameters as a function of temperature.

5.33.3.21 void MultiligandChemisorption::setChargeDensity (const Matrix< double > & *x*)

Calculates and sets the current value of charge density.

5.33.3.22 void MultiligandChemisorption::setIonicStrength (const Matrix< double > & *x*)

Calculates and sets the current value of ionic strength.

5.33.3.23 int MultiligandChemisorption::callSurfaceActivity (const Matrix< double > & *x*)

Calls the activity model and returns an int flag for success or failure.

5.33.3.24 void MultiligandChemisorption::calculateElecticPotential (double *sigma,* double *T,* double *I,* double *rel_epsilon*)

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.33.3.25 double MultiligandChemisorption::calculateEquilibriumCorrection (double *sigma,* double *T,* double *I,* double *rel_epsilon,* int *rxn,* int *ligand*)

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.33.3.26 double MultiligandChemisorption::Eval_RxnResidual (const Matrix< double > & *x,* const Matrix< double > & *gama,* double *T,* double *rel_perm,* int *rxn,* int *ligand*)

Calculates the residual for the ith reaction and lth ligand in the system.

This function will provide a system residual for the ith reaction object involved in the lth ligand's Adsorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.33.3.27 double MultiligandChemisorption::Eval_SiteBalanceResidual (const Matrix< double > & *x,* int *ligand*)

Calculates the residual for the overall site balance for a given ligand.

This function will provide a system residual for the site/ligand balance for the Chemisorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
ligand	index of the ligand of interest of the chemisorption object

5.33.3.28 ChemisorptionReaction& MultiligandChemisorption::getChemisorptionObject (int *ligand*)

Return reference to the adsortpion object corresponding to the ligand.

5.33.3.29 int MultiligandChemisorption::getNumberLigands ( )

Get the number of ligands involved with the surface.

5.33.3.30 int MultiligandChemisorption::getActivityEnum ( )

Get the value of the activity enum set by user.

5.33.3.31 double MultiligandChemisorption::getActivity (int *i*)

Get the ith activity coefficient from the matrix object.

5.33.3.32 double MultiligandChemisorption::getSpecificArea ( )

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.33.3.33 double MultiligandChemisorption::getBulkDensity ( )

Calculate and return bulk density of adsorbent in system (kg/L)

5.33.3.34 double MultiligandChemisorption::getTotalMass ( )

Get the total mass of adsorbent in the system (kg)

5.33.3.35 double MultiligandChemisorption::getTotalVolume ( )

Get the total volume of the system (L)

5.33.3.36 double MultiligandChemisorption::getChargeDensity ( )

Get the value of the surface charge density (C/m^2)

5.33.3.37 double MultiligandChemisorption::getIonicStrength ( )

Get the value of the ionic strength of solution (mol/L)

5.33.3.38 double MultiligandChemisorption::getElectricPotential ( )

Get the value of the electric surface potential (V)

5.33.3.39 bool MultiligandChemisorption::includeSurfaceCharge ( )

Returns true if we are considering surface charging during adsorption.

5.33.3.40 std::string MultiligandChemisorption::getLigandName (int *ligand*)

Get the name of the ligand object indexed by ligand.

5.33.3.41 std::string MultiligandChemisorption::getAdsorbentName ( )

Get the name of the adsorbent.

5.33.4 Member Data Documentation

5.33.4.1 MasterSpeciesList* MultiligandChemisorption::List [protected]

Pointer to the MasterSpeciesList object.

5.33.4.2 int MultiligandChemisorption::num_ligands [protected]

Number of different ligands to consider.

5.33.4.3 std::string MultiligandChemisorption::adsorbent_name [protected]

Name of the adsorbent.

5.33.4.4 int(* MultiligandChemisorption::surface_activity) (const Matrix< double > &logq, Matrix< double > &activity, const void *data) [protected]

Pointer to a surface activity model.

This is a function pointer for a surface activity model. The function must accept the log of the surface concentrations as an argument (logq) and provide the activities for each species (activity). The pointer data is used to pass any additional arguments needed.

Parameters

logq	matrix of the log (base 10) of surface concentrations of all species
activity	matrix of activity coefficients for all surface species (must be overriden)
data	pointer to a data structure needed to calculate activities

5.33.4.5 const void* MultiligandChemisorption::activity_data [protected]

Pointer to the data structure needed for surface activities.

5.33.4.6 int MultiligandChemisorption::act_fun [protected]

Enumeration to represent the choosen surface activity function.

5.33.4.7 Matrix<double> MultiligandChemisorption::activities [protected]

List of the activities calculated by the activity model.

5.33.4.8 double MultiligandChemisorption::specific_area [protected]

Specific surface area of the adsorbent (m^2/kg)

5.33.4.9 double MultiligandChemisorption::total_mass [protected]

Total mass of the adsorbent in the system (kg)

5.33.4.10 double MultiligandChemisorption::total_volume [protected]

Total volume of the system (L)

5.33.4.11 double MultiligandChemisorption::ionic_strength [protected]

Ionic Strength of the system used to adjust equilibria constants (mol/L)

5.33.4.12 double MultiligandChemisorption::charge_density [protected]

Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

5.33.4.13 double MultiligandChemisorption::electric_potential [protected]

Electric surface potential of the adsorbent used to adjust equilibria (V)

5.33.4.14 bool MultiligandChemisorption::IncludeSurfCharge [protected]

True = Includes surface charging corrections, False = Does not consider surface charge.

5.33.4.15 std::vector<ChemisorptionReaction> MultiligandChemisorption::ligand_obj [private]

List of the ligands and reactions they have on the surface.

The documentation for this class was generated from the following file:

- shark.h

5.34 NUM_JAC_DATA Struct Reference

Data structure to form a numerical jacobian matrix with finite differences.

`#include <lark.h>`

Public Attributes

- double eps = sqrt(DBL_EPSILON)
  Perturbation value.
- Matrix< double > Fx
  Vector of function evaluations at x.
- Matrix< double > Fxp
  Vector of function evaluations at x+eps.
- Matrix< double > dxj
  Vector of perturbed x values.

5.34.1 Detailed Description

Data structure to form a numerical jacobian matrix with finite differences.

C-style object to be used in conjunction with the Numerical Jacobian algorithm. This algorithm will used double-precision finite-differences to formulate an approximate Jacobian matrix at the given variable state for the given residual/non-linear function.

5.34.2 Member Data Documentation

5.34.2.1 double NUM_JAC_DATA::eps = sqrt(DBL_EPSILON)

Perturbation value.

5.34.2.2 Matrix<double> NUM_JAC_DATA::Fx

Vector of function evaluations at x.

5.34.2.3 Matrix<double> NUM_JAC_DATA::Fxp

Vector of function evaluations at x+eps.

5.34.2.4 Matrix<double> NUM_JAC_DATA::dxj

Vector of perturbed x values.

The documentation for this struct was generated from the following file:

- lark.h

5.35 OPTRANS_DATA Struct Reference

Data structure for implementation of linear operator transposition.

`#include <lark.h>`

Public Attributes

- Matrix< double > Ii
  The ith column vector of the identity operator.
- Matrix< double > Ai
  The ith column vector of the user's linear operator.

5.35.1 Detailed Description

Data structure for implementation of linear operator transposition.

C-style object used in conjunction with the Operator Transpose algorithm to form an action of A^T*r when A is only available as a linear operator and not a matrix. This is a sub-routine required by GCR and GMRESR to stabilize the outer iterations.

5.35.2 Member Data Documentation

5.35.2.1 Matrix<double> OPTRANS_DATA::Ii

The ith column vector of the identity operator.

5.35.2.2 Matrix<double> OPTRANS_DATA::Ai

The ith column vector of the user's linear operator.

The documentation for this struct was generated from the following file:

- lark.h

5.36 PCG_DATA Struct Reference

Data structure for implementation of the PCG algorithms for symmetric linear systems.

`#include <lark.h>`

Public Attributes

- int maxit = 0
  Maximum allowable iterations - default = min(vector_size,1000)
- int iter = 0
  Actual number of iterations taken.
- double alpha
  Step size for new solution.
- double beta
  Step size for new search direction.
- double tol_rel = 1e-6
  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6
  Absolution tolerance for convergence - default = 1e-6.
- double res
  Absolute residual norm.
- double relres
  Relative residual norm.
- double relres_base
  Initial residual norm.
- double bestres
  Best found residual norm.
- bool Output = true
  True = print messages to console.
- Matrix< double > x
  Current solution to the linear system.
- Matrix< double > bestx
  Best found solution to the linear system.
- Matrix< double > r

Residual vector for the linear system.

- Matrix< double > r_old

  Previous residual vector.

- Matrix< double > z

  Preconditioned residual vector (result of precon function)

- Matrix< double > z_old

  Previous preconditioned residual vector.

- Matrix< double > p

  Search direction.

- Matrix< double > Ap

  Result of matrix-vector multiplication.

5.36.1 Detailed Description

Data structure for implementation of the PCG algorithms for symmetric linear systems.

C-style object used in conjunction with the Preconditioned Conjugate Gradient (PCG) algorithm to iteratively solve a symmetric linear system of equations. This algorithm is optimal if your linear system is symmetric, but will not work at all if your system is asymmetric. For asymmetric systems, use one of the other linear methods.

5.36.2 Member Data Documentation

5.36.2.1 int PCG_DATA::maxit = 0

Maximum allowable iterations - default = min(vector_size,1000)

5.36.2.2 int PCG_DATA::iter = 0

Actual number of iterations taken.

5.36.2.3 double PCG_DATA::alpha

Step size for new solution.

5.36.2.4 double PCG_DATA::beta

Step size for new search direction.

5.36.2.5 double PCG_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.36.2.6 double PCG_DATA::tol_abs = 1e-6

Absolution tolerance for convergence - default = 1e-6.

5.36.2.7 double PCG_DATA::res

Absolute residual norm.

5.36.2.8 double PCG_DATA::relres

Relative residual norm.

5.36.2.9 double PCG_DATA::relres_base

Initial residual norm.

5.36.2.10 double PCG_DATA::bestres

Best found residual norm.

5.36.2.11 bool PCG_DATA::Output = true

True = print messages to console.

5.36.2.12 Matrix<double> PCG_DATA::x

Current solution to the linear system.

5.36.2.13 Matrix<double> PCG_DATA::bestx

Best found solution to the linear system.

5.36.2.14 Matrix<double> PCG_DATA::r

Residual vector for the linear system.

5.36.2.15 Matrix<double> PCG_DATA::r_old

Previous residual vector.

5.36.2.16 Matrix<double> PCG_DATA::z

Preconditioned residual vector (result of precon function)

5.36.2.17 Matrix<double> PCG_DATA::z_old

Previous preconditioned residual vector.

5.36.2.18 Matrix<double> PCG_DATA::p

Search direction.

5.36.2.19 Matrix<double> PCG_DATA::Ap

Result of matrix-vector multiplication.

The documentation for this struct was generated from the following file:

- lark.h

5.37 PeriodicTable Class Reference

Class object that store a digitial copy of all Atom objects.

`#include <eel.h>`

Public Member Functions

- PeriodicTable ()
  Default Constructor - Build Perodic Table.
- ~PeriodicTable ()
  Default Destructor - Destroy the table.
- PeriodicTable (int *n, int N)
  Construct a partial table from a list of atomic numbers.
- PeriodicTable (std::vector< std::string > &Symbol)
  Construct a partial table from a vector of atom symbols.
- PeriodicTable (std::vector< int > &n)
  Construct a partial table from a vector of atomic numbers.
- void DisplayTable ()
  Displays the periodic table via symbols.

Protected Attributes

- std::vector< Atom > Table
  Storage vector for all atoms in the table.

Private Attributes

- int number_elements
  Number of atom objects being stored.

5.37.1 Detailed Description

Class object that store a digitial copy of all Atom objects.

C++ class object to hold digitally registered Atom objects. All registered atoms (Hydrogen to Ununoctium) are stored as in a vector. Currently, this object is unused, but could be modified to be explorable and used as a constant referece for all atoms in the table.

5.37.2 Constructor & Destructor Documentation

5.37.2.1 PeriodicTable::PeriodicTable ( )

Default Constructor - Build Perodic Table.

5.37.2.2 PeriodicTable::~PeriodicTable ( )

Default Destructor - Destroy the table.

5.37.2.3 PeriodicTable::PeriodicTable (int * *n*, int *N*)

Construct a partial table from a list of atomic numbers.

5.37.2.4 PeriodicTable::PeriodicTable (std::vector< std::string > & *Symbol*)

Construct a partial table from a vector of atom symbols.

5.37.2.5 PeriodicTable::PeriodicTable (std::vector< int > & *n*)

Construct a partial table from a vector of atomic numbers.

5.37.3 Member Function Documentation

5.37.3.1 void PeriodicTable::DisplayTable ( )

Displays the periodic table via symbols.

5.37.4 Member Data Documentation

5.37.4.1 std::vector<Atom> PeriodicTable::Table [protected]

Storage vector for all atoms in the table.

5.37.4.2 int PeriodicTable::number_elements [private]

Number of atom objects being stored.

The documentation for this class was generated from the following file:

- eel.h

5.38 PICARD_DATA Struct Reference

Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems.

`#include <lark.h>`

Public Attributes

- int maxit = 0

  Maximum allowable iterations - default = min(3∗vec_size,1000)
- int iter = 0

  Actual number of iterations.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolution tolerance for convergence - default = 1e-6.
- double res

  Residual norm of the iterate.
- double relres

  Relative residual norm of the iterate.
- double relres_base

  Initial residual norm.
- double bestres

  Best found residual norm.
- bool Output = true

  True = print messages to console.
- Matrix< double > x0

  Previous iterate solution vector.
- Matrix< double > bestx

  Best found solution vector.
- Matrix< double > r

  Residual of the non-linear system.

5.38.1 Detailed Description

Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems.

C-style object used in conjunction with the Picard algorithm for solving a non-linear system of equations. This is an extradorinarily simple iterative method by which a weak or loose form of the non-linear system is solved based on an initial guess. User must supplied a residual function for the non-linear system and a function representing the weak solution. Generally, this method is less efficient than Newton methods, but is significantly cheaper.

5.38.2 Member Data Documentation

5.38.2.1 int PICARD_DATA::maxit = 0

Maximum allowable iterations - default = min(3∗vec_size,1000)

5.38.2.2 int PICARD_DATA::iter = 0

Actual number of iterations.

5.38.2.3 double PICARD_DATA::tol_rel = 1e-6

Relative tolerance for convergence - default = 1e-6.

5.38.2.4 double PICARD_DATA::tol_abs = 1e-6

Absolution tolerance for convergence - default = 1e-6.

5.38.2.5 double PICARD_DATA::res

Residual norm of the iterate.

5.38.2.6 double PICARD_DATA::relres

Relative residual norm of the iterate.

5.38.2.7 double PICARD_DATA::relres_base

Initial residual norm.

5.38.2.8 double PICARD_DATA::bestres

Best found residual norm.

5.38.2.9 bool PICARD_DATA::Output = true

True = print messages to console.

5.38.2.10 Matrix<double> PICARD_DATA::x0

Previous iterate solution vector.

5.38.2.11 Matrix<double> PICARD_DATA::bestx

Best found solution vector.

5.38.2.12 Matrix<double> PICARD_DATA::r

Residual of the non-linear system.

The documentation for this struct was generated from the following file:

- lark.h

5.39 PJFNK_DATA Struct Reference

Data structure for the implementation of the PJFNK algorithm for non-linear systems.

`#include <lark.h>`

Public Attributes

- int nl_iter = 0

  Number of non-linear iterations.
- int l_iter = 0

  Number of linear iterations.
- int fun_call = 0

  Actual number of function calls made.
- int nl_maxit = 0

  Maximum allowable non-linear steps.
- int l_maxit = 0

  Maximum allowable linear steps.
- int l_restart = -1

  Number of inner linear steps before restarting (for GMRES, GCR, KMS, etc)
- int linear_solver = -1

  Flag to denote which linear solver to use - default = PJFNK Chooses.
- double nl_tol_abs = 1e-6

  Absolute Convergence tolerance for non-linear system - default = 1e-6.
- double nl_tol_rel = 1e-6

  Relative Convergence tol for the non-linear system - default = 1e-6.
- double lin_tol_rel = 1e-6

  Relative tolerance of the linear solver - default = 1e-6.
- double lin_tol_abs = 1e-6

  Absolute tolerance of the linear solver - default = 1e-6.
- double nl_res

  Absolute redidual norm for the non-linear system.
- double nl_relres

  Relative residual for the non-linear system.
- double nl_res_base

  Initial residual norm for the non-linear system.
- double nl_bestres

  Best found residual norm.
- double eps =sqrt(DBL_EPSILON)

  Value of epsilon used jacvec - default = sqrt(DBL_EPSILON)
- bool NL_Output = true

  True = print PJFNK messages to console.
- bool L_Output = false

  True = print Linear messages to console.
- bool LineSearch = false

  True = use Backtracking Linesearch for global convergence.
- bool Bounce = false

  True = allow Linesearch to go outside local well, False = Strict local convergence.
- bool Converged = false

  True = solution has converged, False = solution has not converged.
- Matrix< double > F

  Stored fuction evaluation at x (also the residual)
- Matrix< double > Fv

  Stored function evaluation at x+eps∗v.
- Matrix< double > v

  Stored vector of x+eps∗v.
- Matrix< double > x

Current solution vector for the non-linear system.

- Matrix< double > bestx

  Best found solution vector to the non-linear system.

- GMRESLP_DATA gmreslp_dat

  Data structure for the GMRESLP method.

- PCG_DATA pcg_dat

  Data structure for the PCG method.

- BiCGSTAB_DATA bicgstab_dat

  Data structure for the BiCGSTAB method.

- CGS_DATA cgs_dat

  Data structure for the CGS method.

- GMRESRP_DATA gmresrp_dat

  Data structure for the GMRESRP method.

- GCR_DATA gcr_dat

  Data structure for the GCR method.

- GMRESR_DATA gmresr_dat

  Data structure for the GMRESR method.

- KMS_DATA kms_dat

  Data structure for the KMS method.

- QR_DATA qr_dat

  Data structure for the QR solve method.

- BACKTRACK_DATA backtrack_dat

  Data structure for the Backtracking Linesearch algorithm.

- const void * res_data

  Data structure pointer for user's residual data.

- const void * precon_data

  Data structure pointer for user's preconditioning data.

- int(* funeval)(const Matrix< double > &x, Matrix< double > &F, const void *res_data)

  Function pointer for the user's function F(x) using there data.

- int(* precon)(const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

  Function pointer for the user's preconditioning function for the linear system.

5.39.1 Detailed Description

Data structure for the implementation of the PJFNK algorithm for non-linear systems.

C-style object to be used in conjunction with the Preconditioned Jacobian-Free Newton-Krylov (PJFNK) method for solving a non-linear system of equations. You can use any of the Krylov methods listed in the krylov_method enum to solve the linear sub-problem. When FOM is specified as the Krylov method, this algorithm becomes equivalent to an exact Newton method. If no Krylov method is specified, then the algorithm will try to pick a method based on the problem size and availability of preconditioning.

5.39.2 Member Data Documentation

5.39.2.1 int PJFNK_DATA::nl_iter = 0

Number of non-linear iterations.

5.39.2.2 int PJFNK_DATA::l_iter = 0

Number of linear iterations.

5.39.2.3 int PJFNK_DATA::fun_call = 0

Actual number of function calls made.

5.39.2.4 int PJFNK_DATA::nl_maxit = 0

Maximum allowable non-linear steps.

5.39.2.5 int PJFNK_DATA::l_maxit = 0

Maximum allowable linear steps.

5.39.2.6 int PJFNK_DATA::l_restart = -1

Number of inner linear steps before restarting (for GMRES, GCR, KMS, etc)

5.39.2.7 int PJFNK_DATA::linear_solver = -1

Flag to denote which linear solver to use - default = PJFNK Chooses.

5.39.2.8 double PJFNK_DATA::nl_tol_abs = 1e-6

Absolute Convergence tolerance for non-linear system - default = 1e-6.

5.39.2.9 double PJFNK_DATA::nl_tol_rel = 1e-6

Relative Convergence tol for the non-linear system - default = 1e-6.

5.39.2.10 double PJFNK_DATA::lin_tol_rel = 1e-6

Relative tolerance of the linear solver - default = 1e-6.

5.39.2.11 double PJFNK_DATA::lin_tol_abs = 1e-6

Absolute tolerance of the linear solver - default = 1e-6.

5.39.2.12 double PJFNK_DATA::nl_res

Absolute redidual norm for the non-linear system.

5.39.2.13 double PJFNK_DATA::nl_relres

Relative residual for the non-linear system.

5.39.2.14 double PJFNK_DATA::nl_res_base

Initial residual norm for the non-linear system.

5.39.2.15 double PJFNK_DATA::nl_bestres

Best found residual norm.

5.39.2.16 double PJFNK_DATA::eps =sqrt(DBL_EPSILON)

Value of epsilon used jacvec - default = sqrt(DBL_EPSILON)

5.39.2.17 bool PJFNK_DATA::NL_Output = true

True = print PJFNK messages to console.

5.39.2.18 bool PJFNK_DATA::L_Output = false

True = print Linear messages to console.

5.39.2.19 bool PJFNK_DATA::LineSearch = false

True = use Backtracking Linesearch for global convergence.

5.39.2.20 bool PJFNK_DATA::Bounce = false

True = allow Linesearch to go outside local well, False = Strict local convergence.

5.39.2.21 bool PJFNK_DATA::Converged = false

True = solution has converged, False = solution has not converged.

5.39.2.22 Matrix<double> PJFNK_DATA::F

Stored fuction evaluation at x (also the residual)

5.39.2.23 Matrix<double> PJFNK_DATA::Fv

Stored function evaluation at x+eps∗v.

5.39.2.24 Matrix<double> PJFNK_DATA::v

Stored vector of x+eps∗v.

5.39.2.25 Matrix<double> PJFNK_DATA::x

Current solution vector for the non-linear system.

5.39.2.26 Matrix<double> PJFNK_DATA::bestx

Best found solution vector to the non-linear system.

5.39.2.27 GMRESLP_DATA PJFNK_DATA::gmreslp_dat

Data structure for the GMRESLP method.

5.39.2.28 PCG_DATA PJFNK_DATA::pcg_dat

Data structure for the PCG method.

5.39.2.29 BiCGSTAB_DATA PJFNK_DATA::bicgstab_dat

Data structure for the BiCGSTAB method.

5.39.2.30 CGS_DATA PJFNK_DATA::cgs_dat

Data structure for the CGS method.

5.39.2.31 GMRESRP_DATA PJFNK_DATA::gmresrp_dat

Data structure for the GMRESRP method.

5.39.2.32 GCR_DATA PJFNK_DATA::gcr_dat

Data structure for the GCR method.

5.39.2.33 GMRESR_DATA PJFNK_DATA::gmresr_dat

Data structure for the GMRESR method.

5.39.2.34 KMS_DATA PJFNK_DATA::kms_dat

Data structure for the KMS method.

5.39.2.35 QR_DATA PJFNK_DATA::qr_dat

Data structure for the QR solve method.

5.39.2.36 BACKTRACK_DATA PJFNK_DATA::backtrack_dat

Data structure for the Backtracking Linesearch algorithm.

5.39.2.37 const void* PJFNK_DATA::res_data

Data structure pointer for user's residual data.

5.39.2.38 const void* PJFNK_DATA::precon_data

Data structure pointer for user's preconditioning data.

5.39.2.39 int(* PJFNK_DATA::funeval) (const Matrix< double > &x, Matrix< double > &F, const void *res_data)

Function pointer for the user's function F(x) using there data.

5.39.2.40 int(* PJFNK_DATA::precon) (const Matrix< double > &r, Matrix< double > &p, const void *precon_data)

Function pointer for the user's preconditioning function for the linear system.

The documentation for this struct was generated from the following file:

- lark.h

5.40 PURE_GAS Struct Reference

Data structure holding all the parameters for each pure gas spieces.

`#include <egret.h>`

Public Attributes

- double molecular_weight
  Given: molecular weights (g/mol)
- double Sutherland_Temp
  Given: Sutherland's Reference Temperature (K)
- double Sutherland_Const
  Given: Sutherland's Constant (K)
- double Sutherland_Viscosity
  Given: Sutherland's Reference Viscosity (g/cm/s)
- double specific_heat
  Given: Specific heat of the gas (J/g/K)
- double molecular_diffusion
  Calculated: molecular diffusivities (cm^ 2/s)
- double dynamic_viscosity
  Calculated: dynamic viscosities (g/cm/s)
- double density
  Calculated: gas densities (g/cm^ 3) {use RE3}.
- double Schmidt
  Calculated: Value of the Schmidt number (-)

5.40.1 Detailed Description

Data structure holding all the parameters for each pure gas spieces.

C-style object that holds the constants and parameters associated with each pure gas species in the overall mixture. This information is used in conjunction with the kinetic theory of gases to produce approximations to many different gas properties needed in simulating gas dynamics, mobility of a gas through porous media, as well as some kinetic adsorption parameters such as diffusivities.

5.40.2 Member Data Documentation

5.40.2.1 double PURE_GAS::molecular_weight

Given: molecular weights (g/mol)

5.40.2.2 double PURE_GAS::Sutherland_Temp

Given: Sutherland's Reference Temperature (K)

5.40.2.3 double PURE_GAS::Sutherland_Const

Given: Sutherland's Constant (K)

5.40.2.4 double PURE_GAS::Sutherland_Viscosity

Given: Sutherland's Reference Viscosity (g/cm/s)

5.40.2.5 double PURE_GAS::specific_heat

Given: Specific heat of the gas (J/g/K)

5.40.2.6 double PURE_GAS::molecular_diffusion

Calculated: molecular diffusivities (cm^2/s)

5.40.2.7 double PURE_GAS::dynamic_viscosity

Calculated: dynamic viscosities (g/cm/s)

5.40.2.8 double PURE_GAS::density

Calculated: gas densities (g/cm^3) {use RE3}.

5.40.2.9 double PURE_GAS::Schmidt

Calculated: Value of the Schmidt number (-)

The documentation for this struct was generated from the following file:

- egret.h

5.41 QR_DATA Struct Reference

Data structure for the implementation of a QR solver given some invertable linear operator.

`#include <lark.h>`

Public Attributes

- Matrix< double > ek

  Unit vector used to extract columns from the linear operator.
- Matrix< double > Ro

  Upper triangular matrix formed from factoring the linear operator.
- Matrix< double > x

  Solution to the linear system.

5.41.1 Detailed Description

Data structure for the implementation of a QR solver given some invertable linear operator.

C-style object to be used in conjuction with a QR solver for invertable linear operators. This method will extract columns from the linear operator and use Householder Reflections to factor the operator into an upper triangular matrix and a unitary reflection matrix. It is generally less efficient to use this method for sparse systems, but is more stable and occassionally more efficient for dense systems.

5.41.2 Member Data Documentation

5.41.2.1 Matrix<double> QR_DATA::ek

Unit vector used to extract columns from the linear operator.

5.41.2.2 Matrix<double> QR_DATA::Ro

Upper triangular matrix formed from factoring the linear operator.

5.41.2.3 Matrix<double> QR_DATA::x

Solution to the linear system.

The documentation for this struct was generated from the following file:

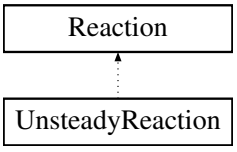
- lark.h

5.42 Reaction Class Reference

Reaction Object.

```
#include <shark.h>
```

Inheritance diagram for Reaction:



Public Member Functions

- Reaction ()

  Default constructor.
- ~Reaction ()

  Default destructor.
- void Initialize_Object (MasterSpeciesList &List)

  Function to initialize the Reaction object from the MasterSpeciesList.
- void Display_Info ()

  Display the reaction information.
- void Set_Stoichiometric (int i, double v)

  Set the ith stoichiometric value.
- void Set_Equilibrium (double logK)

  Set the equilibrium constant in log(K) units.
- void Set_Enthalpy (double H)

  Set the enthalpy of the reaction (J/mol)
- void Set_Entropy (double S)

  Set the entropy of the reaction (J/K/mol)
- void Set_EnthalpyANDEntropy (double H, double S)

  Set both the enthalpy and entropy (J/mol) & (J/K/mol)
- void Set_Energy (double G)

  Set the Gibb's free energy of reaction (J/mol)
- void checkSpeciesEnergies ()

  Function to check MasterList Reference for species energy info.
- void calculateEnergies ()
- void calculateEquilibrium (double T)

  Function to calculate the equilibrium constant based on temperature in K.
- bool haveEquilibrium ()

  Function to return true if equilibrium constant is given or can be calculated.
- double Get_Stoichiometric (int i)

  Fetch the ith stoichiometric value.
- double Get_Equilibrium ()

  Fetch the equilibrium constant (logK)
- double Get_Enthalpy ()

  Fetch the enthalpy of the reaction (J/mol)
- double Get_Entropy ()

  Fetch the entropy of the reaction (J/K/mol)
- double Get_Energy ()

  Fetch the energy of the reaction (J/mol)
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama)

Protected Attributes

- MasterSpeciesList * List

  Pointer to a master species object.
- std::vector< double > Stoichiometric

  Vector of stoichiometric constants corresponding to species list.
- double Equilibrium

  Equilibrium constant for the reaction (logK)
- double enthalpy

Reaction enthalpy (J/mol)

- double entropy

  Reaction entropy (J/K/mol)
- double energy

  Gibb's Free energy of reaction (J/mol)
- bool CanCalcHS

  True if all molecular info is avaiable to calculate dH and dS.
- bool CanCalcG

  True if all molecular info is available to calculate dG.
- bool HaveHS

  True if dH and dS is given, or can be calculated.
- bool HaveG

  True if dG is given, or can be calculated.
- bool HaveEquil

  True as long as Equilibrium is given, or can be calculated.

5.42.1 Detailed Description

Reaction Object.

C++ style object that holds data and functions associated with standard chemical reactions...
i.e., aA + bB <=> cC + dD
These reactions are assumed steady state and are characterized by stoichiometry coefficients and equilibrium/stability constants. Types of reactions that these are valid for would be acid/base reactions, metal-ligand complexation reactions, oxidation-reduction reactions, Henry's Law phase changes, and more. Reactions that this may not be suitable for include mechanisms, adsorption, and precipitation. Those types of reactions would be better handled by more specific objects that inherit from this object.
If all species in the reaction are registered and known species in mola.h AND have known formation energies, then the equilibrium constants for that particular reaction will be calculated based on the species involved in the reaction. However, if using some custom molecule objects, then the reaction equilibrium may not be able to be automatically formed by the routine. In this case, you would need to also supply the equilibrium constant for the particular reaction.

5.42.2 Constructor & Destructor Documentation

5.42.2.1 Reaction::Reaction ( )

Default constructor.

5.42.2.2 Reaction::~Reaction ( )

Default destructor.

5.42.3 Member Function Documentation

5.42.3.1 void Reaction::Initialize_Object (MasterSpeciesList & *List*)

Function to initialize the Reaction object from the MasterSpeciesList.

5.42.3.2 void Reaction::Display_Info ( )

Display the reaction information.

5.42.3.3 void Reaction::Set_Stoichiometric (int *i*, double *v*)

Set the ith stoichiometric value.

This function will set the stoichiometric constant of the ith species in the master list to the given value of v. All values of v are set to zero unless overriden by this function.

Parameters

i	index of the species in the MasterSpeciesList
v	value of the stoichiometric constant for that species in the reaction

5.42.3.4 void Reaction::Set_Equilibrium (double *logK*)

Set the equilibrium constant in log(K) units.

5.42.3.5 void Reaction::Set_Enthalpy (double *H*)

Set the enthalpy of the reaction (J/mol)

5.42.3.6 void Reaction::Set_Entropy (double *S*)

Set the entropy of the reaction (J/K/mol)

5.42.3.7 void Reaction::Set_EnthalpyANDEntropy (double *H,* double *S*)

Set both the enthalpy and entropy (J/mol) & (J/K/mol)

5.42.3.8 void Reaction::Set_Energy (double *G*)

Set the Gibb's free energy of reaction (J/mol)

5.42.3.9 void Reaction::checkSpeciesEnergies ( )

Function to check MasterList Reference for species energy info.

This function will go through the stoichiometry of this reaction and check the molecules in the MasterSpeciesList that correspond to the species present in this reaction for the existance of their formation energies. Based on the states of those energies, it will note internally whether or not it can determine the equilibrium constants based soley on individual species information. If it cannot, then the user must provide either the reaction energies to form the equilibrium constant or the equilibrium constant itself. Function to calculate and set the energy of the reaction

5.42.3.10 void Reaction::calculateEnergies ( )

If the energies of the reaction can be determined from the individual species in the reaction, then this function uses that information. Otherwise, it sets the energies equal to the constants given to the object by the user.

5.42.3.11 void Reaction::calculateEquilibrium (double *T*)

Function to calculate the equilibrium constant based on temperature in K.

5.42.3.12 bool Reaction::haveEquilibrium ( )

Function to return true if equilibrium constant is given or can be calculated.

5.42.3.13 double Reaction::Get_Stoichiometric (int *i*)

Fetch the ith stoichiometric value.

5.42.3.14 double Reaction::Get_Equilibrium ( )

Fetch the equilibrium constant (logK)

5.42.3.15 double Reaction::Get_Enthalpy ( )

Fetch the enthalpy of the reaction (J/mol)

5.42.3.16 double Reaction::Get_Entropy ( )

Fetch the entropy of the reaction (J/K/mol)

5.42.3.17 double Reaction::Get_Energy ( )

Fetch the energy of the reaction (J/mol)

Evaluate a residual for the reaction given variable x=log(C) and activity coefficients gama

5.42.3.18 double Reaction::Eval_Residual (const Matrix< double > & *x*, const Matrix< double > & *gama*)

This function will calculate the reaction residual from this object's stoichiometry, equilibrium constant, log(C) concentrations, and activity coefficients.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step

5.42.4 Member Data Documentation

5.42.4.1 MasterSpeciesList* Reaction::List [protected]

Pointer to a master species object.

5.42.4.2 std::vector<double> Reaction::Stoichiometric [protected]

Vector of stoichiometric constants corresponding to species list.

5.42.4.3 double Reaction::Equilibrium [protected]

Equilibrium constant for the reaction (logK)

5.42.4.4 double Reaction::enthalpy [protected]

Reaction enthalpy (J/mol)

5.42.4.5 double Reaction::entropy [protected]

Reaction entropy (J/K/mol)

5.42.4.6 double Reaction::energy [protected]

Gibb's Free energy of reaction (J/mol)

5.42.4.7 bool Reaction::CanCalcHS [protected]

True if all molecular info is avaiable to calculate dH and dS.

5.42.4.8 bool Reaction::CanCalcG [protected]

True if all molecular info is available to calculate dG.

5.42.4.9 bool Reaction::HaveHS [protected]

True if dH and dS is given, or can be calculated.

5.42.4.10 bool Reaction::HaveG [protected]

True if dG is given, or can be calculated.

5.42.4.11 bool Reaction::HaveEquil [protected]

True as long as Equilibrium is given, or can be calculated.

The documentation for this class was generated from the following file:

- shark.h

5.43 SCOPSOWL_DATA Struct Reference

Primary data structure for SCOPSOWL simulations.

`#include <scopsowl.h>`

Public Attributes

- unsigned long int total_steps

  Running total of all calculation steps.
- int coord_macro

  Coordinate system for large pellet.
- int coord_micro

  Coordinate system for small crystal (if any)
- int level = 2

  Level of coupling between the different scales (default = 2)
- double sim_time

  Stopping time for the simulation (hrs)
- double t_old

  Old time of the simulations (hrs)
- double t

  Current time of the simulations (hrs)
- double t_counter = 0.0

  Counter for the time output.
- double t_print

  Print output at every t_print time (hrs)
- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- bool SurfDiff = true

  True = includes SKUA simulation if Heterogeneous; False = only uses MAGPIE.
- bool Heterogeneous = true

  True = pellet is made of binder and crystals, False = all one phase.
- double gas_velocity

  Superficial Gas Velocity arount pellet (cm/s)
- double total_pressure

  Gas phase total pressure (kPa)
- double gas_temperature

  Gas phase temperature (K)
- double pellet_radius

  Nominal radius of the pellet - macroscale domain (cm)
- double crystal_radius

  Nominal radius of the crystal - microscale domain (um)
- double char_macro

  Characteristic size for macro scale (cm or cm^2) - only if pellet is not spherical.
- double char_micro

  Characteristic size for micro scale (um or um^2) - only if crystal is not spherical.
- double binder_fraction

  Volume of binder per total volume of pellet (-)
- double binder_porosity

  Volume of pores per volume of binder (-)
- double binder_poresize

  Nominal radius of the binder pores (cm)
- double pellet_density

  Mass of the pellet per volume of pellet (kg/L)
- bool DirichletBC = false

True = Dirichlet BC; False = Neumann BC.

- bool NonLinear = true

  True = Non-linear solver; False = Linear solver.

- std::vector< double > y

  Outside mole fractions of each component (-)

- std::vector< double > tempy

  Temporary place holder for gas mole fractions in other locations (-)

- FILE * OutputFile

  Output file pointer to the output file for postprocesses.

- double(* eval_ads)(int i, int l, const void *user_data)

  Function pointer for evaluating adsorption (mol/kg)

- double(* eval_retard)(int i, int l, const void *user_data)

  Function pointer for evaluating retardation (-)

- double(* eval_diff)(int i, int l, const void *user_data)

  Function pointer for evaluating pore diffusion (cm^2/hr)

- double(* eval_surfDiff)(int i, int l, const void *user_data)

  Function pointer for evaluating surface diffusion (um^2/hr)

- double(* eval_kf)(int i, const void *user_data)

  Function pointer for evaluating film mass transfer (cm/hr)

- const void * user_data

  Data structure for users info to calculate parameters.

- MIXED_GAS * gas_dat

  Pointer to the MIXED_GAS data structure (may or may not be used)

- MAGPIE_DATA magpie_dat

  Data structure for a magpie problem (to be used if not using skua)

- std::vector< FINCH_DATA > finch_dat

  Data structure for pore adsorption kinetics for all species (u in mol/L)

- std::vector< SCOPSOWL_PARAM_DATA > param_dat

  Data structure for parameter info for all species.

- std::vector< SKUA_DATA > skua_dat

  Data structure holding a skua object for all nodes (each skua has an object for each species)

5.43.1 Detailed Description

Primary data structure for SCOPSOWL simulations.

C-style object holding necessary information to run a SCOPSOWL simulation. SCOPSOWL is a multi-scale problem involving PDE solution for the macro-scale adsorbent pellet and the micro-scale adsorbent crystals. As such, each SCOPSOWL simulation involves multiple SKUA simulations at the nodes in the macro-scale domain. Alternatively, if the user wishes to specify that the adsorbent is homogeneous, then you can run SCOPSOWL as a single-scale problem. Additionally, you can simplfy the model by assuming that the micro-scale diffusion is very fast, and therefore replace each SKUA simulation with a simpler MAGPIE evaluation. Details on running SCOPSOWL with the various options will be discussed in the SCOPSOWL_SCENARIOS function.

5.43.2 Member Data Documentation

5.43.2.1 unsigned long int SCOPSOWL_DATA::total_steps

Running total of all calculation steps.

5.43.2.2 int SCOPSOWL_DATA::coord_macro

Coordinate system for large pellet.

5.43.2.3 int SCOPSOWL_DATA::coord_micro

Coordinate system for small crystal (if any)

5.43.2.4 int SCOPSOWL_DATA::level = 2

Level of coupling between the different scales (default = 2)

5.43.2.5 double SCOPSOWL_DATA::sim_time

Stopping time for the simulation (hrs)

5.43.2.6 double SCOPSOWL_DATA::t_old

Old time of the simulations (hrs)

5.43.2.7 double SCOPSOWL_DATA::t

Current time of the simulations (hrs)

5.43.2.8 double SCOPSOWL_DATA::t_counter = 0.0

Counter for the time output.

5.43.2.9 double SCOPSOWL_DATA::t_print

Print output at every t_print time (hrs)

5.43.2.10 bool SCOPSOWL_DATA::Print2File = true

True = results to .txt; False = no printing.

5.43.2.11 bool SCOPSOWL_DATA::Print2Console = true

True = results to console; False = no printing.

5.43.2.12 bool SCOPSOWL_DATA::SurfDiff = true

True = includes SKUA simulation if Heterogeneous; False = only uses MAGPIE.

5.43.2.13 bool SCOPSOWL_DATA::Heterogeneous = true

True = pellet is made of binder and crystals, False = all one phase.

5.43.2.14 double SCOPSOWL_DATA::gas_velocity

Superficial Gas Velocity arount pellet (cm/s)

5.43.2.15 double SCOPSOWL_DATA::total_pressure

Gas phase total pressure (kPa)

5.43.2.16 double SCOPSOWL_DATA::gas_temperature

Gas phase temperature (K)

5.43.2.17 double SCOPSOWL_DATA::pellet_radius

Nominal radius of the pellet - macroscale domain (cm)

5.43.2.18 double SCOPSOWL_DATA::crystal_radius

Nominal radius of the crystal - microscale domain (um)

5.43.2.19 double SCOPSOWL_DATA::char_macro

Characteristic size for macro scale (cm or cm^2) - only if pellet is not spherical.

5.43.2.20 double SCOPSOWL_DATA::char_micro

Characteristic size for micro scale (um or um^2) - only if crystal is not spherical.

5.43.2.21 double SCOPSOWL_DATA::binder_fraction

Volume of binder per total volume of pellet (-)

5.43.2.22 double SCOPSOWL_DATA::binder_porosity

Volume of pores per volume of binder (-)

5.43.2.23 double SCOPSOWL_DATA::binder_poresize

Nominal radius of the binder pores (cm)

5.43.2.24 double SCOPSOWL_DATA::pellet_density

Mass of the pellet per volume of pellet (kg/L)

5.43.2.25 bool SCOPSOWL_DATA::DirichletBC = false

True = Dirichlet BC; False = Neumann BC.

5.43.2.26 bool SCOPSOWL_DATA::NonLinear = true

True = Non-linear solver; False = Linear solver.

5.43.2.27 std::vector<double> SCOPSOWL_DATA::y

Outside mole fractions of each component (-)

5.43.2.28 std::vector<double> SCOPSOWL_DATA::tempy

Temporary place holder for gas mole fractions in other locations (-)

5.43.2.29 FILE* SCOPSOWL_DATA::OutputFile

Output file pointer to the output file for postprocesses.

5.43.2.30 double(* SCOPSOWL_DATA::eval_ads) (int i, int l, const void *user_data)

Function pointer for evaluating adsorption (mol/kg)

5.43.2.31 double(* SCOPSOWL_DATA::eval_retard) (int i, int l, const void *user_data)

Function pointer for evaluating retardation (-)

5.43.2.32 double(* SCOPSOWL_DATA::eval_diff) (int i, int l, const void *user_data)

Function pointer for evaluating pore diffusion (cm^2/hr)

5.43.2.33 double(* SCOPSOWL_DATA::eval_surfDiff) (int i, int l, const void *user_data)

Function pointer for evaluating surface diffusion (um^2/hr)

5.43.2.34 double(* SCOPSOWL_DATA::eval_kf) (int i, const void *user_data)

Function pointer for evaluating film mass transfer (cm/hr)

5.43.2.35 const void* SCOPSOWL_DATA::user_data

Data structure for users info to calculate parameters.

5.43.2.36 MIXED_GAS* SCOPSOWL_DATA::gas_dat

Pointer to the MIXED_GAS data structure (may or may not be used)

5.43.2.37 MAGPIE_DATA SCOPSOWL_DATA::magpie_dat

Data structure for a magpie problem (to be used if not using skua)

5.43.2.38 std::vector<FINCH_DATA> SCOPSOWL_DATA::finch_dat

Data structure for pore adsorption kinetics for all species (u in mol/L)

5.43.2.39 std::vector<SCOPSOWL_PARAM_DATA> SCOPSOWL_DATA::param_dat

Data structure for parameter info for all species.

5.43.2.40 std::vector<SKUA_DATA> SCOPSOWL_DATA::skua_dat

Data structure holding a skua object for all nodes (each skua has an object for each species)

The documentation for this struct was generated from the following file:

- scopsowl.h

5.44 SCOPSOWL_OPT_DATA Struct Reference

Data structure for the SCOPSOWL optmization routine.

`#include <scopsowl_opt.h>`

Public Attributes

- int num_curves
  Number of adsorption curves to analyze.
- int evaluation
  Number of times the eval function has been called for a single curve.
- unsigned long int total_eval
  Total number of evaluations needed for completion.
- int current_points
  Number of points in the current curve.
- int num_params = 1
  Number of adjustable parameters for the current curve (currently only supports 1)
- int diffusion_type
  Flag to identify type of diffusion function to use.
- int adsorb_index
  Component index for adsorbable species.
- int max_guess_iter = 20
  Maximum allowed guess iterations (default = 20)
- bool Optimize
  True = run optimization, False = run a comparison.
- bool Rough
  True = use only a rough estimate, False = run full optimization.
- double current_temp
  Temperature for current curve.
- double current_press
  Partial pressure for current curve.
- double current_equil

Equilibrium data point for the current curve.

- double simulation_equil

  Equilibrium simulation point for the current curve.
- double max_bias

  Positive maximum bias plausible for fitting.
- double min_bias

  Negative minimum bias plausible for fitting.
- double e_norm

  Euclidean norm of current fit.
- double f_bias

  Function bias of current fit.
- double e_norm_old

  Euclidean norm of the previous fit.
- double f_bias_old

  Function bias of the previous fit.
- double param_guess

  Parameter guess for the surface/crystal diffusivity.
- double param_guess_old

  Parameter guess for the previous curve.
- double rel_tol_norm = 0.01

  Tolerance for convergence of the guess norm.
- double abs_tol_bias = 1.0

  Tolerance for convergence of the guess bias.
- std::vector< double > y_base

  Gas phase mole fractions in absense of adsorbing species.
- std::vector< double > q_data

  Amount adsorbed at a particular point in current curve.
- std::vector< double > q_sim

  Amount adsorbed based on the simulation.
- std::vector< double > t

  Time points in the current curve.
- FILE * ParamFile

  Output file for parameter results.
- FILE * CompareFile

  Output file for comparison of results.
- SCOPSOWL_DATA owl_dat

  Data structure for the SCOPSOWL simulation.

5.44.1 Detailed Description

Data structure for the SCOPSOWL optmization routine.

C-style object holding information about the optimization routine as well as the standard SCOPSOwl_DATA structure for SCOPSOWL simulations.

5.44.2 Member Data Documentation

5.44.2.1 int SCOPSOWL_OPT_DATA::num_curves

Number of adsorption curves to analyze.

5.44.2.2 int SCOPSOWL_OPT_DATA::evaluation

Number of times the eval function has been called for a single curve.

5.44.2.3 unsigned long int SCOPSOWL_OPT_DATA::total_eval

Total number of evaluations needed for completion.

5.44.2.4 int SCOPSOWL_OPT_DATA::current_points

Number of points in the current curve.

5.44.2.5 int SCOPSOWL_OPT_DATA::num_params = 1

Number of adjustable parameters for the current curve (currently only supports 1)

5.44.2.6 int SCOPSOWL_OPT_DATA::diffusion_type

Flag to identify type of diffusion function to use.

5.44.2.7 int SCOPSOWL_OPT_DATA::adsorb_index

Component index for adsorbable species.

5.44.2.8 int SCOPSOWL_OPT_DATA::max_guess_iter = 20

Maximum allowed guess iterations (default = 20)

5.44.2.9 bool SCOPSOWL_OPT_DATA::Optimize

True = run optimization, False = run a comparison.

5.44.2.10 bool SCOPSOWL_OPT_DATA::Rough

True = use only a rough estimate, False = run full optimization.

5.44.2.11 double SCOPSOWL_OPT_DATA::current_temp

Temperature for current curve.

5.44.2.12 double SCOPSOWL_OPT_DATA::current_press

Partial pressure for current curve.

5.44.2.13 double SCOPSOWL_OPT_DATA::current_equil

Equilibrium data point for the current curve.

5.44.2.14 double SCOPSOWL_OPT_DATA::simulation_equil

Equilibrium simulation point for the current curve.

5.44.2.15 double SCOPSOWL_OPT_DATA::max_bias

Positive maximum bias plausible for fitting.

5.44.2.16 double SCOPSOWL_OPT_DATA::min_bias

Negative minimum bias plausible for fitting.

5.44.2.17 double SCOPSOWL_OPT_DATA::e_norm

Euclidean norm of current fit.

5.44.2.18 double SCOPSOWL_OPT_DATA::f_bias

Function bias of current fit.

5.44.2.19 double SCOPSOWL_OPT_DATA::e_norm_old

Euclidean norm of the previous fit.

5.44.2.20 double SCOPSOWL_OPT_DATA::f_bias_old

Function bias of the previous fit.

5.44.2.21 double SCOPSOWL_OPT_DATA::param_guess

Parameter guess for the surface/crystal diffusivity.

5.44.2.22 double SCOPSOWL_OPT_DATA::param_guess_old

Parameter guess for the previous curve.

5.44.2.23 double SCOPSOWL_OPT_DATA::rel_tol_norm = 0.01

Tolerance for convergence of the guess norm.

5.44.2.24 double SCOPSOWL_OPT_DATA::abs_tol_bias = 1.0

Tolerance for convergence of the guess bias.

5.44.2.25 std::vector<double> SCOPSOWL_OPT_DATA::y_base

Gas phase mole fractions in absense of adsorbing species.

5.44.2.26 std::vector<double> SCOPSOWL_OPT_DATA::q_data

Amount adsorbed at a particular point in current curve.

5.44.2.27 std::vector<double> SCOPSOWL_OPT_DATA::q_sim

Amount adsorbed based on the simulation.

5.44.2.28 std::vector<double> SCOPSOWL_OPT_DATA::t

Time points in the current curve.

5.44.2.29 FILE* SCOPSOWL_OPT_DATA::ParamFile

Output file for parameter results.

5.44.2.30 FILE* SCOPSOWL_OPT_DATA::CompareFile

Output file for comparison of results.

5.44.2.31 SCOPSOWL_DATA SCOPSOWL_OPT_DATA::owl_dat

Data structure for the SCOPSOWL simulation.

The documentation for this struct was generated from the following file:

- scopsowl_opt.h

5.45 SCOPSOWL_PARAM_DATA Struct Reference

Data structure for the species' parameters in SCOPSOWL.

```
#include <scopsowl.h>
```

Public Attributes

- Matrix< double > qAvg
  Average adsorbed amount for a species at each node (mol/kg)
- Matrix< double > qAvg_old
  Old Average adsorbed amount for a species at each node (mol/kg)
- Matrix< double > Qst
  Heat of adsorption for all nodes (J/mol)
- Matrix< double > Qst_old
  Old Heat of adsorption for all nodes (J/mol)
- Matrix< double > dq_dc
  Storage vector for current adsorption slope/strength (dq/dc) (L/kg)
- double xIC
  Initial conditions for adsorbed molefractions.
- double qIntegralAvg

Integral average of adsorption over the entire pellet (mol/kg)

- double qIntegralAvg_old

  Old Integral average of adsorption over the entire pellet (mol/kg)
- double QstAvg

  Integral average heat of adsorption (J/mol)
- double QstAvg_old

  Old integral average heat of adsorption (J/mol)
- double qo

  Boundary value of adsorption if using Dirichlet BCs (mol/kg)
- double Qsto

  Boundary value of adsorption heat if using Dirichlet BCs (J/mol)
- double dq_dco

  Boundary value of adsorption slope for Dirichelt BCs (L/kg)
- double pore_diffusion

  Value for constant pore diffusion (cm^2/hr)
- double film_transfer

  Value for constant film mass transfer (cm/hr)
- double activation_energy

  Activation energy for surface diffusion (J/mol)
- double ref_diffusion

  Reference state surface diffusivity (um^2/hr)
- double ref_temperature

  Reference temperature for empirical adjustments (K)
- double affinity

  Affinity parameter used in empirical adjustments (-)
- double ref_pressure
- bool Adsorbable

  True = species can adsorb; False = species cannot adsorb.
- std::string speciesName

  String to hold the name of each species.

5.45.1 Detailed Description

Data structure for the species' parameters in SCOPSOWL.

C-style object that holds information on all species for a particular SCOPSOWL simulation. Initial conditions, kinetic parameters, and interim matrix objects are stored here for use in various SCOSPSOWL functions.

5.45.2 Member Data Documentation

5.45.2.1 Matrix<double> SCOPSOWL_PARAM_DATA::qAvg

Average adsorbed amount for a species at each node (mol/kg)

5.45.2.2 Matrix<double> SCOPSOWL_PARAM_DATA::qAvg_old

Old Average adsorbed amount for a species at each node (mol/kg)

5.45.2.3 Matrix<double> SCOPSOWL_PARAM_DATA::Qst

Heat of adsorption for all nodes (J/mol)

5.45.2.4 Matrix<double> SCOPSOWL_PARAM_DATA::Qst_old

Old Heat of adsorption for all nodes (J/mol)

5.45.2.5 Matrix<double> SCOPSOWL_PARAM_DATA::dq_dc

Storage vector for current adsorption slope/strength (dq/dc) (L/kg)

5.45.2.6 double SCOPSOWL_PARAM_DATA::xIC

Initial conditions for adsorbed molefractions.

5.45.2.7 double SCOPSOWL_PARAM_DATA::qIntegralAvg

Integral average of adsorption over the entire pellet (mol/kg)

5.45.2.8 double SCOPSOWL_PARAM_DATA::qIntegralAvg_old

Old Integral average of adsorption over the entire pellet (mol/kg)

5.45.2.9 double SCOPSOWL_PARAM_DATA::QstAvg

Integral average heat of adsorption (J/mol)

5.45.2.10 double SCOPSOWL_PARAM_DATA::QstAvg_old

Old integral average heat of adsorption (J/mol)

5.45.2.11 double SCOPSOWL_PARAM_DATA::qo

Boundary value of adsorption if using Dirichlet BCs (mol/kg)

5.45.2.12 double SCOPSOWL_PARAM_DATA::Qsto

Boundary value of adsorption heat if using Dirichlet BCs (J/mol)

5.45.2.13 double SCOPSOWL_PARAM_DATA::dq_dco

Boundary value of adsorption slope for Dirichelt BCs (L/kg)

5.45.2.14 double SCOPSOWL_PARAM_DATA::pore_diffusion

Value for constant pore diffusion (cm^2/hr)

5.45.2.15 double SCOPSOWL_PARAM_DATA::film_transfer

Value for constant film mass transfer (cm/hr)

5.45.2.16 double SCOPSOWL_PARAM_DATA::activation_energy

Activation energy for surface diffusion (J/mol)

5.45.2.17 double SCOPSOWL_PARAM_DATA::ref_diffusion

Reference state surface diffusivity (um^2/hr)

5.45.2.18 double SCOPSOWL_PARAM_DATA::ref_temperature

Reference temperature for empirical adjustments (K)

5.45.2.19 double SCOPSOWL_PARAM_DATA::affinity

Affinity parameter used in empirical adjustments (-)

5.45.2.20 double SCOPSOWL_PARAM_DATA::ref_pressure

5.45.2.21 bool SCOPSOWL_PARAM_DATA::Adsorbable

True = species can adsorb; False = species cannot adsorb.

5.45.2.22 std::string SCOPSOWL_PARAM_DATA::speciesName

String to hold the name of each species.

The documentation for this struct was generated from the following file:

- scopsowl.h

5.46 SHARK_DATA Struct Reference

Data structure for SHARK simulations.

`#include <shark.h>`

Public Attributes

- MasterSpeciesList MasterList

  Master List of species object.
- std::vector< Reaction > ReactionList

  Equilibrium reaction objects.
- std::vector< MassBalance > MassBalanceList

  Mass balance objects.
- std::vector< UnsteadyReaction > UnsteadyList

  Unsteady Reaction objects.
- std::vector< AdsorptionReaction > AdsorptionList

  Equilibrium Adsorption Reaction Objects.
- std::vector< UnsteadyAdsorption > UnsteadyAdsList

  Unsteady Adsorption Reaction Objects.
- std::vector< MultiligandAdsorption > MultiAdsList

  Multiligand Adsorptioin Objects.
- std::vector< ChemisorptionReaction > ChemisorptionList

  Chemisorption Reaction objects.
- std::vector< MultiligandChemisorption > MultiChemList

  Multiligand Chemisorption Reaction Objects.
- std::vector< double(*)(const Matrix< double > &x, SHARK_DATA *shark_dat, const void *data) > OtherList

  Array of Other Residual functions to be defined by user.
- int numvar

  Total number of functions and species.
- int num_ssr

  Number of steady-state reactions.
- int num_mbe

  Number of mass balance equations.
- int num_usr = 0

  Number of unsteady-state reactions.
- int num_ssao = 0

  Number of steady-state adsorption objects.
- int num_usao = 0

  Number of unsteady adsorption objects.
- int num_multi_ssao = 0

  Number of multiligand steady-state adsorption objects.
- int num_sschem = 0

  Number of steady-state chemisorption objects.
- int num_multi_sschem = 0

  Number of multiligand steady-state chemisorption objects.
- std::vector< int > num_ssar

  List of the numbers of reactions in each adsorption object.
- std::vector< int > num_usar

  List of the numbers of reactions in each unsteady adsorption object.
- std::vector< int > num_sschem_rxns

  List of the numbers of reactions in each steady-state chemisorption object.
- std::vector< std::vector< int > > num_multi_ssar

  List of all multiligand objects -> List of ligands and rxns of that ligand.
- std::vector< std::vector< int > > num_multichem_rxns

  List of all multiligand chemisorption objects -> List of num rxns for that ligand.
- std::vector< std::string > ss_ads_names

List of the steady-state adsorbent object names.

- std::vector< std::string > us_ads_names

  List of the unsteady adsorption object names.

- std::vector< std::string > ss_chem_names

  List of the steady-state chemisorption object names.

- std::vector< std::vector< std::string > > ssmulti_names

  List of the names of the ligands in each multiligand object.

- std::vector< std::vector< std::string > > ssmultichem_names

  List of the names of the ligands in each multiligand chemisorption object.

- int num_other = 0

  Number of other functions to be used (default is always 0)

- int act_fun = IDEAL

  Flag denoting the activity function to use (default is IDEAL)

- int reactor_type = BATCH

  Flag denoting the type of reactor considered for the system (default is BATCH)

- int totalsteps = 0

  Total number of iterations.

- int totalcalls = 0

  Total number of residual function calls.

- int timesteps = 0

  Number of time steps taken to complete simulation.

- int pH_index = -1

  Contains the index of the pH variable (set internally)

- int pOH_index = -1

  Contains the index of the pOH variable (set internally)

- double simulationtime = 0.0

  Time to simulate unsteady reactions for (default = 0.0 hrs)

- double dt = 0.1

  Time step size (hrs)

- double dt_min = sqrt(DBL_EPSILON)

  Minimum allowable step size.

- double dt_max = 744.0

  Maximum allowable step size (∼1 month in time)

- double t_out = 0.0

  Time increment by which file output is made (default = print all time steps)

- double t_count = 0.0

  Running count of time increments.

- double time = 0.0

  Current value of time (starts from t = 0.0 hrs)

- double time_old = 0.0

  Previous value of time (start from t = 0.0 hrs)

- double pH = 7.0

  Value of pH if needed (default = 7)

- double pH_step = 0.5

  Value by which to increment pH when doing a speciation curve (default = 0.5)

- double start_temp = 277.15

  Value of the starting temperature used for Temperature Curves (default = 277.15 K)

- double end_temp = 323.15

  Value of the ending temperature used for Temperature Curves (default = 323.15 K)

- double temp_step = 10.0

  Size of the step changes to use for Temperature Curves (default = 10.0 K);.

- double volume = 1.0

  Volume of the domain in liters (default = 1 L)
- double flow_rate = 1.0

  Flow rate in the reactor in L/hr (default = 1 L/hr)
- double xsec_area = 1.0

  Cross sectional area of the reactor in m^2 (default = 1 m^2)
- double Norm = 0.0

  Current value of euclidean norm in solution.
- double dielectric_const = 78.325

  Dielectric constant used in many activity models (default: water = 78.325 (1/K))
- double relative_permittivity = 80.1

  Relative permittivity of the medium (default: water = 80.1 (-))
- double temperature = 298.15

  Solution temperature (default = 25 oC or 298.15 K)
- double ionic_strength = 0.0

  Solution ionic strength in Molar (calculated internally)
- bool steadystate = true

  True = solve steady problem; False = solve transient problem.
- bool ZeroInitialSolids = false

  True = no solids or adsorption initially in the reactor.
- bool TimeAdaptivity = false

  True = solve using variable time step.
- bool const_pH = false

  True = set pH to a constant; False = solve for pH.
- bool SpeciationCurve = false

  True = runs a series of constant pH steady-state problems to produce curves.
- bool TemperatureCurve = false

  True = runs a series of constant temperature steady-state problmes to produce curves.
- bool Console_Output = true

  True = display output to console.
- bool File_Output = false

  True = write output to a file.
- bool Contains_pH = false

  True = system contains pH as a variable (set internally)
- bool Contains_pOH = false

  True = system contains pOH as a variable (set internally)
- bool Converged = false

  True = system converged within tolerance.
- bool LocalMin = true

  True = allow the system to settle for a local minimum if tolerance not reached.
- Matrix< double > X_old

  Solution vector for old time step - log(C)
- Matrix< double > X_new

  Solution vector for current time step - log(C)
- Matrix< double > Conc_old

  Concentration vector for old time step - 10^x.
- Matrix< double > Conc_new

  Concentration vector for current time step - 10^x.
- Matrix< double > activity_new

  Activity matrix for current time step.
- Matrix< double > activity_old

Activity matrix from prior time step.

- int(∗ EvalActivity)(const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Function pointer to evaluate activity coefficients.
- int(∗ Residual)(const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Function pointer to evaluate all residuals in the system.
- int(∗ lin_precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗data)

  Function pointer to form a linear preconditioning operation for the Jacobian.
- PJFNK_DATA Newton_data

  Data structure for the Newton-Krylov solver (see lark.h)
- const void ∗ activity_data

  User defined data structure for an activity model.
- const void ∗ residual_data

  User defined data structure for the residual function.
- const void ∗ precon_data

  User defined data structure for preconditioning.
- const void ∗ other_data

  User define data structure used for user defined residuals.
- FILE ∗ OutputFile

  Output File pointer.
- yaml_cpp_class yaml_object

  yaml object to read and access digitized yaml documents (see yaml_wrapper.h)

5.46.1 Detailed Description

Data structure for SHARK simulations.

C-style object holding data and function pointers associated with solving aqueous speciation and reaction kinetics. This object couples all other objects available in shark.h in order to provide residual calculations for each individual function that makes up the overall system model. Those residuals are brought together inside the residual function and fed into the lark.h PJFNK solver routine. That solver then attempts to find a solution to all non-linear variables simultaneously. Any function or data pointers in this structure can be overriden to change how you interface with and solve the problem. Users may also provide a set of custom residual functions through the "OtherList" vector object. Those residual function must all have the same format.

5.46.2 Member Data Documentation

5.46.2.1 MasterSpeciesList SHARK_DATA::MasterList

Master List of species object.

5.46.2.2 std::vector<Reaction> SHARK_DATA::ReactionList

Equilibrium reaction objects.

5.46.2.3 std::vector<MassBalance> SHARK_DATA::MassBalanceList

Mass balance objects.

5.46.2.4 std::vector<UnsteadyReaction> SHARK_DATA::UnsteadyList

Unsteady Reaction objects.

5.46.2.5 std::vector<AdsorptionReaction> SHARK_DATA::AdsorptionList

Equilibrium Adsorption Reaction Objects.

5.46.2.6 std::vector<UnsteadyAdsorption> SHARK_DATA::UnsteadyAdsList

Unsteady Adsorption Reaction Objects.

5.46.2.7 std::vector<MultiligandAdsorption> SHARK_DATA::MultiAdsList

Multiligand Adsorptioin Objects.

5.46.2.8 std::vector<ChemisorptionReaction> SHARK_DATA::ChemisorptionList

Chemisorption Reaction objects.

5.46.2.9 std::vector<MultiligandChemisorption> SHARK_DATA::MultiChemList

Multiligand Chemisorption Reaction Objects.

5.46.2.10 std::vector< double (*) (const Matrix<double> &x, SHARK_DATA *shark_dat, const void *data) > SHARK_DATA::OtherList

Array of Other Residual functions to be defined by user.

This list of function pointers can be declared and set up by the user in order to add to or change the behavior of the SHARK system. Each one must be declared setup individually by the user. They will be called by the shark↩ _residual function when needed. Alternatively, the user is free to provide their own shark_residual function for the overall system.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
shark_dat	pointer to the SHARK_DATA data structure
data	pointer to a user defined data structure that is used to evaluate this residual

5.46.2.11 int SHARK_DATA::numvar

Total number of functions and species.

5.46.2.12 int SHARK_DATA::num_ssr

Number of steady-state reactions.

5.46.2.13 int SHARK_DATA::num_mbe

Number of mass balance equations.

5.46.2.14 int SHARK_DATA::num_usr = 0

Number of unsteady-state reactions.

5.46.2.15 int SHARK_DATA::num_ssao = 0

Number of steady-state adsorption objects.

5.46.2.16 int SHARK_DATA::num_usao = 0

Number of unsteady adsorption objects.

5.46.2.17 int SHARK_DATA::num_multi_ssao = 0

Number of multiligand steady-state adsorption objects.

5.46.2.18 int SHARK_DATA::num_sschem = 0

Number of steady-state chemisorption objects.

5.46.2.19 int SHARK_DATA::num_multi_sschem = 0

Number of multiligand steady-state chemisorption objects.

5.46.2.20 std::vector<int> SHARK_DATA::num_ssar

List of the numbers of reactions in each adsorption object.

5.46.2.21 std::vector<int> SHARK_DATA::num_usar

List of the numbers of reactions in each unsteady adsorption object.

5.46.2.22 std::vector<int> SHARK_DATA::num_sschem_rxns

List of the numbers of reactions in each steady-state chemisorption object.

5.46.2.23 std::vector< std::vector<int> > SHARK_DATA::num_multi_ssar

List of all multiligand objects -> List of ligands and rxns of that ligand.

5.46.2.24 std::vector< std::vector<int> > SHARK_DATA::num_multichem_rxns

List of all multiligand chemisorption objects -> List of num rxns for that ligand.

5.46.2.25 std::vector<std::string> SHARK_DATA::ss_ads_names

List of the steady-state adsorbent object names.

5.46.2.26 std::vector<std::string> SHARK_DATA::us_ads_names

List of the unsteady adsorption object names.

5.46.2.27 std::vector<std::string> SHARK_DATA::ss_chem_names

List of the steady-state chemisorption object names.

5.46.2.28 std::vector< std::vector<std::string> > SHARK_DATA::ssmulti_names

List of the names of the ligands in each multiligand object.

5.46.2.29 std::vector< std::vector<std::string> > SHARK_DATA::ssmultichem_names

List of the names of the ligands in each multiligand chemisorption object.

5.46.2.30 int SHARK_DATA::num_other = 0

Number of other functions to be used (default is always 0)

5.46.2.31 int SHARK_DATA::act_fun = IDEAL

Flag denoting the activity function to use (default is IDEAL)

5.46.2.32 int SHARK_DATA::reactor_type = BATCH

Flag denoting the type of reactor considered for the system (default is BATCH)

5.46.2.33 int SHARK_DATA::totalsteps = 0

Total number of iterations.

5.46.2.34 int SHARK_DATA::totalcalls = 0

Total number of residual function calls.

5.46.2.35 int SHARK_DATA::timesteps = 0

Number of time steps taken to complete simulation.

5.46.2.36 int SHARK_DATA::pH_index = -1

Contains the index of the pH variable (set internally)

5.46.2.37 int SHARK_DATA::pOH_index = -1

Contains the index of the pOH variable (set internally)

5.46.2.38 double SHARK_DATA::simulationtime = 0.0

Time to simulate unsteady reactions for (default = 0.0 hrs)

5.46.2.39 double SHARK_DATA::dt = 0.1

Time step size (hrs)

5.46.2.40 double SHARK_DATA::dt_min = sqrt(DBL_EPSILON)

Minimum allowable step size.

5.46.2.41 double SHARK_DATA::dt_max = 744.0

Maximum allowable step size ($\sim$1 month in time)

5.46.2.42 double SHARK_DATA::t_out = 0.0

Time increment by which file output is made (default = print all time steps)

5.46.2.43 double SHARK_DATA::t_count = 0.0

Running count of time increments.

5.46.2.44 double SHARK_DATA::time = 0.0

Current value of time (starts from t = 0.0 hrs)

5.46.2.45 double SHARK_DATA::time_old = 0.0

Previous value of time (start from t = 0.0 hrs)

5.46.2.46 double SHARK_DATA::pH = 7.0

Value of pH if needed (default = 7)

5.46.2.47 double SHARK_DATA::pH_step = 0.5

Value by which to increment pH when doing a speciation curve (default = 0.5)

5.46.2.48 double SHARK_DATA::start_temp = 277.15

Value of the starting temperature used for Temperature Curves (default = 277.15 K)

5.46.2.49 double SHARK_DATA::end_temp = 323.15

Value of the ending temperature used for Temperature Curves (default = 323.15 K)

5.46.2.50 double SHARK_DATA::temp_step = 10.0

Size of the step changes to use for Temperature Curves (default = 10.0 K);.

5.46.2.51 double SHARK_DATA::volume = 1.0

Volume of the domain in liters (default = 1 L)

5.46.2.52 double SHARK_DATA::flow_rate = 1.0

Flow rate in the reactor in L/hr (default = 1 L/hr)

5.46.2.53 double SHARK_DATA::xsec_area = 1.0

Cross sectional area of the reactor in m^2 (default = 1 m^2)

5.46.2.54 double SHARK_DATA::Norm = 0.0

Current value of euclidean norm in solution.

5.46.2.55 double SHARK_DATA::dielectric_const = 78.325

Dielectric constant used in many activity models (default: water = 78.325 (1/K))

5.46.2.56 double SHARK_DATA::relative_permittivity = 80.1

Relative permittivity of the medium (default: water = 80.1 (-))

5.46.2.57 double SHARK_DATA::temperature = 298.15

Solution temperature (default = 25 oC or 298.15 K)

5.46.2.58 double SHARK_DATA::ionic_strength = 0.0

Solution ionic strength in Molar (calculated internally)

5.46.2.59 bool SHARK_DATA::steadystate = true

True = solve steady problem; False = solve transient problem.

5.46.2.60 bool SHARK_DATA::ZeroInitialSolids = false

True = no solids or adsorption initially in the reactor.

5.46.2.61 bool SHARK_DATA::TimeAdaptivity = false

True = solve using variable time step.

5.46.2.62 bool SHARK_DATA::const_pH = false

True = set pH to a constant; False = solve for pH.

5.46.2.63 bool SHARK_DATA::SpeciationCurve = false

True = runs a series of constant pH steady-state problems to produce curves.

5.46.2.64 bool SHARK_DATA::TemperatureCurve = false

True = runs a series of constant temperature steady-state problmes to produce curves.

5.46.2.65 bool SHARK_DATA::Console_Output = true

True = display output to console.

5.46.2.66 bool SHARK_DATA::File_Output = false

True = write output to a file.

5.46.2.67 bool SHARK_DATA::Contains_pH = false

True = system contains pH as a variable (set internally)

5.46.2.68 bool SHARK_DATA::Contains_pOH = false

True = system contains pOH as a variable (set internally)

5.46.2.69 bool SHARK_DATA::Converged = false

True = system converged within tolerance.

5.46.2.70 bool SHARK_DATA::LocalMin = true

True = allow the system to settle for a local minimum if tolerance not reached.

5.46.2.71 Matrix<double> SHARK_DATA::X_old

Solution vector for old time step - log(C)

5.46.2.72 Matrix<double> SHARK_DATA::X_new

Solution vector for current time step - log(C)

5.46.2.73 Matrix<double> SHARK_DATA::Conc_old

Concentration vector for old time step - 10^x.

5.46.2.74 Matrix<double> SHARK_DATA::Conc_new

Concentration vector for current time step - 10^x.

5.46.2.75 Matrix<double> SHARK_DATA::activity_new

Activity matrix for current time step.

5.46.2.76 Matrix<double> SHARK_DATA::activity_old

Activity matrix from prior time step.

5.46.2.77 int(* SHARK_DATA::EvalActivity) (const Matrix< double > &x, Matrix< double > &F, const void *data)

Function pointer to evaluate activity coefficients.

This function pointer is called within the shark_residual function to calculate and modify the activity_new matrix entries. When using the SHARK default options, this function pointer will be automatically set to a cooresponding activity function for the list of valid functions from the valid_act enum. User may override this function pointer if they desire. Must be overriden after calling the setup function.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

5.46.2.78 int(* SHARK_DATA::Residual) (const Matrix< double > &x, Matrix< double > &F, const void *data)

Function pointer to evaluate all residuals in the system.

This function will be fed into the PJFNK solver (see lark.h) to solve the non-linear system of equations. By default, this pointer will be the shark_residual function (see below). However, the user may override the function and provide their own residuals for the PJFNK solver to operate on.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of residuals that are to be altered from the functions in the system
data	pointer to a data structure needed to evaluate the activity model

5.46.2.79 int(* SHARK_DATA::lin_precon) (const Matrix< double > &r, Matrix< double > &p, const void *data)

Function pointer to form a linear preconditioning operation for the Jacobian.

This function will be fed into the linear solver used for each non-linear step in PJFNK (see lark.h). By default, we cannot provide any linear preconditioner, because we do not know the form or sparcity of the Jacobian before hand. It will be the user's responsibility to form their own preconditioner until we can figure out a generic way to precondition the system.

5.46.2.80 PJFNK_DATA SHARK_DATA::Newton_data

Data structure for the Newton-Krylov solver (see lark.h)

5.46.2.81 const void* SHARK_DATA::activity_data

User defined data structure for an activity model.

5.46.2.82 const void* SHARK_DATA::residual_data

User defined data structure for the residual function.

5.46.2.83 const void* SHARK_DATA::precon_data

User defined data structure for preconditioning.

5.46.2.84 const void* SHARK_DATA::other_data

User define data structure used for user defined residuals.

5.46.2.85 FILE* SHARK_DATA::OutputFile

Output File pointer.

5.46.2.86 yaml_cpp_class SHARK_DATA::yaml_object

yaml object to read and access digitized yaml documents (see yaml_wrapper.h)

The documentation for this struct was generated from the following file:

- shark.h

5.47 SKUA_DATA Struct Reference

Data structure for all simulation information in SKUA.

`#include <skua.h>`

Public Attributes

- unsigned long int total_steps

  Running total of all calculation steps.
- int coord

  Used to determine the coordinates of the problem.
- double sim_time

  Stopping time for the simulation (hrs)
- double t_old

  Old time of the simulations (hrs)
- double t

  Current time of the simulations (hrs)
- double t_counter = 0.0

  Counts for print times for output (hrs)
- double t_print

  Prints out every t_print time (hrs)
- double qTn

  Old total amounts adsorbed (mol/kg)
- double qTnp1

  New total amounts adsorbed (mol/kg)
- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- double gas_velocity

  Superficial Gas Velocity around pellet (cm/s)
- double pellet_radius

  Nominal radius of the pellet/crystal (um)
- double char_measure

  Length or Area if in Cylindrical or Cartesian coordinates (um or um^2)
- bool DirichletBC = true

  True = Dirichlet BC; False = Neumann BC.
- bool NonLinear = true

  True = Non-linear solver; False = Linear solver.
- std::vector< double > y

  Outside mole fractions of each component (-)
- FILE * OutputFile

  Output file pointer to the output file.
- double(* eval_diff)(int i, int l, const void *user_data)

  Function pointer for evaluating surface diffusivity.
- double(* eval_kf)(int i, const void *user_data)

  Function pointer for evaluating film mass transfer.
- const void * user_data

  Data structure for user's information needed in parameter functions.
- MAGPIE_DATA magpie_dat

  Data structure for adsorption equilibria (see magpie.h)
- MIXED_GAS * gas_dat

  Pointer to the MIXED_GAS data structure (see egret.h)
- std::vector< FINCH_DATA > finch_dat

  Data structure for adsorption kinetics (see finch.h)
- std::vector< SKUA_PARAM > param_dat

  Data structure for SKUA specific parameters.

5.47.1 Detailed Description

Data structure for all simulation information in SKUA.

C-style object holding all data, functions, and other objects needed to successfully run a SKUA simulation. This object holds system information, such as boundary condition type, adsorbent size, and total adsorption, and also contains structure for EGRET (egret.h), FINCH (finch.h), and MAGPIE (magpie.h) calculations. Function pointers for evaluation of the surface diffusivity and film mass transfer coefficients can be overriden by the user to change the behavior of the SKUA simulation. However, defaults are also provided for these functions.

5.47.2 Member Data Documentation

5.47.2.1 unsigned long int SKUA_DATA::total_steps

Running total of all calculation steps.

5.47.2.2 int SKUA_DATA::coord

Used to determine the coordinates of the problem.

5.47.2.3 double SKUA_DATA::sim_time

Stopping time for the simulation (hrs)

5.47.2.4 double SKUA_DATA::t_old

Old time of the simulations (hrs)

5.47.2.5 double SKUA_DATA::t

Current time of the simulations (hrs)

5.47.2.6 double SKUA_DATA::t_counter = 0.0

Counts for print times for output (hrs)

5.47.2.7 double SKUA_DATA::t_print

Prints out every t_print time (hrs)

5.47.2.8 double SKUA_DATA::qTn

Old total amounts adsorbed (mol/kg)

5.47.2.9 double SKUA_DATA::qTnp1

New total amounts adsorbed (mol/kg)

5.47.2.10 bool SKUA_DATA::Print2File = true

True = results to .txt; False = no printing.

5.47.2.11 bool SKUA_DATA::Print2Console = true

True = results to console; False = no printing.

5.47.2.12 double SKUA_DATA::gas_velocity

Superficial Gas Velocity arount pellet (cm/s)

5.47.2.13 double SKUA_DATA::pellet_radius

Nominal radius of the pellet/crystal (um)

5.47.2.14 double SKUA_DATA::char_measure

Length or Area if in Cylindrical or Cartesian coordinates (um or um^2)

5.47.2.15 bool SKUA_DATA::DirichletBC = true

True = Dirichlet BC; False = Neumann BC.

5.47.2.16 bool SKUA_DATA::NonLinear = true

True = Non-linear solver; False = Linear solver.

5.47.2.17 std::vector<double> SKUA_DATA::y

Outside mole fractions of each component (-)

5.47.2.18 FILE* SKUA_DATA::OutputFile

Output file pointer to the output file.

5.47.2.19 double(* SKUA_DATA::eval_diff) (int i, int l, const void *user_data)

Function pointer for evaluating surface diffusivity.

5.47.2.20 double(* SKUA_DATA::eval_kf) (int i, const void *user_data)

Function pointer for evaluating film mass transfer.

5.47.2.21 const void* SKUA_DATA::user_data

Data structure for user's information needed in parameter functions.

5.47.2.22 MAGPIE_DATA SKUA_DATA::magpie_dat

Data structure for adsorption equilibria (see magpie.h)

5.47.2.23 MIXED_GAS* SKUA_DATA::gas_dat

Pointer to the MIXED_GAS data structure (see egret.h)

5.47.2.24 std::vector<FINCH_DATA> SKUA_DATA::finch_dat

Data structure for adsorption kinetics (see finch.h)

5.47.2.25 std::vector<SKUA_PARAM> SKUA_DATA::param_dat

Data structure for SKUA specific parameters.

The documentation for this struct was generated from the following file:

- skua.h

5.48 SKUA_OPT_DATA Struct Reference

Data structure for the SKUA Optimization Routine.

`#include <skua_opt.h>`

Public Attributes

- int num_curves
  Number of adsorption curves to analyze.
- int evaluation
  Number of times the eval function has been called for a single curve.
- unsigned long int total_eval
  Total number of evaluations needed for completion.
- int current_points
  Number of points in the current curve.
- int num_params = 1
  Number of adjustable parameters for the current curve.
- int diffusion_type
  Flag to identify type of diffusion function to use.
- int adsorb_index
  Component index for adsorbable species.
- int max_guess_iter = 20
  Maximum allowed guess iterations (default = 20)
- bool Optimize
  True = run optimization, False = run a comparison.
- bool Rough
  True = use only a rough estimate, False = run full optimization.
- double current_temp

Temperature for current curve.

- double current_press

  Partial pressure for current curve.
- double current_equil

  Equilibrium data point for the current curve.
- double simulation_equil

  Equilibrium simulation point for the current curve.
- double max_bias

  Positive maximum bias plausible for fitting.
- double min_bias

  Negative minimum bias plausible for fitting.
- double e_norm

  Euclidean norm of current fit.
- double f_bias

  Function bias of current fit.
- double e_norm_old

  Euclidean norm of the previous fit.
- double f_bias_old

  Function bias of the previous fit.
- double param_guess

  Parameter guess for the surface/crystal diffusivity.
- double param_guess_old

  Parameter guess for the previous curve.
- double rel_tol_norm = 0.1

  Tolerance for convergence of the guess norm.
- double abs_tol_bias = 0.1

  Tolerance for convergence of the guess bias.
- std::vector< double > y_base

  Gas phase mole fractions in absense of adsorbing species.
- std::vector< double > q_data

  Amount adsorbed at a particular point in current curve.
- std::vector< double > q_sim

  Amount adsorbed based on the simulation.
- std::vector< double > t

  Time points in the current curve.
- FILE * ParamFile

  Output file for parameter results.
- FILE * CompareFile

  Output file for comparison of results.
- SKUA_DATA skua_dat

  Data structure for the SKUA simulation.

5.48.1 Detailed Description

Data structure for the SKUA Optimization Routine.

C-style object holding data and pointers necessary for running a SKUA optimization. It contains information about the type of optimization requested, the current status of the optimization, the data being compared against, and the SKUA_DATA object for the evaluation of a SKUA simulation. The pointers in the structure are for the two output files produced by the routine: (i) parameter results and (ii) model comparison results.

5.48.2 Member Data Documentation

5.48.2.1 int SKUA_OPT_DATA::num_curves

Number of adsorption curves to analyze.

5.48.2.2 int SKUA_OPT_DATA::evaluation

Number of times the eval function has been called for a single curve.

5.48.2.3 unsigned long int SKUA_OPT_DATA::total_eval

Total number of evaluations needed for completion.

5.48.2.4 int SKUA_OPT_DATA::current_points

Number of points in the current curve.

5.48.2.5 int SKUA_OPT_DATA::num_params = 1

Number of adjustable parameters for the current curve.

5.48.2.6 int SKUA_OPT_DATA::diffusion_type

Flag to identify type of diffusion function to use.

5.48.2.7 int SKUA_OPT_DATA::adsorb_index

Component index for adsorbable species.

5.48.2.8 int SKUA_OPT_DATA::max_guess_iter = 20

Maximum allowed guess iterations (default = 20)

5.48.2.9 bool SKUA_OPT_DATA::Optimize

True = run optimization, False = run a comparison.

5.48.2.10 bool SKUA_OPT_DATA::Rough

True = use only a rough estimate, False = run full optimization.

5.48.2.11 double SKUA_OPT_DATA::current_temp

Temperature for current curve.

5.48.2.12 double SKUA_OPT_DATA::current_press

Partial pressure for current curve.

5.48.2.13 double SKUA_OPT_DATA::current_equil

Equilibrium data point for the current curve.

5.48.2.14 double SKUA_OPT_DATA::simulation_equil

Equilibrium simulation point for the current curve.

5.48.2.15 double SKUA_OPT_DATA::max_bias

Positive maximum bias plausible for fitting.

5.48.2.16 double SKUA_OPT_DATA::min_bias

Negative minimum bias plausible for fitting.

5.48.2.17 double SKUA_OPT_DATA::e_norm

Euclidean norm of current fit.

5.48.2.18 double SKUA_OPT_DATA::f_bias

Function bias of current fit.

5.48.2.19 double SKUA_OPT_DATA::e_norm_old

Euclidean norm of the previous fit.

5.48.2.20 double SKUA_OPT_DATA::f_bias_old

Function bias of the previous fit.

5.48.2.21 double SKUA_OPT_DATA::param_guess

Parameter guess for the surface/crystal diffusivity.

5.48.2.22 double SKUA_OPT_DATA::param_guess_old

Parameter guess for the previous curve.

5.48.2.23 double SKUA_OPT_DATA::rel_tol_norm = 0.1

Tolerance for convergence of the guess norm.

5.48.2.24 double SKUA_OPT_DATA::abs_tol_bias = 0.1

Tolerance for convergence of the guess bias.

5.48.2.25 std::vector<double> SKUA_OPT_DATA::y_base

Gas phase mole fractions in absense of adsorbing species.

5.48.2.26 std::vector<double> SKUA_OPT_DATA::q_data

Amount adsorbed at a particular point in current curve.

5.48.2.27 std::vector<double> SKUA_OPT_DATA::q_sim

Amount adsorbed based on the simulation.

5.48.2.28 std::vector<double> SKUA_OPT_DATA::t

Time points in the current curve.

5.48.2.29 FILE* SKUA_OPT_DATA::ParamFile

Output file for parameter results.

5.48.2.30 FILE* SKUA_OPT_DATA::CompareFile

Output file for comparison of results.

5.48.2.31 SKUA_DATA SKUA_OPT_DATA::skua_dat

Data structure for the SKUA simulation.

The documentation for this struct was generated from the following file:

- skua_opt.h

5.49 SKUA_PARAM Struct Reference

Data structure for species' parameters in SKUA.

```
#include <skua.h>
```

Public Attributes

- double activation_energy
- double ref_diffusion
- double ref_temperature
- double affinity
- double ref_pressure
- double film_transfer
- double xIC
- double y_eff
- double Qstn
- double Qstnp1
- double xn
- double xnp1
- bool Adsorbable
- std::string speciesName

5.49.1 Detailed Description

Data structure for species' parameters in SKUA.

C-style object holding data and parameters associated with the gas/solid species in the overall SKUA system. These parameters are used in to modify surface diffusivity with temperature, establish film mass transfer coefficients, formulate the initial conditions, and store solution results for heat of adsorption and adsorbed mole fractions. One of these objects will be created for each species in the gas system.

5.49.2 Member Data Documentation

5.49.2.1 double SKUA_PARAM::activation_energy

5.49.2.2 double SKUA_PARAM::ref_diffusion

5.49.2.3 double SKUA_PARAM::ref_temperature

5.49.2.4 double SKUA_PARAM::affinity

5.49.2.5 double SKUA_PARAM::ref_pressure

5.49.2.6 double SKUA_PARAM::film_transfer

5.49.2.7 double SKUA_PARAM::xIC

5.49.2.8 double SKUA_PARAM::y_eff

5.49.2.9 double SKUA_PARAM::Qstn

5.49.2.10 double SKUA_PARAM::Qstnp1

5.49.2.11 double SKUA_PARAM::xn

5.49.2.12 double SKUA_PARAM::xnp1

5.49.2.13 bool SKUA_PARAM::Adsorbable

5.49.2.14 std::string SKUA_PARAM::speciesName

The documentation for this struct was generated from the following file:

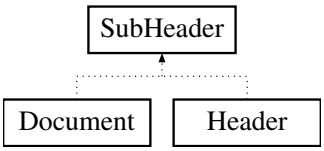
- skua.h

5.50 SubHeader Class Reference

Object for the Lowest level of Header for the yaml_wrapper.

`#include <yaml_wrapper.h>`

Inheritance diagram for SubHeader:



Public Member Functions

- SubHeader ()
  Default Constructor.
- ~SubHeader ()
  Default Destructor.
- SubHeader (const SubHeader &subheader)
  Copy constructor.
- SubHeader (const KeyValueMap &map)
  Construction by existing map.
- SubHeader (std::string name)
  Construction by name only.
- SubHeader (std::string name, const KeyValueMap &map)
  Construction by name and map.
- SubHeader & operator= (const SubHeader &sub)
  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)
  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const
  Return the ValueType at the give key.
- KeyValueMap & getMap ()
  Returns reference to the KeyValueMap object.
- void clear ()
  Empty out data contents.
- void addPair (std::string key, std::string val)
  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int type)
  Adds a pair object and asserts a type.
- void setName (std::string name)
  Sets the name of the subheader.
- void setAlias (std::string alias)
  Set the alias without type specification.
- void setAlias (std::string alias, int state)
  Sets the alias and state of the subheader.
- void setNameAliasPair (std::string name, std::string alias, int state)
  Sets the name and alias of the subheader.
- void setState (int state)

Sets the state of the subheader.

- void DisplayContents ()

  Display the contents of the subheader.
- std::string getName ()

  Return the name of the subheader.
- std::string getAlias ()

  Return the alias of the subheader, if one exists.
- bool isAlias ()

  Returns true if subheader is an alias.
- bool isAnchor ()

  Returns true if subheader is an anchor.
- int getState ()

  Returns the state of the subheader.

Protected Attributes

- KeyValueMap Data_Map

  A Map of Keys and Values.
- std::string name

  Name of the subheader.
- std::string alias

  Name of the alias for the subheader.
- int state

  State of the header.

5.50.1 Detailed Description

Object for the Lowest level of Header for the yaml_wrapper.

C++ Object for sub-headers in a yaml document. This object contains a KeyValueMap that holds a set of key-value pairs for data listed under a sub-header in yaml files. It is the lowest allowable recursion of headers in a yaml document and so is the base class for Header and Document, which themselves can contain KeyValueMaps as well as maps for other header-like objects.

SubHeaders are recognized by a unique name and/or alias while being put together in other higher document structures. Additionally, each header or sub-header will have a state to denote whether the object is a yaml alias, anchor, or niether. This is used in the yaml documents to ensure that aliases for anchors have the correct data moved over into the new structures.

5.50.2 Constructor & Destructor Documentation

5.50.2.1 SubHeader::SubHeader ( )

Default Constructor.

5.50.2.2 SubHeader::~SubHeader ( )

Default Destructor.

5.50.2.3 SubHeader::SubHeader (const SubHeader & *subheader*)

Copy constructor.

5.50.2.4 SubHeader::SubHeader (const KeyValueMap & *map*)

Construction by existing map.

5.50.2.5 SubHeader::SubHeader (std::string *name*)

Construction by name only.

5.50.2.6 SubHeader::SubHeader (std::string *name,* const KeyValueMap & *map*)

Construction by name and map.

5.50.3 Member Function Documentation

5.50.3.1 SubHeader& SubHeader::operator= (const SubHeader & *sub*)

Equals overload.

5.50.3.2 ValueTypePair& SubHeader::operator[ ] (const std::string *key*)

Return the ValueType reference at the given key.

5.50.3.3 ValueTypePair SubHeader::operator[ ] (const std::string *key*) const

Return the ValueType at the give key.

5.50.3.4 KeyValueMap& SubHeader::getMap ( )

Returns reference to the KeyValueMap object.

5.50.3.5 void SubHeader::clear ( )

Empty out data contents.

5.50.3.6 void SubHeader::addPair (std::string *key,* std::string *val*)

Adds a pair object to the map (with only strings)

5.50.3.7 void SubHeader::addPair (std::string *key,* std::string *val,* int *type*)

Adds a pair object and asserts a type.

5.50.3.8 void SubHeader::setName (std::string *name*)

Sets the name of the subheader.

5.50.3.9 void SubHeader::setAlias (std::string *alias*)

Set the alias without type specification.

5.50.3.10 void SubHeader::setAlias (std::string *alias,* int *state*)

Sets the alias and state of the subheader.

5.50.3.11 void SubHeader::setNameAliasPair (std::string *name,* std::string *alias,* int *state*)

Sets the name and alias of the subheader.

5.50.3.12 void SubHeader::setState (int *state*)

Sets the state of the subheader.

5.50.3.13 void SubHeader::DisplayContents ( )

Display the contents of the subheader.

5.50.3.14 std::string SubHeader::getName ( )

Return the name of the subheader.

5.50.3.15 std::string SubHeader::getAlias ( )

Return the alias of the subheader, if one exists.

5.50.3.16 bool SubHeader::isAlias ( )

Returns true if subheader is an alias.

5.50.3.17 bool SubHeader::isAnchor ( )

Returns true if subheader is an anchor.

5.50.3.18 int SubHeader::getState ( )

Returns the state of the subheader.

5.50.4 Member Data Documentation

5.50.4.1 KeyValueMap SubHeader::Data_Map [protected]

A Map of Keys and Values.

5.50.4.2 std::string SubHeader::name [protected]

Name of the subheader.

5.50.4.3 std::string SubHeader::alias [protected]

Name of the alias for the subheader.

5.50.4.4 int SubHeader::state [protected]

State of the header.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.51 SYSTEM_DATA Struct Reference

System Data Structure.

`#include <magpie.h>`

Public Attributes

- double T
  System Temperature (K)
- double PT
  Total Pressure (kPa)
- double qT
  Total Amount adsorbed (mol/kg)
- double PI
  Total Lumped Spreading Pressure (mol/kg)
- double pi
  Actual Spreading pressure (J/m^2)
- double As
  Specific surface area of adsorbent (m^2/kg)
- int N
  Total Number of Components.
- int I
- int J
- int K
  Special indices used to keep track of sub-systems.
- unsigned long int total_eval
  Counter to keep track of total number of non-linear steps.
- double avg_norm
  Used to store all norms from evaluations then average at end of run.
- double max_norm
  Used to store the maximum e.norm calculated from non-linear iterations.
- int Sys
  Number of sub-systems to solve.
- int Par
  Number of binary parameters to solve for.
- bool Recover
  If Recover == false, standard GPAST using y's as knowns.
- bool Carrier
  If there is an inert carrier gas, Carrier == true.
- bool Ideal
  If the behavior of the system is determined to be ideal, then Ideal == true.
- bool Output
  Boolean to suppress output if desired (true = display, false = no display.

5.51.1 Detailed Description

System Data Structure.

C-style object holding all the data associated with the overall system to be modeled.

5.51.2 Member Data Documentation

5.51.2.1 double SYSTEM_DATA::T

System Temperature (K)

5.51.2.2 double SYSTEM_DATA::PT

Total Pressure (kPa)

5.51.2.3 double SYSTEM_DATA::qT

Total Amount adsorbed (mol/kg)

5.51.2.4 double SYSTEM_DATA::PI

Total Lumped Spreading Pressure (mol/kg)

5.51.2.5 double SYSTEM_DATA::pi

Actual Spreading pressure (J/m^2)

5.51.2.6 double SYSTEM_DATA::As

Specific surface area of adsorbent (m^2/kg)

5.51.2.7 int SYSTEM_DATA::N

Total Number of Components.

5.51.2.8 int SYSTEM_DATA::I

5.51.2.9 int SYSTEM_DATA::J

5.51.2.10 int SYSTEM_DATA::K

Special indices used to keep track of sub-systems.

5.51.2.11 unsigned long int SYSTEM_DATA::total_eval

Counter to keep track of total number of non-linear steps.

5.51.2.12 double SYSTEM_DATA::avg_norm

Used to store all norms from evaluations then average at end of run.

5.51.2.13 double SYSTEM_DATA::max_norm

Used to store the maximum e.norm calculated from non-linear iterations.

5.51.2.14 int SYSTEM_DATA::Sys

Number of sub-systems to solve.

5.51.2.15 int SYSTEM_DATA::Par

Number of binary parameters to solve for.

5.51.2.16 bool SYSTEM_DATA::Recover

If Recover == false, standard GPAST using y's as knowns.

5.51.2.17 bool SYSTEM_DATA::Carrier

If there is an inert carrier gas, Carrier == true.

5.51.2.18 bool SYSTEM_DATA::Ideal

If the behavior of the system is determined to be ideal, then Ideal == true.

5.51.2.19 bool SYSTEM_DATA::Output

Boolean to suppress output if desired (true = display, false = no display.

The documentation for this struct was generated from the following file:

- magpie.h

5.52 TRAJECTORY_DATA Struct Reference

`#include <Trajectory.h>`

Public Attributes

- double mu_0 = 12.57e-7

  permeability of free space, H/m
- double rho_f = 1000.0

  Fluid density, Kg/m3.
- double eta = 0.001
- double Hamaker = 1.3e-21
- double Temp = 298
- double k = 1.38e-23
- double Rs
- double L
- double porosity
- double V_separator
- double a
- double V_wire
- double L_wire
- double A_separator
- double A_wire
- double B0
- double H0
- double Ms = 0.6
- double b
- double chi_p
- double rho_p = 8000.0
- double Q_in
- double V0
- double Y_initial = 20.0
- double dt
- double M
- double mp
- double beta
- double q_bar
- double sigma_v
- double sigma_vz
- double sigma_z
- double sigma_n
- double sigma_m
- double n_rand
- double m_rand
- double s_rand
- double t_rand
- Matrix< double > POL
- Matrix< double > H
- Matrix< double > dX
- Matrix< double > dY
- Matrix< double > Vr
- Matrix< double > Vt
- Matrix< double > X
- Matrix< double > Y
- Matrix< int > Cap

5.52.1 Member Data Documentation

5.52.1.1 double TRAJECTORY_DATA::mu_0 = 12.57e-7

permeability of free space, H/m

5.52.1.2 double TRAJECTORY_DATA::rho_f = 1000.0

Fluid density, Kg/m3.

5.52.1.3 double TRAJECTORY_DATA::eta = 0.001

5.52.1.4 double TRAJECTORY_DATA::Hamaker = 1.3e-21

5.52.1.5 double TRAJECTORY_DATA::Temp = 298

5.52.1.6 double TRAJECTORY_DATA::k = 1.38e-23

5.52.1.7 double TRAJECTORY_DATA::Rs

5.52.1.8 double TRAJECTORY_DATA::L

5.52.1.9 double TRAJECTORY_DATA::porosity

5.52.1.10 double TRAJECTORY_DATA::V_separator

5.52.1.11 double TRAJECTORY_DATA::a

5.52.1.12 double TRAJECTORY_DATA::V_wire

5.52.1.13 double TRAJECTORY_DATA::L_wire

5.52.1.14 double TRAJECTORY_DATA::A_separator

5.52.1.15 double TRAJECTORY_DATA::A_wire

5.52.1.16 double TRAJECTORY_DATA::B0

5.52.1.17 double TRAJECTORY_DATA::H0

5.52.1.18 double TRAJECTORY_DATA::Ms = 0.6

5.52.1.19 double TRAJECTORY_DATA::b

5.52.1.20 double TRAJECTORY_DATA::chi_p

5.52.1.21 double TRAJECTORY_DATA::rho_p = 8000.0

5.52.1.22 double TRAJECTORY_DATA::Q_in

5.52.1.23 double TRAJECTORY_DATA::V0

5.52.1.24 double TRAJECTORY_DATA::Y_initial = 20.0

5.52.1.25 double TRAJECTORY_DATA::dt

5.52.1.26 double TRAJECTORY_DATA::M

5.52.1.27 double TRAJECTORY_DATA::mp

5.52.1.28 double TRAJECTORY_DATA::beta

5.52.1.29 double TRAJECTORY_DATA::q_bar

5.52.1.30 double TRAJECTORY_DATA::sigma_v

5.52.1.31 double TRAJECTORY_DATA::sigma_vz

5.52.1.32 double TRAJECTORY_DATA::sigma_z

5.52.1.33 double TRAJECTORY_DATA::sigma_n

5.52.1.34 double TRAJECTORY_DATA::sigma_m

5.52.1.35 double TRAJECTORY_DATA::n_rand

5.52.1.36 double TRAJECTORY_DATA::m_rand

5.52.1.37 double TRAJECTORY_DATA::s_rand

5.52.1.38 double TRAJECTORY_DATA::t_rand

5.52.1.39 Matrix<double> TRAJECTORY_DATA::POL

5.52.1.40 Matrix<double> TRAJECTORY_DATA::H

5.52.1.41 Matrix<double> TRAJECTORY_DATA::dX

5.52.1.42 Matrix<double> TRAJECTORY_DATA::dY

5.52.1.43 Matrix<double> TRAJECTORY_DATA::Vr

5.52.1.44 Matrix<double> TRAJECTORY_DATA::Vt

5.52.1.45 Matrix<double> TRAJECTORY_DATA::X

5.52.1.46 Matrix<double> TRAJECTORY_DATA::Y

5.52.1.47 Matrix<int> TRAJECTORY_DATA::Cap

The documentation for this struct was generated from the following file:

- Trajectory.h

5.53 UI_DATA Struct Reference

Data structure holding the UI arguments.

`#include <ui.h>`

Public Attributes

- ValueTypePair value_type
  Data pair for input, tells what the input is and it's type.
- std::vector< std::string > user_input
  What is read in from the console at any point.
- std::vector< std::string > input_files
  A vector of input file names and directories given by user.
- std::string path
  Path to where input files are located.
- int count = 0
  Number of times a questing has been asked.
- int max = 3
  Maximum allowable recursions of a question.
- int option
  Current option choosen by the user.
- bool Path = false
  True if user gives path as an option.
- bool Files = false
  True if user gives input files as an option.
- bool MissingArg = true
  True if an input argument is missing; False if everything is ok.
- bool BasicUI = true
  True if using Basic UI; False if using Advanced UI.
- int argc
  Number of console arguments given on input.
- const char * argv [ ]
  Actual console arguments given at execution.

5.53.1 Detailed Description

Data structure holding the UI arguments.

C-Style object for interfacing with users request upon execution of the program. User input is stored in objects below and a series of booleans is used to determine how and what to execute.

5.53.2 Member Data Documentation

5.53.2.1 ValueTypePair UI_DATA::value_type

Data pair for input, tells what the input is and it's type.

5.53.2.2 std::vector<std::string> UI_DATA::user_input

What is read in from the console at any point.

5.53.2.3 std::vector<std::string> UI_DATA::input_files

A vector of input file names and directories given by user.

5.53.2.4 std::string UI_DATA::path

Path to where input files are located.

5.53.2.5 int UI_DATA::count = 0

Number of times a questing has been asked.

5.53.2.6 int UI_DATA::max = 3

Maximum allowable recursions of a question.

5.53.2.7 int UI_DATA::option

Current option choosen by the user.

5.53.2.8 bool UI_DATA::Path = false

True if user gives path as an option.

5.53.2.9 bool UI_DATA::Files = false

True if user gives input files as an option.

5.53.2.10 bool UI_DATA::MissingArg = true

True if an input argument is missing; False if everything is ok.

5.53.2.11 bool UI_DATA::BasicUI = true

True if using Basic UI; False if using Advanced UI.

5.53.2.12 int UI_DATA::argc

Number of console arguments given on input.

5.53.2.13 const char* UI_DATA::argv[ ]

Actual console arguments given at execution.

The documentation for this struct was generated from the following file:

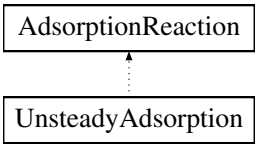
- ui.h

5.54 UnsteadyAdsorption Class Reference

Unsteady Adsorption Reaction Object.

`#include <shark.h>`

Inheritance diagram for UnsteadyAdsorption:



Public Member Functions

- UnsteadyAdsorption ()
  Default Constructor.
- ∼UnsteadyAdsorption ()
  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int n)
  Function to call the initialization of objects sequentially.
- void Display_Info ()
  Display the adsorption reaction information (PLACE HOLDER)
- void modifyDeltas (MassBalance &mbo)
  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()
  Find and set the adsorbed species indices for each reaction object.
- int checkAqueousIndices ()
  Function to check and report errors in the aqueous species indices.
- void setActivityModelInfo (int(∗act)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data), const void ∗act_data)
  Function to set the surface activity model and data pointer.
- void setAqueousIndex (int rxn_i, int species_i)
  Set the primary aqueous species index for the ith reaction.
- int setAqueousIndexAuto ()
  Automatically sets the primary aqueous species index based on reactions.
- void setActivityEnum (int act)
  Set the surface activity enum value.
- void setMolarFactor (int rxn_i, double m)
  Set the molar factor for the ith reaction (mol/mol)
- void setVolumeFactor (int i, double v)
  Set the ith volume factor for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)
  Set the ith area factor for the species list (m^2/mol)
- void setSpecificArea (double a)
  Set the specific area for the adsorbent (m^2/kg)
- void setSpecificMolality (double a)
  Set the specific molality for the adsorbent (mol/kg)
- void setSurfaceCharge (double c)
  Set the surface charge of the uncomplexed ligands.

- void setTotalMass (double m)

  Set the total mass of the adsorbent (kg)
- void setTotalVolume (double v)

  Set the total volume of the system (L)
- void setAreaBasisBool (bool opt)

  Set the basis boolean directly.
- void setSurfaceChargeBool (bool opt)

  Set the boolean for inclusion of surface charging.
- void setBasis (std::string option)

  Set the basis of the adsorption problem from the given string arg.
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent to the given string.
- void updateActivities ()

  Set the old activities as the new activities before doing next time step.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void calculateRates (double T)

  Calculates all reaction rate parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- double calculateActiveFraction (const Matrix< double > &x)

  Calculates the fraction of the surface that is active and available.
- double calculateSurfaceChargeDensity (const Matrix< double > &x)

  Function to calculate the surface charge density based on concentrations.
- double calculatePsi (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateAqueousChargeExchange (int i)

  Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int i)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_↩ perm, int i)

  Calculates the residual for the ith reaction in the system.
- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old, const Matrix< double > &gama_new, const Matrix< double > &gama_old, double T, double rel_perm, int i)

  Calculates the unsteady residual for the ith reaction in the system.
- double Eval_ReactionRate (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int i)

  Function to calculate the explicit or implicit rate of reaction.
- double Eval_IC_Residual (const Matrix< double > &x, int i)

  Calculate the unsteady residual for initial conditions.
- double Explicit_Eval (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int i)

  Return an approximate explicit solution to our unsteady adsorption variable (mol/kg)
- UnsteadyReaction & getReaction (int i)

Return reference to the ith reaction object in the adsorption object.

- double getMolarFactor (int i)

  Get the ith reaction's molar factor for adsorption (mol/mol)
- double getVolumeFactor (int i)

  Get the ith volume factor (species not involved return zeros) (cm^3/mol)
- double getAreaFactor (int i)

  Get the ith area factor (species not involved return zeros) (m^2/mol)
- double getActivity (int i)

  Get the ith activity factor for the surface species.
- double getOldActivity (int i)

  Get the ith old activity factor for the surface species.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getSpecificMolality ()

  Get the specific molality of the adsorbent (mol/kg)
- double getSurfaceCharge ()

  Get the surface charge of the adsorbent.
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- int getNumberRxns ()

  Get the number of reactions involved in the adsorption object.
- int getAdsorbIndex (int i)

  Get the index of the adsorbed species in the ith reaction.
- int getAqueousIndex (int i)

  Get the index of the primary aqueous species in the ith reaction.
- int getActivityEnum ()

  Return the enum representing the choosen activity function.
- bool isAreaBasis ()

  Returns true if we are in the Area Basis, False if in Molar Basis.
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getAdsorbentName ()

  Returns the name of the adsorbent as a string.

Protected Attributes

- Matrix< double > activities_old

  List of the old activities calculated by the activity model.

Private Attributes

- std::vector< UnsteadyReaction > ads_rxn

  List of reactions involved with adsorption.

Additional Inherited Members

5.54.1 Detailed Description

Unsteady Adsorption Reaction Object.

C++ Object to handle data and functions associated with forumlating unsteady adsorption reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure.

5.54.2 Constructor & Destructor Documentation

5.54.2.1 UnsteadyAdsorption::UnsteadyAdsorption ( )

Default Constructor.

5.54.2.2 UnsteadyAdsorption::~UnsteadyAdsorption ( )

Default Destructor.

5.54.3 Member Function Documentation

5.54.3.1 void UnsteadyAdsorption::Initialize_Object (MasterSpeciesList & *List,* int *n*)

Function to call the initialization of objects sequentially.

5.54.3.2 void UnsteadyAdsorption::Display_Info ( )

Display the adsorption reaction information (PLACE HOLDER)

5.54.3.3 void UnsteadyAdsorption::modifyDeltas (MassBalance & *mbo*)

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.54.3.4 int UnsteadyAdsorption::setAdsorbIndices ( )

Find and set the adsorbed species indices for each reaction object.

This function searches through the Reaction objects in UnsteadyAdsorption to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.54.3.5 int UnsteadyAdsorption::checkAqueousIndices ( )

Function to check and report errors in the aqueous species indices.

5.54.3.6 void UnsteadyAdsorption::setActivityModelInfo (int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void *data) *act,* const void * *act_data*)

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.54.3.7 void UnsteadyAdsorption::setAqueousIndex (int *rxn_i,* int *species_i*)

Set the primary aqueous species index for the ith reaction.

5.54.3.8 int UnsteadyAdsorption::setAqueousIndexAuto ( )

Automatically sets the primary aqueous species index based on reactions.

This function will go through all species and all reactions in the adsorption object and automatically set the primary aqueous species index based on the stoicheometry of the reaction. It will also check and make sure that the primary aqueous index species appears opposite of the adsorbed species in the reactions. Note: This function assumes that the adsorbed indices have already been set.

5.54.3.9 void UnsteadyAdsorption::setActivityEnum (int *act*)

Set the surface activity enum value.

5.54.3.10 void UnsteadyAdsorption::setMolarFactor (int *rxn_i,* double *m*)

Set the molar factor for the ith reaction (mol/mol)

5.54.3.11 void UnsteadyAdsorption::setVolumeFactor (int *i,* double *v*)

Set the ith volume factor for the species list (cm^3/mol)

5.54.3.12 void UnsteadyAdsorption::setAreaFactor (int *i,* double *a*)

Set the ith area factor for the species list (m^2/mol)

5.54.3.13 void UnsteadyAdsorption::setSpecificArea (double *a*)

Set the specific area for the adsorbent (m^2/kg)

5.54.3.14 void UnsteadyAdsorption::setSpecificMolality (double *a*)

Set the specific molality for the adsorbent (mol/kg)

5.54.3.15 void UnsteadyAdsorption::setSurfaceCharge (double *c*)

Set the surface charge of the uncomplexed ligands.

5.54.3.16 void UnsteadyAdsorption::setTotalMass (double *m*)

Set the total mass of the adsorbent (kg)

5.54.3.17 void UnsteadyAdsorption::setTotalVolume (double *v*)

Set the total volume of the system (L)

5.54.3.18 void UnsteadyAdsorption::setAreaBasisBool (bool *opt*)

Set the basis boolean directly.

5.54.3.19 void UnsteadyAdsorption::setSurfaceChargeBool (bool *opt*)

Set the boolean for inclusion of surface charging.

5.54.3.20 void UnsteadyAdsorption::setBasis (std::string *option*)

Set the basis of the adsorption problem from the given string arg.

5.54.3.21 void UnsteadyAdsorption::setAdsorbentName (std::string *name*)

Set the name of the adsorbent to the given string.

5.54.3.22 void UnsteadyAdsorption::updateActivities ( )

Set the old activities as the new activities before doing next time step.

5.54.3.23 void UnsteadyAdsorption::calculateAreaFactors ( )

Calculates the area factors used from the van der Waals volumes.

5.54.3.24 void UnsteadyAdsorption::calculateEquilibria (double *T*)

Calculates all equilibrium parameters as a function of temperature.

5.54.3.25 void UnsteadyAdsorption::calculateRates (double *T*)

Calculates all reaction rate parameters as a function of temperature.

5.54.3.26 void UnsteadyAdsorption::setChargeDensity (const Matrix< double > & *x*)

Calculates and sets the current value of charge density.

5.54.3.27 void UnsteadyAdsorption::setIonicStrength (const Matrix< double > & *x*)

Calculates and sets the current value of ionic strength.

5.54.3.28 int UnsteadyAdsorption::callSurfaceActivity (const Matrix< double > & *x*)

Calls the activity model and returns an int flag for success or failure.

5.54.3.29 double UnsteadyAdsorption::calculateActiveFraction (const Matrix< double > & *x*)

Calculates the fraction of the surface that is active and available.

5.54.3.30 double UnsteadyAdsorption::calculateSurfaceChargeDensity (const Matrix< double > & *x*)

Function to calculate the surface charge density based on concentrations.

This function is used to calculate the surface charge density of the adsorbed species based on the charges and concentrations of the adsorbed species. The calculation is used to correct the adsorption equilibria constant based on a localized surface charge balance. This requires that you know the molality of the uncomplexed ligand species on the surface, as well as the specific surface area for the adsorbent.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.54.3.31 double UnsteadyAdsorption::calculatePsi (double *sigma,* double *T,* double *I,* double *rel_epsilon*)

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.54.3.32 double UnsteadyAdsorption::calculateAqueousChargeExchange (int *i*)

Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.

This function will look at all aqueous species involved in the ith adsorption reaction and sum up their stoicheometries and charges to see what the net change in charge is caused by the adsorption of charged species in solution. It is then used to adjust or correct the equilibrium constant for the given adsorption reaction.

Parameters

i	index of the reaction of interest for the adsorption object

5.54.3.33 double UnsteadyAdsorption::calculateEquilibriumCorrection (double *sigma,* double *T,* double *I,* double *rel_epsilon,* int *i*)

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
i	index of the reaction of interest for the adsorption object

5.54.3.34 double UnsteadyAdsorption::Eval_Residual (const Matrix< double > & *x,* const Matrix< double > & *gama,* double *T,* double *rel_perm,* int *i*)

Calculates the residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.54.3.35 double UnsteadyAdsorption::Eval_Residual (const Matrix< double > & *x_new,* const Matrix< double > & *x_old,* const Matrix< double > & *gama_new,* const Matrix< double > & *gama_old,* double *T,* double *rel_perm,* int *i*)

Calculates the unsteady residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Unsteady Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x_new	matrix of the current log(C) concentration values at the current non-linear step
gama_new	matrix of current activity coefficients for each species at the current non-linear step
x_old	matrix of the old log(C) concentration values at the current non-linear step
gama_old	matrix of old activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.54.3.36 double UnsteadyAdsorption::Eval_ReactionRate (const Matrix< double > & *x,* const Matrix< double > & *gama,* double *T,* double *rel_perm,* int *i*)

Function to calculate the explicit or implicit rate of reaction.

This function will calculate the rate/extent of the unsteady adsorption reaction given the log(C) concentrations and aqueous activities, as well as temperature and permittivity. The temperature and permittivity are used to make surface charge corrections to the equilibria and rate constants.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.54.3.37 double UnsteadyAdsorption::Eval_IC_Residual (const Matrix< double > & *x,* int *i*)

Calculate the unsteady residual for initial conditions.

Setting the intial conditions for all variables in the system requires a speciation calculation. However, we want the unsteady variables to be set to their respective initial conditions. Using this residual function imposes an equality constraint on those non-linear, unsteady variables allowing the rest of the speciation problem to be solved via PJFNK iterations.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
i	index of the reaction of interest for the adsorption object

5.54.3.38 double UnsteadyAdsorption::Explicit_Eval (const Matrix< double > & *x,* const Matrix< double > & *gama,* double *T,* double *rel_perm,* int *i*)

Return an approximate explicit solution to our unsteady adsorption variable (mol/kg)

This function will approximate the concentration of the unsteady variables based on an explicit time discretization. The purpose of this function is to try to provide the PJFNK method with a good initial guess for the values of the non-linear, unsteady variables. If we do not provide a good initial guess to these variables, then the PJFNK method may not converge to the correct solution, because the unsteady problem is the most difficult to solve.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.54.3.39 UnsteadyReaction& UnsteadyAdsorption::getReaction (int *i*)

Return reference to the ith reaction object in the adsorption object.

5.54.3.40 double UnsteadyAdsorption::getMolarFactor (int *i*)

Get the ith reaction's molar factor for adsorption (mol/mol)

5.54.3.41 double UnsteadyAdsorption::getVolumeFactor (int *i*)

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

5.54.3.42 double UnsteadyAdsorption::getAreaFactor (int *i*)

Get the ith area factor (species not involved return zeros) (m^2/mol)

5.54.3.43 double UnsteadyAdsorption::getActivity (int *i*)

Get the ith activity factor for the surface species.

5.54.3.44 double UnsteadyAdsorption::getOldActivity (int *i*)

Get the ith old activity factor for the surface species.

5.54.3.45 double UnsteadyAdsorption::getSpecificArea ( )

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.54.3.46 double UnsteadyAdsorption::getSpecificMolality ( )

Get the specific molality of the adsorbent (mol/kg)

5.54.3.47 double UnsteadyAdsorption::getSurfaceCharge ( )

Get the surface charge of the adsorbent.

5.54.3.48 double UnsteadyAdsorption::getBulkDensity ( )

Calculate and return bulk density of adsorbent in system (kg/L)

5.54.3.49 double UnsteadyAdsorption::getTotalMass ( )

Get the total mass of adsorbent in the system (kg)

5.54.3.50 double UnsteadyAdsorption::getTotalVolume ( )

Get the total volume of the system (L)

5.54.3.51 double UnsteadyAdsorption::getChargeDensity ( )

Get the value of the surface charge density (C/m^2)

5.54.3.52 double UnsteadyAdsorption::getIonicStrength ( )

Get the value of the ionic strength of solution (mol/L)

5.54.3.53 int UnsteadyAdsorption::getNumberRxns ( )

Get the number of reactions involved in the adsorption object.

5.54.3.54 int UnsteadyAdsorption::getAdsorbIndex (int *i*)

Get the index of the adsorbed species in the ith reaction.

5.54.3.55 int UnsteadyAdsorption::getAqueousIndex (int *i*)

Get the index of the primary aqueous species in the ith reaction.

5.54.3.56 int UnsteadyAdsorption::getActivityEnum ( )

Return the enum representing the choosen activity function.

5.54.3.57 bool UnsteadyAdsorption::isAreaBasis ( )

Returns true if we are in the Area Basis, False if in Molar Basis.

5.54.3.58 bool UnsteadyAdsorption::includeSurfaceCharge ( )

Returns true if we are considering surface charging during adsorption.

5.54.3.59 std::string UnsteadyAdsorption::getAdsorbentName ( )

Returns the name of the adsorbent as a string.

5.54.4 Member Data Documentation

5.54.4.1 Matrix<double> UnsteadyAdsorption::activities_old [protected]

List of the old activities calculated by the activity model.

5.54.4.2 std::vector<UnsteadyReaction> UnsteadyAdsorption::ads_rxn [private]

List of reactions involved with adsorption.

The documentation for this class was generated from the following file:

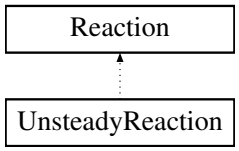
- shark.h

5.55 UnsteadyReaction Class Reference

Unsteady Reaction Object (inherits from Reaction)

`#include <shark.h>`

Inheritance diagram for UnsteadyReaction:



Public Member Functions

- UnsteadyReaction ()

  Default Constructor.
- ~UnsteadyReaction ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List)

  Function to initialize the UnsteadyReaction object from the MasterSpeciesList.
- void Display_Info ()

  Display the unsteady reaction information.
- void Set_Species_Index (int i)

  Set the Unsteady species index by number.
- void Set_Species_Index (std::string formula)

  Set the Unsteady species index by formula.
- void Set_Stoichiometric (int i, double v)

  Set the ith stoichiometric value (see Reaction object)
- void Set_Equilibrium (double v)

  Set the equilibrium constant (logK) (see Reaction object)
- void Set_Enthalpy (double H)

  Set the enthalpy of the reaction (J/mol) (see Reaction object)
- void Set_Entropy (double S)

  Set the entropy of the reaction (J/K/mol) (see Reaction object)
- void Set_EnthalpyANDEntropy (double H, double S)

  Set both the enthalpy and entropy (J/mol) & (J/K/mol) (see Reaction object)
- void Set_Energy (double G)

  Set the Gibb's free energy of reaction (J/mol) (see Reaction object)
- void Set_InitialValue (double ic)

  Set the initial value of the unsteady variable.
- void Set_MaximumValue (double max)

  Set the maximum value of the unsteady variable to a given value max (mol/L)
- void Set_Forward (double forward)

  Set the forward rate for the reaction (mol/L/hr)
- void Set_Reverse (double reverse)

  Set the reverse rate for the reaction (mol/L/hr)
- void Set_ForwardRef (double Fref)

  Set the forward reference rate (mol/L/hr)
- void Set_ReverseRef (double Rref)

Set the reverse reference rate (mol/L/hr)

- void Set_ActivationEnergy (double E)

  Set the activation energy for the reaction (J/mol)
- void Set_Affinity (double b)

  Set the temperature affinity parameter for the reaction.
- void Set_TimeStep (double dt)

  Set the time step for the current simulation.
- void checkSpeciesEnergies ()

  Function to check MasterSpeciesList for species energy info (see Reaction object)
- void calculateEnergies ()

  Function to calculate the energy of the reaction (see Reaction object)
- void calculateEquilibrium (double T)

  Function to calculate the equilibrium constant (see Reaction object)
- void calculateRate (double T)

  Function to calculate the rate constant based on given temperature.
- bool haveEquilibrium ()

  True if equilibrium constant is given or can be calculated (see Reaction object)
- bool haveRate ()

  Function to return true if you have the forward or reverse rate calculated.
- bool haveForwardRef ()

  Function to return true if you have the forward reference rate.
- bool haveReverseRef ()

  Function to return true if you have the reverse reference rate.
- bool haveForward ()

  Function to return true if you have the forward rate.
- bool haveReverse ()

  Function to return true if you have the reverse rate.
- int Get_Species_Index ()

  Fetch the index of the Unsteady species.
- double Get_Stoichiometric (int i)

  Fetch the ith stoichiometric value.
- double Get_Equilibrium ()

  Fetch the equilibrium constant (logK)
- double Get_Enthalpy ()

  Fetch the enthalpy of the reaction.
- double Get_Entropy ()

  Fetch the entropy of the reaction.
- double Get_Energy ()

  Fetch the energy of the reaction.
- double Get_InitialValue ()

  Fetch the initial value of the variable.
- double Get_MaximumValue ()

  Fetch the maximum value of the variable.
- double Get_Forward ()

  Fetch the forward rate.
- double Get_Reverse ()

  Fetch the reverse rate.
- double Get_ForwardRef ()

  Fetch the forward reference rate.
- double Get_ReverseRef ()

  Fetch the reverse reference rate.

- double Get_ActivationEnergy ()

  Fetch the activation energy for the reaction.
- double Get_Affinity ()

  Fetch the temperature affinity for the reaction.
- double Get_TimeStep ()

  Fetch the time step.
- double Eval_ReactionRate (const Matrix< double > &x, const Matrix< double > &gama)

  Calculate reation rate (dC/dt) from concentrations and activities.
- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old, const Matrix< double > &gama_new, const Matrix< double > &gama_old)

  Calculate the unsteady residual for the reaction using and implicit time discretization.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama)

  Calculate the steady-state residual for this reaction (see Reaction object)
- double Eval_IC_Residual (const Matrix< double > &x)

  Calculate the unsteady residual for initial conditions.
- double Explicit_Eval (const Matrix< double > &x, const Matrix< double > &gama)

  Return an approximate explicit solution to our unsteady variable (mol/L)

Protected Attributes

- double initial_value

  Initial value given at t=0 (in mol/L)
- double max_value

  Maximum value plausible (in mol/L)
- double forward_rate

  Forward reaction rate constant (in (mol/L)^ n/hr)
- double reverse_rate

  Reverse reaction rate constant (in (mol/L)^ n/hr)
- double forward_ref_rate

  Forward reference rate constant (in (mol/L)^ n/hr)
- double reverse_ref_rate

  Reverse reference rate constant (in (mol/L)^ n/hr)
- double activation_energy

  Activation or barrier energy for the reaction (J/mol)
- double temperature_affinity

  Temperature affinity parameter (dimensionless)
- double time_step

  Time step size for current step.
- bool HaveForward

  True if can calculate, or was given the forward rate.
- bool HaveReverse

  True if can calculate, or was given the reverse rate.
- bool HaveForRef

  True if given the forward reference rate.
- bool HaveRevRef

  True if given the reverse reference rate.
- int species_index

  Index in MasterList of Unsteady Species.

Additional Inherited Members

5.55.1 Detailed Description

Unsteady Reaction Object (inherits from Reaction)

C++ style object that holds data and functions associated with unsteady chemical reactions...
i.e., aA + bB <– reverse : forward –> cC + dD
This is essentially the same as the steady reaction, but we now have a forward and reverse reaction rate to deal with. It should be noted that this is a very simple kinetic reaction model based on splitting an overall equilibrium reaction into an overall forward and reverse reaction model. Therefore, it is not expected that this representation of the reaction will provide high accuracy results for reaction kinetics, but should at least provide an overall idea of the process occuring.

5.55.2 Constructor & Destructor Documentation

5.55.2.1 UnsteadyReaction::UnsteadyReaction ( )

Default Constructor.

5.55.2.2 UnsteadyReaction::~UnsteadyReaction ( )

Default Destructor.

5.55.3 Member Function Documentation

5.55.3.1 void UnsteadyReaction::Initialize_Object (MasterSpeciesList & *List*)

Function to initialize the UnsteadyReaction object from the MasterSpeciesList.

5.55.3.2 void UnsteadyReaction::Display_Info ( )

Display the unsteady reaction information.

5.55.3.3 void UnsteadyReaction::Set_Species_Index (int *i*)

Set the Unsteady species index by number.

This function will set the unsteady species index by the index i given. That given index must correspond to the index of the species in the MasterSpeciesList that is being considered as the unsteady species.

Parameters

i	index of the unsteady species in the MasterSpeciesList

5.55.3.4 void UnsteadyReaction::Set_Species_Index (std::string *formula*)

Set the Unsteady species index by formula.

This function will check the MasterSpeciesList for the molecule object that has the given formula, then set the unsteady species index based on the index of that species in the master list.

Parameters

formula	molecular formula of the unsteady species (see mola.h for standard formatting)

5.55.3.5 void UnsteadyReaction::Set_Stoichiometric (int *i,* double *v*)

Set the ith stoichiometric value (see Reaction object)

5.55.3.6 void UnsteadyReaction::Set_Equilibrium (double *v*)

Set the equilibrium constant (logK) (see Reaction object)

5.55.3.7 void UnsteadyReaction::Set_Enthalpy (double *H*)

Set the enthalpy of the reaction (J/mol) (see Reaction object)

5.55.3.8 void UnsteadyReaction::Set_Entropy (double *S*)

Set the entropy of the reaction (J/K/mol) (see Reaction object)

5.55.3.9 void UnsteadyReaction::Set_EnthalpyANDEntropy (double *H,* double *S*)

Set both the enthalpy and entropy (J/mol) & (J/K/mol) (see Reaction object)

5.55.3.10 void UnsteadyReaction::Set_Energy (double *G*)

Set the Gibb's free energy of reaction (J/mol) (see Reaction object)

5.55.3.11 void UnsteadyReaction::Set_InitialValue (double *ic*)

Set the initial value of the unsteady variable.

This function sets the initial concentration value for the unsteady species to the given value ic (mol/L). Only unsteady species need to be given an initial value. All other species initial values for the overall system is setup based on a speciation calculation performed while holding the unsteady variables constant at their respective initial values.

Parameters

ic	initial concentration value for the unsteady object (mol/L)

5.55.3.12 void UnsteadyReaction::Set_MaximumValue (double *max*)

Set the maximum value of the unsteady variable to a given value max (mol/L)

This function will be called internally to help bound the unsteady variable to reasonable maximum values. That maximum is usually based on the mass balances for the current non-linear iteration.

Parameters

max	maximum allowable value for the unsteady variable (mol/L)

5.55.3.13 void UnsteadyReaction::Set_Forward (double *forward*)

Set the forward rate for the reaction (mol/L/hr)

5.55.3.14 void UnsteadyReaction::Set_Reverse (double *reverse*)

Set the reverse rate for the reaction (mol/L/hr)

5.55.3.15 void UnsteadyReaction::Set_ForwardRef (double *Fref*)

Set the forward reference rate (mol/L/hr)

Unlike just setting the forward rate, this function sets a reference forward rate of the reaction that can be used to correct the overall forward rate based on system temperature and Arrhenius Rate Equation constants.

Parameters

Fref	forward reference rate constant (mol/L/hr)

5.55.3.16 void UnsteadyReaction::Set_ReverseRef (double *Rref*)

Set the reverse reference rate (mol/L/hr)

Unlike just setting the reverse rate, this function sets a reference reverse rate of the reaction that can be used to correct the overall reverse rate based on system temperature and Arrhenius Rate Equation constants.

Parameters

Rref	reverse reference rate constant (mol/L/hr)

5.55.3.17 void UnsteadyReaction::Set_ActivationEnergy (double *E*)

Set the activation energy for the reaction (J/mol)

This function will set the activation energy for the reaction to the given value of E. Note that we will only set one value for activation energy, even though there are rates for forward and reverse reactions. This is because we use the ratio of the rates and the equilibrium constant to establish the other rate. Therefore, we only need either the forward or reverse rate and the equilibrium constant to set all the rates.

Parameters

E	activation energy for the forward or reverse rate, depending on which was given

5.55.3.18 void UnsteadyReaction::Set_Affinity (double *b*)

Set the temperature affinity parameter for the reaction.

This function will set the temperature affinity for the reaction to the given value of b. Note that we will only set one value for temperature affinity, even though there are rates for forward and reverse reactions. This is because we use the ratio of the rates and the equilibrium constant to establish the other rate. Therefore, we only need either the forward or reverse rate and the equilibrium constant to set all the rates.

Parameters

b	temperature affinity for the forward or reverse rate, depending on which was given

5.55.3.19 void UnsteadyReaction::Set_TimeStep (double *dt*)

Set the time step for the current simulation.

5.55.3.20 void UnsteadyReaction::checkSpeciesEnergies ( )

Function to check MasterSpeciesList for species energy info (see Reaction object)

5.55.3.21 void UnsteadyReaction::calculateEnergies ( )

Function to calculate the energy of the reaction (see Reaction object)

5.55.3.22 void UnsteadyReaction::calculateEquilibrium (double *T*)

Function to calculate the equilibrium constant (see Reaction object)

5.55.3.23 void UnsteadyReaction::calculateRate (double *T*)

Function to calculate the rate constant based on given temperature.

This function will calculate and set either the forward or reverse rate for the unsteady reaction based on what information was given. If the forward rate information was given, then it sets the reverse rate and visa versa. If nothing was set correctly, an error will occur.

Parameters

T	temperature of the system in Kelvin

5.55.3.24 bool UnsteadyReaction::haveEquilibrium ( )

True if equilibrium constant is given or can be calculated (see Reaction object)

5.55.3.25 bool UnsteadyReaction::haveRate ( )

Function to return true if you have the forward or reverse rate calculated.

5.55.3.26 bool UnsteadyReaction::haveForwardRef ( )

Function to return true if you have the forward reference rate.

5.55.3.27 bool UnsteadyReaction::haveReverseRef ( )

Function to return true if you have the reverse reference rate.

5.55.3.28 bool UnsteadyReaction::haveForward ( )

Function to return true if you have the forward rate.

5.55.3.29 bool UnsteadyReaction::haveReverse ( )

Function to return true if you have the reverse rate.

5.55.3.30 int UnsteadyReaction::Get_Species_Index ( )

Fetch the index of the Unsteady species.

5.55.3.31 double UnsteadyReaction::Get_Stoichiometric (int *i*)

Fetch the ith stoichiometric value.

5.55.3.32 double UnsteadyReaction::Get_Equilibrium ( )

Fetch the equilibrium constant (logK)

5.55.3.33 double UnsteadyReaction::Get_Enthalpy ( )

Fetch the enthalpy of the reaction.

5.55.3.34 double UnsteadyReaction::Get_Entropy ( )

Fetch the entropy of the reaction.

5.55.3.35 double UnsteadyReaction::Get_Energy ( )

Fetch the energy of the reaction.

5.55.3.36 double UnsteadyReaction::Get_InitialValue ( )

Fetch the initial value of the variable.

5.55.3.37 double UnsteadyReaction::Get_MaximumValue ( )

Fetch the maximum value of the variable.

5.55.3.38 double UnsteadyReaction::Get_Forward ( )

Fetch the forward rate.

5.55.3.39 double UnsteadyReaction::Get_Reverse ( )

Fetch the reverse rate.

5.55.3.40 double UnsteadyReaction::Get_ForwardRef ( )

Fetch the forward reference rate.

5.55.3.41 double UnsteadyReaction::Get_ReverseRef ( )

Fetch the reverse reference rate.

5.55.3.42 double UnsteadyReaction::Get_ActivationEnergy ( )

Fetch the activation energy for the reaction.

5.55.3.43 double UnsteadyReaction::Get_Affinity ( )

Fetch the temperature affinity for the reaction.

5.55.3.44 double UnsteadyReaction::Get_TimeStep ( )

Fetch the time step.

5.55.3.45 double UnsteadyReaction::Eval_ReactionRate (const Matrix< double > & *x,* const Matrix< double > & *gama*)

Calculate reation rate (dC/dt) from concentrations and activities.

This function calculates the right hand side of the unsteady reaction equation based on the available rates, the current values of the non-linear variables (x=log(C)), and the activity coefficients (gama).

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step

5.55.3.46 double UnsteadyReaction::Eval_Residual (const Matrix< double > & *x_new,* const Matrix< double > & *x_old,* const Matrix< double > & *gama_new,* const Matrix< double > & *gama_old*)

Calculate the unsteady residual for the reaction using and implicit time discretization.

This function uses the current time step and states of the non-linear variables and activities to form the residual contribution of the unsteady reaction. The time dependent functions are discretized using an implicit finite difference for best stability.

Parameters

x_new	matrix of the log(C) concentration values at the current non-linear step
gama_new	matrix of activity coefficients for each species at the current non-linear step
x_old	matrix of the log(C) concentration values at the previous non-linear step
gama_old	matrix of activity coefficients for each species at the previous non-linear step

5.55.3.47 double UnsteadyReaction::Eval_Residual (const Matrix< double > & *x*, const Matrix< double > & *gama*)

Calculate the steady-state residual for this reaction (see Reaction object)

5.55.3.48 double UnsteadyReaction::Eval_IC_Residual (const Matrix< double > & *x*)

Calculate the unsteady residual for initial conditions.

Setting the intial conditions for all variables in the system requires a speciation calculation. However, we want the unsteady variables to be set to their respective initial conditions. Using this residual function imposes an equality constraint on those non-linear, unsteady variables allowing the rest of the speciation problem to be solved via PJFNK iterations.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.55.3.49 double UnsteadyReaction::Explicit_Eval (const Matrix< double > & *x*, const Matrix< double > & *gama*)

Return an approximate explicit solution to our unsteady variable (mol/L)

This function will approximate the concentration of the unsteady variables based on an explicit time discretization. The purpose of this function is to try to provide the PJFNK method with a good initial guess for the values of the non-linear, unsteady variables. If we do not provide a good initial guess to these variables, then the PJFNK method may not converge to the correct solution, because the unsteady problem is the most difficult to solve.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step

5.55.4 Member Data Documentation

5.55.4.1 double UnsteadyReaction::initial_value [protected]

Initial value given at t=0 (in mol/L)

5.55.4.2 double UnsteadyReaction::max_value [protected]

Maximum value plausible (in mol/L)

5.55.4.3 double UnsteadyReaction::forward_rate [protected]

Forward reaction rate constant (in (mol/L)^n/hr)

5.55.4.4 double UnsteadyReaction::reverse_rate [protected]

Reverse reaction rate constant (in (mol/L)^n/hr)

5.55.4.5 double UnsteadyReaction::forward_ref_rate [protected]

Forward reference rate constant (in (mol/L)^n/hr)

5.55.4.6 double UnsteadyReaction::reverse_ref_rate [protected]

Reverse reference rate constant (in (mol/L)^n/hr)

5.55.4.7 double UnsteadyReaction::activation_energy [protected]

Activation or barrier energy for the reaction (J/mol)

5.55.4.8 double UnsteadyReaction::temperature_affinity [protected]

Temperature affinity parameter (dimensionless)

5.55.4.9 double UnsteadyReaction::time_step [protected]

Time step size for current step.

5.55.4.10 bool UnsteadyReaction::HaveForward [protected]

True if can calculate, or was given the forward rate.

5.55.4.11 bool UnsteadyReaction::HaveReverse [protected]

True if can calculate, or was given the reverse rate.

5.55.4.12 bool UnsteadyReaction::HaveForRef [protected]

True if given the forward reference rate.

5.55.4.13 bool UnsteadyReaction::HaveRevRef [protected]

True if given the reverse reference rate.

5.55.4.14 int UnsteadyReaction::species_index [protected]

Index in MasterList of Unsteady Species.

The documentation for this class was generated from the following file:

- shark.h

5.56 ValueTypePair Class Reference

Value-Type Pair object to recognize data type of a string that was read.

`#include <yaml_wrapper.h>`

Public Member Functions

- ValueTypePair ()
  Default constructor.
- ~ValueTypePair ()
  Default destructor.
- ValueTypePair (const std::pair< std::string, int > &vt)
  Constructor by pair.
- ValueTypePair (std::string value, int type)
  Construction by string and int.
- ValueTypePair (const ValueTypePair &vt)
  Copy constructor.
- ValueTypePair & operator= (const ValueTypePair &vt)
  Equals operator overload.
- void editValue (std::string value)
  Edits value to pair with UNKOWN type.
- void editPair (std::string value, int type)
  Creates a paired Value-Type from the given args.
- void findType ()
  Determines the data type of the object.
- void assertType (int type)
  Forces a specific data type.
- void DisplayPair ()
  Display the pair information.
- std::string getString ()
  Returns the value of the pair as a string.
- bool getBool ()
  Returns the value of the pair as a bool.
- double getDouble ()
  Returns the value of the pair as a double.
- int getInt ()
  Returns the value of the pair as an int.
- std::string getValue ()
  Returns the value of the pair as it was given.
- int getType ()
  Returns the type of the pair.
- std::pair< std::string, int > & getPair ()
  Returns reference to the actual pair object.

Private Attributes

- std::pair< std::string, int > Value_Type
  pair object holding the Value and Type info
- int type
  Type of the value.

5.56.1 Detailed Description

Value-Type Pair object to recognize data type of a string that was read.

C++ Object that creates a pair between a read in value as a string and an enum denoting what the data type of that string is. This object is primarily used in the other yaml_wrapper objects, but can also be used for any string that you want to parse to identify it's type. The supported types are denoted in the data_type enum and can be determined automatically by the findType() function or can be specified by the assertType() function.

5.56.2 Constructor & Destructor Documentation

5.56.2.1 ValueTypePair::ValueTypePair ( )

Default constructor.

5.56.2.2 ValueTypePair::~ValueTypePair ( )

Default destructor.

5.56.2.3 ValueTypePair::ValueTypePair (const std::pair< std::string, int > & *vt*)

Constructor by pair.

5.56.2.4 ValueTypePair::ValueTypePair (std::string *value,* int *type*)

Construction by string and int.

5.56.2.5 ValueTypePair::ValueTypePair (const ValueTypePair & *vt*)

Copy constructor.

5.56.3 Member Function Documentation

5.56.3.1 ValueTypePair& ValueTypePair::operator= (const ValueTypePair & *vt*)

Equals operator overload.

5.56.3.2 void ValueTypePair::editValue (std::string *value*)

Edits value to pair with UNKOWN type.

5.56.3.3 void ValueTypePair::editPair (std::string *value,* int *type*)

Creates a paired Value-Type from the given args.

5.56.3.4 void ValueTypePair::findType ( )

Determines the data type of the object.

5.56.3.5 void ValueTypePair::assertType (int *type*)

Forces a specific data type.

5.56.3.6 void ValueTypePair::DisplayPair ( )

Display the pair information.

5.56.3.7 std::string ValueTypePair::getString ( )

Returns the value of the pair as a string.

5.56.3.8 bool ValueTypePair::getBool ( )

Returns the value of the pair as a bool.

5.56.3.9 double ValueTypePair::getDouble ( )

Returns the value of the pair as a double.

5.56.3.10 int ValueTypePair::getInt ( )

Returns the value of the pair as an int.

5.56.3.11 std::string ValueTypePair::getValue ( )

Returns the value of the pair as it was given.

5.56.3.12 int ValueTypePair::getType ( )

Returns the type of the pair.

5.56.3.13 std::pair<std::string,int>& ValueTypePair::getPair ( )

Returns reference to the actual pair object.

5.56.4 Member Data Documentation

5.56.4.1 std::pair<std::string,int> ValueTypePair::Value_Type [private]

pair object holding the Value and Type info

5.56.4.2 int ValueTypePair::type [private]

Type of the value.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.57 yaml_cpp_class Class Reference

Primary object used when reading and digitally storing yaml files.

`#include <yaml_wrapper.h>`

Public Member Functions

- yaml_cpp_class ()
  Default constructor.
- ~yaml_cpp_class ()
  Default destructor.
- int setInputFile (const char *file)
  Set the input file to be read.
- int readInputFile ()
  Reads through input file and stores into YamlWrapper.
- int cleanup ()
  Deletes yaml_c objects and closes the input file.
- int executeYamlRead (const char *file)
  Runs the full execution of initialization, reading, and cleaning.
- YamlWrapper & getYamlWrapper ()
  Returns reference to the YamlWrapper Object.
- void DisplayContents ()
  Print out the contents of the read to the console window.

Private Attributes

- YamlWrapper yaml_wrapper
  YamlWrapper object where digital file is stored.
- FILE * input_file
  Function pointer to the yaml formatted file.
- const char * file_name
  Name of the file to be parsed and read.
- yaml_parser_t token_parser
  C-YAML parser object for token based parsing.
- yaml_token_t current_token
  C-YAML token object for the current token in the file.
- yaml_token_t previous_token
  C-YAML token object for the previous token in the file.

5.57.1 Detailed Description

Primary object used when reading and digitally storing yaml files.

C++ Object that holds the YamlWrapper object and the C-YAML objects necessary for reading and parsing a yaml formatted file. This is the primary object that users are expected to work with when using yaml_wrapper.h to read input files. It contains functions necessary to setup a read instance, read and parse the input, place the parsed input results into the digital YamlWrapper object, and allow the user to query that object.

The two main functions that the typical user will need are: (i) executeYamlRead() and (ii) getYamlWrapper. Make sure that the read function was called prior to querying the YamlWrapper structure. Do not call the cleanup() function if using executeYamlRead(). That function will be automattically called after the read is complete.

5.57.2 Constructor & Destructor Documentation

5.57.2.1 yaml_cpp_class::yaml_cpp_class ( )

Default constructor.

5.57.2.2 yaml_cpp_class::~yaml_cpp_class ( )

Default destructor.

5.57.3 Member Function Documentation

5.57.3.1 int yaml_cpp_class::setInputFile (const char * *file*)

Set the input file to be read.

5.57.3.2 int yaml_cpp_class::readInputFile ( )

Reads through input file and stores into YamlWrapper.

5.57.3.3 int yaml_cpp_class::cleanup ( )

Deletes yaml_c objects and closes the input file.

5.57.3.4 int yaml_cpp_class::executeYamlRead (const char * *file*)

Runs the full execution of initialization, reading, and cleaning.

5.57.3.5 YamlWrapper& yaml_cpp_class::getYamlWrapper ( )

Returns reference to the YamlWrapper Object.

5.57.3.6 void yaml_cpp_class::DisplayContents ( )

Print out the contents of the read to the console window.

5.57.4 Member Data Documentation

5.57.4.1 YamlWrapper yaml_cpp_class::yaml_wrapper [private]

YamlWrapper object where digital file is stored.

5.57.4.2 FILE* yaml_cpp_class::input_file [private]

Function pointer to the yaml formatted file.

5.57.4.3 const char* yaml_cpp_class::file_name [private]

Name of the file to be parsed and read.

5.57.4.4 yaml_parser_t yaml_cpp_class::token_parser [private]

C-YAML parser object for token based parsing.

5.57.4.5 yaml_token_t yaml_cpp_class::current_token [private]

C-YAML token object for the current token in the file.

5.57.4.6 yaml_token_t yaml_cpp_class::previous_token [private]

C-YAML token object for the previous token in the file.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.58 YamlWrapper Class Reference

Object for the entire yaml file holding all documents, header, sub-headers, keys, and values.

`#include <yaml_wrapper.h>`

Public Member Functions

- YamlWrapper ()
  Default constructor.
- ~YamlWrapper ()
  Default destructor.
- YamlWrapper (const YamlWrapper &yaml)
  Copy constructor.
- YamlWrapper (std::string key, const Document &doc)
  Constructor by a single document.
- YamlWrapper & operator= (const YamlWrapper &yaml)
  Equals overload.
- Document & operator() (const std::string key)
  Return the Document reference at the given key.
- Document operator() (const std::string key) const
  Return the Document at the given key.
- std::map< std::string, Document > & getDocMap ()
  Return reference to the document map.
- Document & getDocument (std::string key)
  Return reference to the document at the key.
- std::map< std::string, Document >::const_iterator end () const
  Returns a const iterator pointing to the end of the list.
- std::map< std::string, Document >::iterator end ()

Returns an iterator pointing to the end of the list.

- std::map< std::string, Document >::const_iterator begin () const

  Returns a const iterator pointing to the begining of the list.

- std::map< std::string, Document >::iterator begin ()

  Returns an iterator pointing to the begining of the list.

- void clear ()

  Clear out the yaml object.

- void resetKeys ()

  Resets all the keys in DocumentMap to match document names.

- void changeKey (std::string oldKey, std::string newKey)

  Change a given oldKey in the map to the newKey given.

- void revalidateAllKeys ()

  Resets and validates all keys in the structure.

- void DisplayContents ()

  Display the contents of the wrapper.

- void addDocKey (std::string key)

  Add a key to the document map.

- void copyAnchor2Alias (std::string alias, Document &ref)

  Find the anchor in the map, and copy to the Document reference given.

- int size ()

  Return the size of the document map.

- Document & getAnchoredDoc (std::string alias)

  Return the reference to the document that is anchored with the given alias.

- Document & getDocFromHeadAlias (std::string alias)

  Return reference to the document that contains the header with the given alias.

- Document & getDocFromSubAlias (std::string alias)

  Return reference to the document that contains the subheader with the given alias.

Private Attributes

- std::map< std::string, Document > Doc_Map

  Map of the documents contained within the wrapper.

5.58.1 Detailed Description

Object for the entire yaml file holding all documents, header, sub-headers, keys, and values.

C++ Object for the yaml file. This object holds a map of all Documents in the yaml file. Each document holds a map of Key-values and Headers. The Headers hold maps of Key-values and SubHeaders, and each SubHeader can hold more Key-values.

This object is used to represent a digital and queryable structure for all information contained within a yaml file. There are some limitations to what can be held here, and those limitations are based on the limitations in the C-YAML Library token parser. The main limitation is that the deepest level of allowable recursion in the file is SubHeader. Meaning that you are not allowed to have Sub-SubHeaders underneath a SubHeader object. This imposes a hard limit to number of nested lists that can be in a single Document object.

When using yaml_cpp_class, this object will generally be what you query into to get the information from your yaml input files. From this object, functions and operators are provided to give you the capability of querying down into any allowable level of the file by the keys that are were used in the file. Be sure that you are querying the correct objects by the correct keys, otherwise errors and exceptions will be thrown.

5.58.2 Constructor & Destructor Documentation

5.58.2.1 YamlWrapper::YamlWrapper ( )

Default constructor.

5.58.2.2 YamlWrapper::~YamlWrapper ( )

Default destructor.

5.58.2.3 YamlWrapper::YamlWrapper (const YamlWrapper & *yaml*)

Copy constructor.

5.58.2.4 YamlWrapper::YamlWrapper (std::string *key,* const Document & *doc*)

Constructor by a single document.

5.58.3 Member Function Documentation

5.58.3.1 YamlWrapper& YamlWrapper::operator= (const YamlWrapper & *yaml*)

Equals overload.

5.58.3.2 Document& YamlWrapper::operator() (const std::string *key*)

Return the Document reference at the given key.

5.58.3.3 Document YamlWrapper::operator() (const std::string *key*) const

Return the Document at the given key.

5.58.3.4 std::map<std::string, Document>& YamlWrapper::getDocMap ( )

Return reference to the document map.

5.58.3.5 Document& YamlWrapper::getDocument (std::string *key*)

Return reference to the document at the key.

5.58.3.6 std::map<std::string, Document>::const_iterator YamlWrapper::end ( ) const

Returns a const iterator pointing to the end of the list.

5.58.3.7 std::map<std::string, Document>::iterator YamlWrapper::end ( )

Returns an iterator pointing to the end of the list.

5.58.3.8 std::map<std::string, Document>::const_iterator YamlWrapper::begin ( ) const

Returns a const iterator pointing to the begining of the list.

5.58.3.9 std::map<std::string, Document>::iterator YamlWrapper::begin ( )

Returns an iterator pointing to the begining of the list.

5.58.3.10 void YamlWrapper::clear ( )

Clear out the yaml object.

5.58.3.11 void YamlWrapper::resetKeys ( )

Resets all the keys in DocumentMap to match document names.

5.58.3.12 void YamlWrapper::changeKey (std::string *oldKey,* std::string *newKey*)

Change a given oldKey in the map to the newKey given.

5.58.3.13 void YamlWrapper::revalidateAllKeys ( )

Resets and validates all keys in the structure.

5.58.3.14 void YamlWrapper::DisplayContents ( )

Display the contents of the wrapper.

5.58.3.15 void YamlWrapper::addDocKey (std::string *key*)

Add a key to the document map.

5.58.3.16 void YamlWrapper::copyAnchor2Alias (std::string *alias,* Document & *ref*)

Find the anchor in the map, and copy to the Document reference given.

5.58.3.17 int YamlWrapper::size ( )

Return the size of the document map.

5.58.3.18 Document& YamlWrapper::getAnchoredDoc (std::string *alias*)

Return the reference to the document that is anchored with the given alias.

5.58.3.19 Document& YamlWrapper::getDocFromHeadAlias (std::string *alias*)

Return reference to the document that contains the header with the given alias.

5.58.3.20 Document& YamlWrapper::getDocFromSubAlias (std::string *alias*)

Return reference to the document that contains the subheader with the given alias.

5.58.4 Member Data Documentation

5.58.4.1 std::map<std::string, Document> YamlWrapper::Doc_Map [private]

Map of the documents contained within the wrapper.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

6 File Documentation

6.1 dogfish.h File Reference

Diffusion Object Governing Fiber Interior Sorption History.

```
#include "finch.h"
#include "mola.h"
```

Classes

- struct DOGFISH_PARAM
  Data structure for species-specific parameters.
- struct DOGFISH_DATA
  Primary data structure for running the DOGFISH application.

Functions

- void print2file_species_header (FILE *Output, DOGFISH_DATA *dog_dat, int i)
  Function to print a species based header for the output file.
- void print2file_DOGFISH_header (DOGFISH_DATA *dog_dat)
  Function to print a time and space header for the output file.
- void print2file_DOGFISH_result_old (DOGFISH_DATA *dog_dat)
  Function to print out the old time results for the output file.
- void print2file_DOGFISH_result_new (DOGFISH_DATA *dog_dat)
  Function to print out the new time results for the output file.
- double default_Retardation (int i, int l, const void *data)
  Default function for the retardation coefficient.
- double default_IntraDiffusion (int i, int l, const void *data)
  Default function for the intraparticle diffusion coefficient.
- double default_FilmMTCoeff (int i, const void *data)
  Default function for the film mass transfer coefficient.

- double default_SurfaceConcentration (int i, const void *data)

  Default function for the fiber surface concentration.
- int setup_DOGFISH_DATA (FILE *file, double(*eval_R)(int i, int l, const void *user_data), double(*eval_DI)(int i, int l, const void *user_data), double(*eval_kf)(int i, const void *user_data), double(*eval_qs)(int i, const void *user_data), const void *user_data, DOGFISH_DATA *dog_dat)

  Function will set up the memory and pointers for use in the DOGFISH simulations.
- int DOGFISH_Executioner (DOGFISH_DATA *dog_dat)

  Function to serially call all other functions need to solve the system at one time step.
- int set_DOGFISH_ICs (DOGFISH_DATA *dog_dat)

  Function called to evaluate the initial conditions for the time dependent problem.
- int set_DOGFISH_timestep (DOGFISH_DATA *dog_dat)

  Function sets the time step size for the next step forward in the simulation.
- int DOGFISH_preprocesses (DOGFISH_DATA *dog_dat)

  Function to perform preprocess actions to be used before calling any solver.
- int set_DOGFISH_params (const void *user_data)

  Function to calculate the values of all parameters for all species at all nodes.
- int DOGFISH_postprocesses (DOGFISH_DATA *dog_dat)

  Function to perform post-solve actions such as printing out results.
- int DOGFISH_reset (DOGFISH_DATA *dog_dat)

  Function to reset the matrices and vectors and prepare for next time step.
- int DOGFISH (DOGFISH_DATA *dog_dat)

  Function performs all necessary steps to step the diffusion simulation through time.
- int DOGFISH_TESTS ()

  Running DOGFISH tests.

6.1.1 Detailed Description

Diffusion Object Governing Fiber Interior Sorption History.

dogfish.cpp

This set of objects and functions is used to numerically solve linear or non-linear diffusion physics of aqueous ions into cylindrical adsorbent fibers. Boundary conditions for this problem could be a film mass transfer, reaction, or dirichlet condition depending on the type of problem being solve.

Warning

Functions and methods in this file are still under construction.

Author

Austin Ladshaw

Date

04/09/2015

Copyright

6.1.2 Function Documentation

6.1.2.1 void print2file_species_header (FILE * *Output,* DOGFISH_DATA * *dog_dat,* int *i*)

Function to print a species based header for the output file.

6.1.2.2 void print2file_DOGFISH_header (DOGFISH_DATA * *dog_dat*)

Function to print a time and space header for the output file.

6.1.2.3 void print2file_DOGFISH_result_old (DOGFISH_DATA * *dog_dat*)

Function to print out the old time results for the output file.

6.1.2.4 void print2file_DOGFISH_result_new (DOGFISH_DATA * *dog_dat*)

Function to print out the new time results for the output file.

6.1.2.5 double default_Retardation (int *i,* int *l,* const void * *data*)

Default function for the retardation coefficient.

The default retardation coefficient for this problem is 1.0 for all time and space. Therefore, this function will only ever return a 1.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
data	pointer to the DOGFISH_DATA structure

6.1.2.6 double default_IntraDiffusion (int *i,* int *l,* const void * *data*)

Default function for the intraparticle diffusion coefficient.

The default intraparticle diffusivity is to assume that each species i has a constant diffusivity. Therefore, this function returns the value of the parameter intraparticle_diffusion from the DOGFISH_PARAM structure for each adsorbing species i. Each species may have a different diffusivity.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
data	pointer to the DOGFISH_DATA structure

6.1.2.7 double default_FilmMTCoeff (int *i,* const void * *data*)

Default function for the film mass transfer coefficient.

The default film mass transfer coefficient will be to assume that this value is a constant for each species i. Therefore, this function returns the parameter value of film_transfer_coeff from the DOGFISH_PARAM structure for each adsorbing species i.

Parameters

i	index for the ith adsorbing species
data	pointer to the DOGFISH_DATA structure

6.1.2.8 double default_SurfaceConcentration (int *i*, const void * *data*)

Default function for the fiber surface concentration.

The default fiber surface concentration will be to assume that this value is a constant for each species i. Therefore, this function returns the parameter value of surface_concentration from the DOGFISH_PARAM structure for each adsorbing species i.

Parameters

i	index for the ith adsorbing species
data	pointer to the DOGFISH_DATA structure

6.1.2.9 int setup_DOGFISH_DATA (FILE * *file*, double(*)(int i, int l, const void *user_data) *eval_R*, double(*)(int i, int l, const void *user_data) *eval_DI*, double(*)(int i, const void *user_data) *eval_kf*, double(*)(int i, const void *user_data) *eval_qs*, const void * *user_data*, DOGFISH_DATA * *dog_dat*)

Function will set up the memory and pointers for use in the DOGFISH simulations.

The pointers to the output file, parameter functions, and data structures are passed into this function to setup the problem in memory. This function must always be called prior to calling any other DOGFISH routine and after the DOGFISH_DATA structure has been initialized.

Parameters

file	pointer to the output file to print out results
eval_R	function pointer for the retardation coefficient function
eval_DI	function pointer for the intraparticle diffusion function
eval_kf	function pointer for the film mass transfer function
eval_qs	function pointer for the surface concentration function
user_data	pointer for the user's own data structure (only if using custom functions)
dog_dat	pointer for the DOGFISH_DATA structure

6.1.2.10 int DOGFISH_Executioner (DOGFISH_DATA * *dog_dat*)

Function to serially call all other functions need to solve the system at one time step.

This function will call the DOGFISH_preprocesses function, followed by the FINCH solver functions for each species i, then call the DOGFISH_postprocesses function. After completion, this would have solved the diffusion physics for a single time step.

6.1.2.11 int set_DOGFISH_ICs (DOGFISH_DATA * *dog_dat*)

Function called to evaluate the initial conditions for the time dependent problem.

This function will use information in DOGFISH_DATA to setup the initial conditions, initial parameter values, and initial sorption averages for each species. This function always assumes a constant initial condition for the sorption of each species.

6.1.2.12 int set_DOGFISH_timestep (DOGFISH_DATA * *dog_dat*)

Function sets the time step size for the next step forward in the simulation.

This function will set the next time step size based on the spatial discretization of the fiber. Maximum time step size is locked at 0.5 hours.

6.1.2.13 int DOGFISH_preprocesses (DOGFISH_DATA * *dog_dat*)

Function to perform preprocess actions to be used before calling any solver.

This function will call all of the parameter functions in order to establish boundary condition parameter values prior to calling the FINCH solvers.

6.1.2.14 int set_DOGFISH_params (const void * *user_data*)

Function to calculate the values of all parameters for all species at all nodes.

This function is passed to the FINCH_DATA data structure and set as the setparams function pointer. FINCH calls this function during it's solver routine to setup the non-linear form of the problem and solve the non-linear system.

Parameters

user_data	this is actually the DOGFISH_DATA structure, but is passed anonymously to FINCH

6.1.2.15 int DOGFISH_postprocesses (DOGFISH_DATA * *dog_dat*)

Function to perform post-solve actions such as printing out results.

This function increments the total_steps counter in DOGFISH_DATA to keep a running total of all solver steps taken. Additionally, it prints out the results of the current time simulation to the output file.

6.1.2.16 int DOGFISH_reset (DOGFISH_DATA * *dog_dat*)

Function to reset the matrices and vectors and prepare for next time step.

This function will reset the matrix and vector information of DOGFISH_DATA and FINCH_DATA to prepare for the next simulation step in time.

6.1.2.17 int DOGFISH (DOGFISH_DATA * *dog_dat*)

Function performs all necessary steps to step the diffusion simulation through time.

This function calls the initial conditions, set time step, executioner, and reset functions to step the simulation through time. It will only exit when the simulation time is reached or if an error occurs.

6.1.2.18 int DOGFISH_TESTS ( )

Running DOGFISH tests.

This function is called from the UI to run a test simulation of DOGFISH. Ouput is stored in a DOGFISH_Test↩ Output.txt file in a sub-directory "output" from the directory in which the executable was called.

6.2 dove.h File Reference

Dynamic ODE solver with Various Established methods.

```
#include "macaw.h"
#include "lark.h"
#include "yaml_wrapper.h"
```

Classes

- class Dove

  Dynamic ODE-solver with Various Established methods (DOVE) object.

Macros

- #define DOVE_HPP_

Enumerations

- enum integrate_type { IMPLICIT, EXPLICIT }

  Enumeration for the list of valid time integration types.
- enum integrate_subtype {
  BE, FE, CN, BDF2,
  RK4, RKF }

  Enumeration for the list of valid time integration subtypes.
- enum timestep_type { CONSTANT, ADAPTIVE, FEHLBERG }

  Enumeration for the list of valid time stepper types.
- enum linesearch_type { BT, ABT, NO_LS }

  Enumeration for the list of valid line search methods.
- enum precond_type {
  JACOBI, TRIDIAG, UGS, LGS,
  SGS }

  Enumeration for the list of valid preconditioning options.

Functions

- int residual_BE (const Matrix< double > &u, Matrix< double > &Res, const void *data)

  Residual function for implicit-BE method.
- int precond_Jac_BE (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Jacobi preconditioner on the implicit-BE method.
- int precond_Tridiag_BE (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Tridiagonal preconditioner on the implicit-BE method.
- int precond_UpperGS_BE (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BE method.
- int precond_LowerGS_BE (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BE method.
- int precond_SymmetricGS_BE (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BE method.
- int residual_CN (const Matrix< double > &u, Matrix< double > &Res, const void *data)

  Residual function for implicit-CN method.
- int precond_Jac_CN (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Jacobi preconditioner on the implicit-CN method.
- int precond_Tridiag_CN (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Tridiagonal preconditioner on the implicit-CN method.
- int precond_UpperGS_CN (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-CN method.
- int precond_LowerGS_CN (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-CN method.
- int precond_SymmetricGS_CN (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-CN method.
- int residual_BDF2 (const Matrix< double > &u, Matrix< double > &Res, const void *data)

  Residual function for implicit-BDF2 method.
- int precond_Jac_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Jacobi preconditioner on the implicit-BDF2 method.
- int precond_Tridiag_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Tridiagonal preconditioner on the implicit-BDF2 method.
- int precond_UpperGS_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BDF2 method.
- int precond_LowerGS_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BDF2 method.
- int precond_SymmetricGS_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void *data)

  Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BDF2 method.
- double default_func (int i, const Matrix< double > &u, double t, const void *data)

  Default function.
- double default_coeff (int i, const Matrix< double > &u, double t, const void *data)

  Default time coefficient function.
- double default_jacobi (int i, int j, const Matrix< double > &u, double t, const void *data)

  Default Jacobian element function.
- int DOVE_TESTS ()

  Test function for DOVE kernel.

6.2.1 Detailed Description

Dynamic ODE solver with Various Established methods.

This file creates objects and subroutines for solving systems of Ordinary Differential Equations using various established methods. The basic idea is that a user will create a function to calculate all the right-hand sides of a system of ODEs, then pass that function to the DOVE routine, which will seek a numerical solution to that system.

Methods for Integration

BE = Backwards-Euler FE = Forwards-Euler CN = Crank-Nicholson BDF2 = Backwards-Differentiation-Formula-2 RK4 = Runge-Kutta-4 RKF = Runge-Kutta-Fehlberg

References for Various Methods

BE and BDF2 => S. Eckert, H. Baaser, D. Gross, O. Scherf, "A BDF2 integration method with step size control for elasto-plasticity," Comp. Mech., 34, 377-386, 2004.

CN and FE => J.W. Thomas, Introduction to Numerical Methods for Partial Differential Equations, Springer, ISBN 0-387-97999-9

RK4 and RKF => B.S. Desale, N.R. Dasre, "Numerical Solution of the System of Six Coupled Nonlinear ODEs by Runge-Kutta Fourth Order Method," Applied Math. Sci., 7, 287 - 305, 2013.

Author

Austin Ladshaw

Date

09/25/2017

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for Post-Doc research in the area of adsorption and surface science. Copyright (c) 2017, all rights reserved.

6.2.2 Macro Definition Documentation

6.2.2.1 #define DOVE_HPP_

6.2.3 Enumeration Type Documentation

6.2.3.1 enum integrate_type

Enumeration for the list of valid time integration types.

The only types that have been defined are for Implicit and Explicit methods. Sub-type enumeration is used to denote the specific methods.

Enumerator

IMPLICIT

EXPLICIT

6.2.3.2 enum integrate_subtype

Enumeration for the list of valid time integration subtypes.

Theses subtypes define the specific scheme to be used. The table below gives a brief description of each.

Parameters

BE	Backwards-Euler: Standard implicit method.
FE	Forwards-Euler: Standard explicit method.
CN	Crank-Nicholson: Time averaged, 2nd order implicit scheme.
BDF2	Backwards-Differentiation-Formula-2: 2nd order implicit method.
RK4	Runge-Kutta-4: 4th order explicit method.
RKF	Runge-Kutta-Fehlberg: 4th order explicit method with 5th order error control.

Enumerator

BE

FE

CN

BDF2

RK4

RKF

6.2.3.3 enum timestep_type

Enumeration for the list of valid time stepper types.

Type of time stepper to be used by Dove.

Parameters

CONSTANT	time stepper will use a constant dt value for all time steps.
ADAPTIVE	time stepper will adjust the time step according to simulation success.
FEHLBERG	time stepper will adjust time step according to desired error tolerance.

Enumerator

CONSTANT

ADAPTIVE

FEHLBERG

6.2.3.4 enum linesearch_type

Enumeration for the list of valid line search methods.

Type of line search method to be used by Dove.

Parameters

BT	uses a basic backtracking linesearch algorithm.
ABT	uses an adaptive backtracking linesearch method.
NO_LS	no line searching will be used.

Enumerator

BT

ABT

NO_LS

6.2.3.5 enum precond_type

Enumeration for the list of valid preconditioning options.

Type of preconditioner to apply to linear iterations.

Parameters

JACOBI	uses a simple Jacobi iteration as preconditioning.
TRIDIAG	uses a Tridiagonal solve as preconditioning.
UGS	uses an Upper-Gauss-Seidel iteration as preconditioning.
LGS	uses a Lower-Gauss-Seidel iteration as preconditioning.
SGS	uses a Symmetric-Gauss-Seidel iteration as preconditioning.

Enumerator

JACOBI

TRIDIAG

UGS

LGS

SGS

6.2.4 Function Documentation

6.2.4.1 int residual_BE (const Matrix< double > & *u*, Matrix< double > & *Res*, const void * *data*)

Residual function for implicit-BE method.

This function will be passed to PJFNK as the residual function for the Dove object. In this function, DOVE will call the user defined rate functions to create a vector of residuals at the current iterate. That information will be passed into the pjfnk function (see lark.h) to iteratively solve the system of equations at a single time step.

Res[i] = Rnp1[i]*unp1[i] - Rn[i]*un[i] - dt*func[i](unp1)

6.2.4.2 int precond_Jac_BE (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Jacobi preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Jacobi preconditioning: Solve Dp=v for p using input vector v and the diagonals (D) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt*jacobi[i][i](unp1)

6.2.4.3 int precond_Tridiag_BE (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Tridiagonal preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Tridiagonal preconditioning: Solve (TD)p=v for p using input vector v and a Tridiagonal (TD) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩ i/du_j = Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.4 int precond_UpperGS_BE (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

UGS preconditioning: Solve (U*)p=v+Lp for p using input vector v with an Upper Triangular (U*) of the full jacobian and a strict lower triangular (L) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩ i/du_j = Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.5 int precond_LowerGS_BE (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

LGS preconditioning: Solve (L*)p=v+Up for p using input vector v and a Lower Triangular (L*) of the full jacobian. and a strict upper triangular (U) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩ i/du_j = Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.6 int precond_SymmetricGS_BE (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

SGS preconditioning: Solve (J)p=v for p using input vector v with the Jacobian matrix (J) approximately by first solving as an Upper-Gauss-Seidel, then as a Lower-Gauss-Seidel.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩ i/du_j = Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.7 int residual_CN (const Matrix< double > & *u*, Matrix< double > & *Res*, const void * *data*)

Residual function for implicit-CN method.

This function will be passed to PJFNK as the residual function for the Dove object. In this function, DOVE will call the user defined rate functions to create a vector of residuals at the current iterate. That information will be passed into the pjfnk function (see lark.h) to iteratively solve the system of equations at a single time step.

Res[i] = Rnp1[i]*unp1[i] - Rn[i]*un[i] - 0.5*dt*func[i](unp1) - 0.5*dt*func[i](un)

6.2.4.8 int precond_Jac_CN (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Jacobi preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Jacobi preconditioning: Solve Dp=v for p using input vector v and the diagonals (D) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5*dt*jacobi[i][i](unp1)

6.2.4.9 int precond_Tridiag_CN (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Tridiagonal preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Tridiagonal preconditioning: Solve (TD)p=v for p using input vector v and a Tridiagonal (TD) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5*dt*jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5*dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5*dt*jacobi[i][j](unp1) for i!=j

6.2.4.10 int precond_UpperGS_CN (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

UGS preconditioning: Solve (U*)p=v+Lp for p using input vector v with an Upper Triangular (U*) of the full jacobian and a strict lower triangular (L) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5*dt*jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5*dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5*dt*jacobi[i][j](unp1) for i!=j

6.2.4.11 int precond_LowerGS_CN (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

LGS preconditioning: Solve (L*)p=v+Up for p using input vector v and a Lower Triangular (L*) of the full jacobian. and a strict upper triangular (U) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5*dt*jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5*dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5*dt*jacobi[i][j](unp1) for i!=j

6.2.4.12 int precond_SymmetricGS_CN (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

SGS preconditioning: Solve (J)p=v for p using input vector v with the Jacobian matrix (J) approximately by first solving as an Upper-Gauss-Seidel, then as a Lower-Gauss-Seidel.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5*dt*jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5*dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5*dt*jacobi[i][j](unp1) for i!=j

6.2.4.13 int residual_BDF2 (const Matrix< double > & *u*, Matrix< double > & *Res*, const void * *data*)

Residual function for implicit-BDF2 method.

This function will be passed to PJFNK as the residual function for the Dove object. In this function, DOVE will call the user defined rate functions to create a vector of residuals at the current iterate. That information will be passed into the pjfnk function (see lark.h) to iteratively solve the system of equations at a single time step. Note that the first time step will be the same as the BE method, then each subsequent time step will be made as a function of un+1, un, and un-1 time levels.

Res[i] = an*Rnp1[i]*unp1[i] - bn*Rn[i]*un[i] + cn*Rnnm1[i]*unm1[i] - dt*func[i](unp1)

where an = (1+2*rn)/(1+rn) ; bn = (1+rn) ; cn = (rn*rn)/(1+rn) and where rn = dt/dt_old

Note

if rn = 0 (i.e. for first step) then this is same as BE method

6.2.4.14 int precond_Jac_BDF2 (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Jacobi preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Jacobi preconditioning: Solve Dp=v for p using input vector v and the diagonals (D) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an*Rnp1[i] - dt*jacobi[i][i](unp1)

6.2.4.15 int precond_Tridiag_BDF2 (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Tridiagonal preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Tridiagonal preconditioning: Solve (TD)p=v for p using input vector v and a Tridiagonal (TD) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an*Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an*Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.16 int precond_UpperGS_BDF2 (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

UGS preconditioning: Solve (U*)p=v+Lp for p using input vector v with an Upper Triangular (U*) of the full jacobian and a strict lower triangular (L) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an*Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an*Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.17 int precond_LowerGS_BDF2 (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

LGS preconditioning: Solve (L*)p=v+Up for p using input vector v and a Lower Triangular (L*) of the full jacobian. and a strict upper triangular (U) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an*Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an*Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.18 int precond_SymmetricGS_BDF2 (const Matrix< double > & *v*, Matrix< double > & *p*, const void * *data*)

Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

SGS preconditioning: Solve (J)p=v for p using input vector v with the Jacobian matrix (J) approximately by first solving as an Upper-Gauss-Seidel, then as a Lower-Gauss-Seidel.

Diagonals for BDF2 are of the form: dR_i/du_i = an*Rnp1[i] - dt*jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an*Rnp1[i] - dt*jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt*jacobi[i][j](unp1) for i!=j

6.2.4.19 double default_func (int *i,* const Matrix< double > & *u,* double *t,* const void * *data*)

Default function.

6.2.4.20 double default_coeff (int *i,* const Matrix< double > & *u,* double *t,* const void * *data*)

Default time coefficient function.

6.2.4.21 double default_jacobi (int *i,* int *j,* const Matrix< double > & *u,* double *t,* const void * *data*)

Default Jacobian element function.

6.2.4.22 int DOVE_TESTS ( )

Test function for DOVE kernel.

This function sets up and solves a test problem for DOVE. It is callable from the UI.

6.3 eel.h File Reference

Easy-access Element Library.

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include "error.h"
```

Classes

- class Atom

  Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions)
- class PeriodicTable

  Class object that store a digitial copy of all Atom objects.

Functions

- int EEL_TESTS ()

  Test function to exercise the class objects and check for errors.

6.3.1 Detailed Description

Easy-access Element Library.

eel.cpp

This file contains two C++ objects: (i) Atom and (ii) PeriodicTable.

```
The Atom class defines all relavent information necessary for dealing with actual
atoms. However, this is not necessarilly all the information that one may need for
any simulation dealing with atoms. Instead, it is really just a place holder used
to construct Molecules and hold oxidation state and molecular/atomic wieght information.

The PeriodicTable class creates a digital version of a complete periodic table. Further
development of this object can make it possible to query this structure for a particular
atom upon user request.
```

Warning

The Atom class is mostly complete, but the PeriodicTable object is just a place holder.

Author

Austin Ladshaw

Date

02/23/2015

Copyright

6.3.2 Function Documentation

6.3.2.1 int EEL_TESTS ( )

Test function to exercise the class objects and check for errors.

6.4 egret.h File Reference

Estimation of Gas-phase pRopErTies.

```
#include "macaw.h"
```

Classes

- struct PURE_GAS

  Data structure holding all the parameters for each pure gas spieces.
- struct MIXED_GAS

  Data structure holding information necessary for computing mixed gas properties.

Macros

- #define Rstd 8.3144621

  Gas Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)
- #define RE3 8.3144621E+3

  Gas Constant in cm^3∗kPa/K/mol (Convenient for density calculations)
- #define Po 100.0

  Standard state pressure (kPa)
- #define Cstd(p, T) ((p)/(Rstd∗T))

  Calculation of concentration/density from partial pressure (Cstd = mol/L)
- #define CE3(p, T) ((p)/(RE3∗T))

  Calculation of concentration/density from partial pressure (CE3 = mol/cm^3)
- #define Pstd(c, T) ((c)∗Rstd∗T)

  Calculation of partial pressure from concentration/density (c = mol/L)
- #define PE3(c, T) ((c)∗RE3∗T)

  Calculation of partial pressure from concentration/density (c = mol/cm^3)
- #define Nu(mu, rho) ((mu)/(rho))

  Calculation of kinematic viscosity from dynamic viscosity and density (cm^2/s)
- #define PSI(T) (0.873143 + (0.000072375∗T))

  Calculation of temperature correction factor for dynamic viscosity.
- #define Dp_ij(Dij, PT) ((PT∗Dij)/Po)

  Calculation of the corrected binary diffusivity (cm^2/s)
- #define D_ij(MWi, MWj, rhoi, rhoj, mui, muj) ((4.0 / sqrt(2.0)) ∗ pow(((1/MWi)+(1/MWj)),0.5)) / pow((pow((pow((rhoi/(1.385∗mui)),2.0)/MWi),0.25)+ pow((pow((rhoj/(1.385∗muj)),2.0)/MWj),0.25)),2.0)

  Calculation of binary diffusion based on MW, density, and viscosity info (cm^2/s)
- #define Mu(muo, To, C, T) (muo ∗ ((To + C)/(T + C)) ∗ pow((T/To),1.5))

  Calculation of single species viscosity from Sutherland's Equ. (g/cm/s)
- #define D_ii(rhoi, mui) (1.385∗mui/rhoi)

  Calculation of self-diffusivity (cm^2/s)
- #define ReNum(u, L, nu) (u∗L/nu)

  Calculation of the Reynold's Number (-)
- #define ScNum(nu, D) (nu/D)

  Calculation of the Schmidt Number (-)
- #define FilmMTCoeff(D, L, Re, Sc) ((D/L)∗(2.0 + (1.1∗pow(Re,0.6)∗pow(Sc,0.3))))

  Calculation of film mass transfer coefficient (cm/s)

Functions

- int initialize_data (int N, MIXED_GAS ∗gas_dat)

  Function to initialize the MIXED_GAS structure based on number of gas species.
- int set_variables (double PT, double T, double us, double L, std::vector< double > &y, MIXED_GAS ∗gas_dat)

  Function to set the values of the parameters in the gas phase.
- int calculate_properties (MIXED_GAS ∗gas_dat)

  Function to calculate the gas properties based on information in MIXED_GAS.
- int EGRET_TESTS ()

  Function runs a series of tests for the EGRET file.

6.4.1 Detailed Description

Estimation of Gas-phase pRopErTies.

egret.cpp

This file is responsible for estimating various temperature, pressure, and concentration dependent parameters to be used in other models for gas phase adsorption, mass transfer, and or mass transport. The goal of this file is to eliminate redundancies in code such that the higher level programs operate more efficiently and cleanly. Calculations made here are based on kinetic theory of gases, ideal gas law, and some emperical models that were developed to account for changes in density and viscosity with changes in temperature between standard temperatures and up to 1000 K.

Author

Austin Ladshaw

Date

01/29/2015

Copyright

6.4.2 Macro Definition Documentation

6.4.2.1 #define Rstd 8.3144621

Gas Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)

6.4.2.2 #define RE3 8.3144621E+3

Gas Constant in cm^3∗kPa/K/mol (Convenient for density calculations)

6.4.2.3 #define Po 100.0

Standard state pressure (kPa)

6.4.2.4 #define Cstd(*p, T*) ((p)/(**Rstd**∗T))

Calculation of concentration/density from partial pressure (Cstd = mol/L)

6.4.2.5 #define CE3(*p, T*) ((p)/(**RE3**∗T))

Calculation of concentration/density from partial pressure (CE3 = mol/cm^3)

6.4.2.6 #define Pstd(*c, T*) ((c)∗**Rstd**∗T)

Calculation of partial pressure from concentration/density (c = mol/L)

6.4.2.7 #define PE3(*c, T*) ((c)∗RE3∗T)

Calculation of partial pressure from concentration/density (c = mol/cm^3)

6.4.2.8 #define Nu(*mu, rho*) ((mu)/(rho))

Calculation of kinematic viscosity from dynamic viscosity and density (cm^2/s)

6.4.2.9 #define PSI(*T*) (0.873143 + (0.000072375∗T))

Calculation of temperature correction factor for dynamic viscosity.

6.4.2.10 #define Dp_ij(*Dij, PT*) ((PT∗Dij)/Po)

Calculation of the corrected binary diffusivity (cm^2/s)

6.4.2.11 #define D_ij(*MWi, MWj, rhoi, rhoj, mui, muj*) ((4.0 / sqrt(2.0)) ∗ pow(((1/MWi)+(1/MWj)),0.5)) / pow((pow((pow((rhoi/(1.385∗mui)),2.0)/MWi),0.25)+ pow((pow((rhoj/(1.385∗muj)),2.0)/MWj),0.25)),2.0)

Calculation of binary diffusion based on MW, density, and viscosity info (cm^2/s)

6.4.2.12 #define Mu(*muo, To, C, T*) (muo ∗ ((To + C)/(T + C)) ∗ pow((T/To),1.5))

Calculation of single species viscosity from Sutherland's Equ. (g/cm/s)

6.4.2.13 #define D_ii(*rhoi, mui*) (1.385∗mui/rhoi)

Calculation of self-diffusivity (cm^2/s)

6.4.2.14 #define ReNum(*u, L, nu*) (u∗L/nu)

Calculation of the Reynold's Number (-)

6.4.2.15 #define ScNum(*nu, D*) (nu/D)

Calculation of the Schmidt Number (-)

6.4.2.16 #define FilmMTCoeff(*D, L, Re, Sc*) ((D/L)∗(2.0 + (1.1∗pow(Re,0.6)∗pow(Sc,0.3))))

Calculation of film mass transfer coefficient (cm/s)

6.4.3 Function Documentation

6.4.3.1 int initialize_data (int *N,* MIXED_GAS ∗ *gas_dat*)

Function to initialize the MIXED_GAS structure based on number of gas species.

This function will initialize the sizes of all vector objects in the MIXED_GAS structure based on the number of gas species indicated by N.

6.4.3.2 int set_variables (double *PT,* double *T,* double *us,* double *L,* std::vector< double > & *y,* MIXED_GAS ∗ *gas_dat*)

Function to set the values of the parameters in the gas phase.

The gas phase properties are a function of total pressure, gas temperature, gas velocity, characteristic length, and the mole fractions of each species in the gas phase. Prior to calculating the gas phase properties, these parameters must be set and updated as they change.

Parameters

PT	total gas pressure in kPa
T	gas temperature in K
us	gas velocity in cm/s
L	characteristic length in cm (this depends on the particular system)
y	vector of gas mole fractions of each species in the mixture
gas_dat	pointer to the MIXED_GAS data structure

6.4.3.3 int calculate_properties (MIXED_GAS * *gas_dat*)

Function to calculate the gas properties based on information in MIXED_GAS.

This function uses the kinetic theory of gases, combined with other semi-empirical models, to predict and approximate several properties of the mixed gas phase that might be necessary when running any gas dynamical simulation. This includes mass and energy transfer equations, as well as adsorption kinetics in porous adsorbents.

6.4.3.4 int EGRET_TESTS ( )

Function runs a series of tests for the EGRET file.

The test looks at a standard air with 5 primary species of interest and calculates the gas properties from 273 K to 373 K. This function can be called from the UI.

6.5 error.h File Reference

All error types are defined here.

```
#include <iostream>
```

Macros

- #define mError(i)

Enumerations

- enum error_type {
  generic_error, file_dne, indexing_error, magpie_reverse_error,
  simulation_fail, invalid_components, invalid_boolean, invalid_molefraction,
  invalid_gas_sum, invalid_solid_sum, scenario_fail, out_of_bounds,
  non_square_matrix, dim_mis_match, empty_matrix, opt_no_support,
  invalid_fraction, ortho_check_fail, unstable_matrix, no_diffusion,
  negative_mass, negative_time, matvec_mis_match, arg_matrix_same,
  singular_matrix, matrix_too_small, invalid_size, nullptr_func,
  invalid_norm, vector_out_of_bounds, zero_vector, tensor_out_of_bounds,
  non_real_edge, nullptr_error, invalid_atom, invalid_proton,
  invalid_neutron, invalid_electron, invalid_valence, string_parse_error,
  unregistered_name, rxn_rate_error, invalid_species, duplicate_variable,
  missing_information, invalid_type, key_not_found, anchor_alias_dne,
  initial_error, not_a_token, read_error, invalid_console_input }

  List of names for error type.

Functions

- void error (int flag)

  Error function customizes output message based on flag.

6.5.1 Detailed Description

All error types are defined here.

error.cpp

This file defines all the different errors that may occur in any simulation in any file. Those errors are recognized by an enum with is then passed through to the error.cpp file that customizes the error message to the console. A macro will also print out the file name and line number where the error occured.

Author

Austin Ladshaw

Date

04/28/2014

Copyright

6.5.2 Macro Definition Documentation

6.5.2.1 #define mError(*i*)

Value:

```
{error(i);          \
std::cout << "Source: " << __FILE__ << "\nLine: " << __LINE__ << std::endl;}
```

6.5.3 Enumeration Type Documentation

6.5.3.1 enum error_type

List of names for error type.

Enumerator

generic_error

file_dne

indexing_error

magpie_reverse_error

simulation_fail

invalid_components
invalid_boolean
invalid_molefraction
invalid_gas_sum
invalid_solid_sum
scenario_fail
out_of_bounds
non_square_matrix
dim_mis_match
empty_matrix
opt_no_support
invalid_fraction
ortho_check_fail
unstable_matrix
no_diffusion
negative_mass
negative_time
matvec_mis_match
arg_matrix_same
singular_matrix
matrix_too_small
invalid_size
nullptr_func
invalid_norm
vector_out_of_bounds
zero_vector
tensor_out_of_bounds
non_real_edge
nullptr_error
invalid_atom
invalid_proton
invalid_neutron
invalid_electron
invalid_valence
string_parse_error
unregistered_name
rxn_rate_error
invalid_species
duplicate_variable
missing_information
invalid_type
key_not_found
anchor_alias_dne
initial_error
not_a_token
read_error
invalid_console_input

6.5.4 Function Documentation

6.5.4.1 void error (int *flag*)

Error function customizes output message based on flag.

This error function is reference in the error.cpp file, but is not called by any other file. Instead, all other files call the mError(i) macro that expands into this error function call plus prints out the file name and line number where the error occured.

6.6 finch.h File Reference

Flux-limiting Implicit Non-oscillatory Conservative High-resolution scheme.

```
#include "macaw.h"
#include "lark.h"
```

Classes

- struct FINCH_DATA

  Data structure for the FINCH object.

Enumerations

- enum finch_solve_type { FINCH_Picard, LARK_Picard, LARK_PJFNK }

  List of enum options to define the solver type in FINCH.
- enum finch_coord_type { Cartesian, Cylindrical, Spherical }

  List of enum options to define the coordinate system in FINCH.

Functions

- double max (std::vector< double > &values)

  Function returns the maximum in a list of values.
- double min (std::vector< double > &values)

  Function returns the minimum in a list of values.
- double minmod (std::vector< double > &values)

  Function returns the result of the minmod function acting on a list of values.
- int uTotal (FINCH_DATA *dat)

  Function integrates the conserved quantity to return it's total in the domain.
- int uAverage (FINCH_DATA *dat)

  Function integrates the conserved quantity to reture it's average in the domain.
- int check_Mass (FINCH_DATA *dat)

  Function checks the unp1 vector for negative values and will adjust if needed.
- int l_direct (FINCH_DATA *dat)

  Function solves the discretized FINCH problem directly by assuming it is linear.
- int lark_picard_step (const Matrix< double > &x, Matrix< double > &G, const void *data)

  Function to perform the necessary LARK Picard iterative method (not typically used)
- int nl_picard (FINCH_DATA *dat)

Function to solve the discretized FINCH problem iteratively by assuming it is non-linear.

- int setup_FINCH_DATA (int(*user_callroutine)(const void *user_data), int(*user_setic)(const void *user_data), int(*user_timestep)(const void *user_data), int(*user_preprocess)(const void *user_data), int(*user_solve)(const void *user_data), int(*user_setparams)(const void *user_data), int(*user_discretize)(const void *user_data), int(*user_bcs)(const void *user_data), int(*user_res)(const Matrix< double > &x, Matrix< double > &res, const void *user_data), int(*user_precon)(const Matrix< double > &b, Matrix< double > &p, const void *user_data), int(*user_postprocess)(const void *user_data), int(*user_reset)(const void *user_data), FINCH_DATA *dat, const void *param_data)

  Function to setup memory and set user defined functions into the FINCH object.
- void print2file_dim_header (FILE *Output, FINCH_DATA *dat)

  Function will print out a dimension header for FINCH output.
- void print2file_time_header (FILE *Output, FINCH_DATA *dat)

  Function will print out a time header for FINCH output.
- void print2file_result_old (FILE *Output, FINCH_DATA *dat)

  Function will print out the old results to the variable u.
- void print2file_result_new (FILE *Output, FINCH_DATA *dat)

  Function will print out the new results to the variable u.
- void print2file_newline (FILE *Output, FINCH_DATA *dat)

  Function will force print out a blank line.
- void print2file_tab (FILE *Output, FINCH_DATA *dat)

  Function will force print out a tab.
- int default_execution (const void *user_data)

  Default executioner function for FINCH.
- int default_ic (const void *user_data)

  Default initial conditions function for FINCH.
- int default_timestep (const void *user_data)

  Default time step function for FINCH.
- int default_preprocess (const void *user_data)

  Default preprocesses function for FINCH.
- int default_solve (const void *user_data)

  Default solve function for FINCH.
- int default_params (const void *user_data)

  Default params function for FINCH.
- int minmod_discretization (const void *user_data)

  Minmod Discretization function for FINCH.
- int vanAlbada_discretization (const void *user_data)

  Van Albada Discretization function for FINCH.
- int ospre_discretization (const void *user_data)

  Ospre Discretization function for FINCH.
- int default_bcs (const void *user_data)

  Default boundary conditions function for FINCH.
- int default_res (const Matrix< double > &x, Matrix< double > &res, const void *user_data)

  Default residual function for FINCH.
- int default_precon (const Matrix< double > &b, Matrix< double > &p, const void *user_data)

  Default preconditioning function for FINCH.
- int default_postprocess (const void *user_data)
- int default_reset (const void *user_data)

  Default reset function for FINCH.
- int FINCH_TESTS ()

  Function runs a particular FINCH test.

6.6.1 Detailed Description

Flux-limiting Implicit Non-oscillatory Conservative High-resolution scheme.

finch.cpp

This is a conservative finite differences scheme based on the Kurganov and Tadmoor (2000) MUSCL scheme for non-linear conservation laws. It can solve 1-D conservation law problems in three different coordinate systems: (i) Cartesian - axial, (ii) Cylindrical - radial, and (iii) Spherical - radial. It is the backbone algorithm behind all 1-D PDE problems in the ecosystem software.

The form of the general conservation law problem that FINCH solves is...

z^d∗R∗du/dt = d/dz(z^d∗D∗du/dz) - d/dz(z^d∗v∗u) - z^d∗k∗u + z^d∗S

where R, D, v, k, and S are the parameters of the problem and d, z, and u are the coordinates, spatial dimension, and conserved quantities, respectively. The parameter R is a retardation coefficient, D is a diffusion coefficient, v is a velocity, k is a reaction coefficient, and S is a forcing function or source/sink term.

FINCH supports the use of both Dirichlet and Neuman boundary conditions as the input/inlet condition and uses the No Flux (or Natural) boundary condition for the output/outlet of the domain. For radial problems, the outlet is always taken to the the center of the cylindrical or spherical particle. This enforces the symmetry of the problem. For axial problems, the outlet is determined by the sign of the velocity term and is therefore choosen by the routine based on the actual flow direction in the domain.

Parameters of the problem can be coupled to the variable u and also be functions of space and time. The coupling of the parameters with the variable forces the problem to become non-linear, which requires iteration to solve. The default iterative method is a built-in Picard's method. This method is equivalent to an inexact Newton method, because we use the Linear Solve of this system as a weak approximation to the non-linear solve. Generally, this method is sufficient and is the most efficient. However, if a problem is particularly difficult to solve, then we can call some of the non-linear solvers developed in LARK. If PJFNK is used, then the Linear Solve for the FINCH problem is used as the Preconditioner for the Linear Solve in PJFNK.

This algorithm comes packaged with three different slope limiter functions to stabilize the velocity term for highly advectively dominate problems. The available slope limiters are: (i) minmod, (ii) van Albada, and (iii) ospre. By default, the FINCH setup function will set the slope limiter to ospre, because this method provides a reasonable compromise between accuracy and efficiency.

Slope Limiter Stats:

minmod -> Highest Accuracy, Lowest Efficiency
van Albada -> Lowest Accuracy, Highest Efficiency
ospre -> Average Accuracy, Average Efficiency

Author

Austin Ladshaw

Date

01/29/2015

Copyright

6.6.2 Enumeration Type Documentation

6.6.2.1 enum finch_solve_type

List of enum options to define the solver type in FINCH.

Enumerator

FINCH_Picard

LARK_Picard

LARK_PJFNK

6.6.2.2 enum finch_coord_type

List of enum options to define the coordinate system in FINCH.

Enumerator

Cartesian

Cylindrical

Spherical

6.6.3 Function Documentation

6.6.3.1 double max (std::vector< double > & *values*)

Function returns the maximum in a list of values.

6.6.3.2 double min (std::vector< double > & *values*)

Function returns the minimum in a list of values.

6.6.3.3 double minmod (std::vector< double > & *values*)

Function returns the result of the minmod function acting on a list of values.

6.6.3.4 int uTotal (FINCH_DATA * *dat*)

Function integrates the conserved quantity to return it's total in the domain.

6.6.3.5 int uAverage (FINCH_DATA * *dat*)

Function integrates the conserved quantity to reture it's average in the domain.

6.6.3.6 int check_Mass (FINCH_DATA * *dat*)

Function checks the unp1 vector for negative values and will adjust if needed.

This function can be turned off or on in the FINCH_DATA structure. Typically, you will want to leave this on so that the routine does not return negative values for u. However, if you want to get negative values of u, then turn this option off.

6.6.3.7 int l_direct (FINCH_DATA * *dat*)

Function solves the discretized FINCH problem directly by assuming it is linear.

6.6.3.8 int lark_picard_step (const Matrix< double > & *x*, Matrix< double > & *G*, const void * *data*)

Function to perform the necessary LARK Picard iterative method (not typically used)

6.6.3.9 int nl_picard (FINCH_DATA * *dat*)

Function to solve the discretized FINCH problem iteratively by assuming it is non-linear.

Note

If the problem is actually linear, then this will solve it in one iteration. So it may be best to always assume the problem is non-linear.

6.6.3.10 int setup_FINCH_DATA (int(*)(const void *user_data) *user_callroutine*, int(*)(const void *user_data) *user_setic*, int(*)(const void *user_data) *user_timestep*, int(*)(const void *user_data) *user_preprocess*, int(*)(const void *user_data) *user_solve*, int(*)(const void *user_data) *user_setparams*, int(*)(const void *user_data) *user_discretize*, int(*)(const void *user_data) *user_bcs*, int(*)(const Matrix< double > &x, Matrix< double > &res, const void *user_data) *user_res*, int(*)(const Matrix< double > &b, Matrix< double > &p, const void *user_data) *user_precon*, int(*)(const void *user_data) *user_postprocess*, int(*)(const void *user_data) *user_reset*, FINCH_DATA * *dat*, const void * *param_data*)

Function to setup memory and set user defined functions into the FINCH object.

This function MUST be called prior to running any FINCH based simulation. However, you are only every required to provide this function with the FINCH_DATA pointer. It is recommended, however, that you do provide the user↩ _setparams and param_data pointers, as these will likely vary significantly from problem to problem.

After the problem is setup in memory, you do not technically have to have FINCH call all of it's own functions. You can write your own executioner, initial conditions, and other functions and decided how and when everything is called. Then just call the solve function in FINCH_DATA when you want to use the FINCH solver. This is how FINCH is used in SKUA, SCOPSOWL, DOGFISH, and MONKFISH.

Parameters

user_callroutine	function pointer the the call routine function
user_setic	function pointer to set initial conditions for problem
user_timestep	function pointer to set the next time step
user_preprocess	function pointer to setup a preprocess operation
user_solve	function pointer to solve the system of equations
user_setparams	function pointer to set the parameters in the problem (always override this)
user_discretize	function pointer to select discretization scheme for the problem
user_bcs	function pointer to evaluate boundary conditions for the problem
user_res	function pointer to evaluate non-linear residuals for the problem
user_precon	function pointer to perform a linear preconditioning operation
user_postprocess	function pointer to setup a postprocess operation
user_reset	function pointer to reset stateful data for next simulation
dat	pointer to the FINCH_DATA structure
param_data	user supplied pointer to a data structure needed in user_setparams

6.6.3.11 void print2file_dim_header (FILE * *Output,* FINCH_DATA * *dat*)

Function will print out a dimension header for FINCH output.

6.6.3.12 void print2file_time_header (FILE * *Output,* FINCH_DATA * *dat*)

Function will print out a time header for FINCH output.

6.6.3.13 void print2file_result_old (FILE * *Output,* FINCH_DATA * *dat*)

Function will print out the old results to the variable u.

6.6.3.14 void print2file_result_new (FILE * *Output,* FINCH_DATA * *dat*)

Function will print out the new results to the variable u.

6.6.3.15 void print2file_newline (FILE * *Output,* FINCH_DATA * *dat*)

Function will force print out a blank line.

6.6.3.16 void print2file_tab (FILE * *Output,* FINCH_DATA * *dat*)

Function will force print out a tab.

6.6.3.17 int default_execution (const void * *user_data*)

Default executioner function for FINCH.

The default executioner function for FINCH assumes the user_data parameter is the FINCH_DATA structure and calls the preprocesses, solve, postprocesses, checkMass, uTotal, and uAverage functions in that order.

6.6.3.18 int default_ic (const void * *user_data*)

Default initial conditions function for FINCH.

The default initial condition function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the initial values of all system parameters according to the given constants in that structure.

6.6.3.19 int default_timestep (const void * *user_data*)

Default time step function for FINCH.

The default time step function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the time step to 1/2 the mesh size or bases the time step off of the CFL condition if the problem is not being solved iteratively and involves an advective portion.

6.6.3.20 int default_preprocess (const void * *user_data*)

Default preprocesses function for FINCH.

The default preprocesses function for FINCH assumes the user_data parameter is the FINCH_DATA structure and does nothing.

6.6.3.21 int default_solve (const void * *user_data*)

Default solve function for FINCH.

The default solve function for FINCH assumes the user_data parameter is the FINCH_DATA structure and calls the corresponding solution method depending on the users conditions.

6.6.3.22 int default_params (const void * *user_data*)

Default params function for FINCH.

The default params function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the values of all parameters at all nodes equal to the values of those parameters at the boundaries.

6.6.3.23 int minmod_discretization (const void * *user_data*)

Minmod Discretization function for FINCH.

The minmod discretization function for FINCH assumes the user_data parameter is the FINCH_DATA structure and discretizes the time and space portion of the problem with 2nd order finite differences and uses the minmod slope limiter function to stabilize the advective physics.

6.6.3.24 int vanAlbada_discretization (const void * *user_data*)

Van Albada Discretization function for FINCH.

The van Albada discretization function for FINCH assumes the user_data parameter is the FINCH_DATA structure and discretizes the time and space portion of the problem with 2nd order finite differences and uses the van Albada slope limiter function to stabilize the advective physics.

6.6.3.25 int ospre_discretization (const void * *user_data*)

Ospre Discretization function for FINCH.

The ospre discretization function for FINCH assumes the user_data parameter is the FINCH_DATA structure and discretizes the time and space portion of the problem with 2nd order finite differences and uses the ospre slope limiter function to stabilize the advective physics. This is the default discretization function.

6.6.3.26 int default_bcs (const void * *user_data*)

Default boundary conditions function for FINCH.

The default boundary conditions function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the boundary conditions according to the type of problem requested. The input BCs will always be either Neumann or Dirichlet and the output BC will always be a zero flux Neumann BC.

6.6.3.27 int default_res (const Matrix< double > & *x,* Matrix< double > & *res,* const void * *user_data*)

Default residual function for FINCH.

The default residual function for FINCH assumes the user_data parameter is the FINCH_DATA structure and calls the setparams function (passing the param_data structure), the discretization function, and the set BCs functions, in that order. It then forms the implicit and explicit side residuals that go into the iterative solver.

6.6.3.28 int default_precon (const Matrix< double > & *b*, Matrix< double > & *p*, const void * *user_data*)

Default preconditioning function for FINCH.

The default preconditioning function for FINCH assumes the user_data parameter is the FINCH_DATA structure and performs a tridiagonal linear solve using a Modified Thomas Algorithm. This preconditioner will solve the linear problem exactly if there is no advective portion of the physics. Additionally, this preconditioner is also used as the basis for forming the default FINCH non-linear iterations and is sufficient for solving most problems.

6.6.3.29 int default_postprocess (const void * *user_data*)

The default postprocesses function for FINCH assumes the user_data parameter is the FINCH_DATA structure and does nothing.

6.6.3.30 int default_reset (const void * *user_data*)

Default reset function for FINCH.

The default reset function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets all old state parameters and variables to the new state.

6.6.3.31 int FINCH_TESTS ( )

Function runs a particular FINCH test.

The FINCH_TESTS function is used to exercise and test out the FINCH algorithms for correctness, efficiency, and accuracy. This test should never report a failure.

6.7 flock.h File Reference

FundamentaL Off-gas Collection of Kernels.

```
#include "macaw.h"
#include "egret.h"
#include "finch.h"
#include "lark.h"
#include "skua.h"
#include "scopsowl.h"
#include "gsta_opt.h"
#include "magpie.h"
#include "skua_opt.h"
#include "scopsowl_opt.h"
#include "yaml_wrapper.h"
#include "dove.h"
```

6.7.1 Detailed Description

FundamentaL Off-gas Collection of Kernels.

This is just a .h file that holds all the includes necessary to develop and run simulations for adsorption and/or mass/energy transfer problems for gaseous systems. Include this file into any other project or source code that needs the methods below.

Files Included in FLOCK

macaw.h egret.h finch.h lark.h skua.h scopsowl.h gsta_opt.h magpie.h skua_opt.h scopsowl_opt.h yaml_wrapper.h dove.h

Author

Austin Ladshaw

Date

04/28/2014

Copyright

6.8 gsta_opt.h File Reference

Generalized Statistical Thermodynamic Adsorption (GSTA) Optimization Routine.

```
#include "lmcurve.h"
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include "error.h"
```

Classes

- struct GSTA_OPT_DATA

  Data structure used in the GSTA optimization routines.

Macros

- #define Po 100.0

  Standard State Pressure - Units: kPa.
- #define R 8.3144621

  Gas Constant - Units: J/(K∗mol) = kB ∗ Na.
- #define Na 6.0221413E+23

  Avagadro's Number - Units: molecules/mol.

Functions

- int roundIt (double d)

  Function rounds a double to an integer.
- int twoFifths (int m)

  Function returns the rounded two-fifths result of int m.
- int orderMag (double x)

  Function returns the order of magnitude for the parameter x.
- int minValue (std::vector< int > &array)

  Function returns the minimum integer in an array of integers.
- int minIndex (std::vector< double > &array)

  Function returns the index of the minimum integer in an array of integers.
- int avgPar (std::vector< int > &array)

  Function returns the average integer value in an array of integers.
- double avgValue (std::vector< double > &array)

  Function returns an average in an array of doubles.
- double weightedAvg (double *enorm, double *x, int n)

  Function returns a weighted average in an array.
- double rSq (double *x, double *y, double slope, double vint, int m_dat)

  Function calculates the Coefficient of Determination (R Squared) for the temperature regression.
- bool isSmooth (double *par, void *data)

  Function looks at the list of parameters to check if they are smoothly changing.
- void orthoLinReg (double *x, double *y, double *par, int m_dat, int n_par)

  Function performs an Orthogonal Linear Regression on a set of data.
- void eduGuess (double *P, double *q, double *par, int k, int m_dat, void *data)

  Function will formed an educated guess for the next set of parameters in the GSTA analysis.
- double gstaFunc (double p, const double *K, double qmax, int n_par)

  Function evaluates the result of the GSTA isotherm model.
- double gstaObjFunc (double *t, double *y, double *par, int m_dat, void *data)

  Function to evaulate the GSTA objective function value.
- void eval_GSTA (const double *par, int m_dat, const void *data, double *fvec, int *info)

  Function to evaluate the GSTA model and feed into the lmfit routine.
- int gsta_optimize (const char *fileName)

  Function to perform the GSTA optimization routine.

6.8.1 Detailed Description

Generalized Statistical Thermodynamic Adsorption (GSTA) Optimization Routine.

gsta_opt.cpp

Optimization routine developed for the GSTA isotherm and data analysis. This algorithm was the primary subject of a publication made in Fluid Phase Equilibria. Please refer to the below paper for technical information about the algorithms.

Reference: Ladshaw, Yiacoumi, Tsouris, and DePaoli, Fluid Phase Equilibria, 388, 169-181, 2015.

The GSTA model was first introduced by Llano-Restrepo and Mosquera (2009). Please refer to the below reference for theoretical information about the model.

Reference: Llano-Restrepo and Mosquera, Fluid Phase Equilibria, 283, 73-88, 2009.

Author

Austin Ladshaw

Date

12/17/2013

Copyright

6.8.2 Macro Definition Documentation

6.8.2.1 #define Po 100.0

Standard State Pressure - Units: kPa.

6.8.2.2 #define R 8.3144621

Gas Constant - Units: J/(K∗mol) = kB ∗ Na.

6.8.2.3 #define Na 6.0221413E+23

Avagadro's Number - Units: molecules/mol.

6.8.3 Function Documentation

6.8.3.1 int roundIt (double *d*)

Function rounds a double to an integer.

This function returns a rounded value of d. Rounding up for any decimal larger than 0.5 and down for all else.

6.8.3.2 int twoFifths (int *m*)

Function returns the rounded two-fifths result of int m.

This function is used to determine what the maximum number of parameters should be based on the number of data points m. It is designed to prevent the algorithms from "over fitting" the data.

6.8.3.3 int orderMag (double *x*)

Function returns the order of magnitude for the parameter x.

This function is used to help create initial guesses for the new GSTA parameters that are being optimized for. In order to make sure that those parameters are considered relavent in the optimization routine, we need to make the initial guesses to be around the same order of magnitude of the other GSTA parameters.

6.8.3.4 int minValue (std::vector< int > & *array*)

Function returns the minimum integer in an array of integers.

This function is used to determine the minimum number of GSTA parameters that were required to adequately fit the isotherm data.

6.8.3.5 int minIndex (std::vector< double > & *array*)

Function returns the index of the minimum integer in an array of integers.

This function identifies the index of the minimum number of parameters needed for the GSTA model to fit the data. This index is common for all vectors in the GSTA_OPT_DATA structure and is used to identify the most suitable solution.

6.8.3.6 int avgPar (std::vector< int > & *array*)

Function returns the average integer value in an array of integers.

This function is used to identify the average number of parameters that all the data fitting needed for each GSTA analysis.

6.8.3.7 double avgValue (std::vector< double > & *array*)

Function returns an average in an array of doubles.

6.8.3.8 double weightedAvg (double * *enorm*, double * *x*, int *n*)

Function returns a weighted average in an array.

This averaging scheme is used to approximate the qmax parameter for the GSTA isotherm model, if that value is unknown. The weighting is based on the euclidean norms of all the fits of the data. Smaller norms are more heavily weighted since they represent a better fit of the data. Once averaging is complete and we have an estimate for qmax, the entire algorithm is re-run holding that qmax constant.

Parameters

enorm	array of euclidean norms from the fitting of the data
x	array of optimum qmax values to be averaged
n	the number of enorm and x values in the array

6.8.3.9 double rSq (double * *x*, double * *y*, double *slope*, double *vint*, int *m_dat*)

Function calculates the Coefficient of Determination (R Squared) for the temperature regression.

This function is used to determine the "fittness" of the linear regression performed on the temperature independent parameters of the GSTA isotherm. A good linear regression should return a value between 1.0 and 0.9.

Parameters

x	observations in the x-axis
y	observations in the y-axis

Parameters

slope	slope of the linear regression
vint	intercept of the linear regression
m_dat	number of data points used in the linear regression

6.8.3.10 bool isSmooth (double * *par*, void * *data*)

Function looks at the list of parameters to check if they are smoothly changing.

This function takes the parameter array par and GSTA_OPT_DATA structure and checks to see if those parameters are changing smoothly. If they are erratic or non-smooth, then it could be an indication of "over fitting" of the data.

6.8.3.11 void orthoLinReg (double * *x*, double * *y*, double * *par*, int *m_dat*, int *n_par*)

Function performs an Orthogonal Linear Regression on a set of data.

This function takes an array of x and y observations and performs an orthogonal linear regression on that information to find optimum parameters for slope and intercept.

Parameters

x	array of x-axis observations
y	array of y-axis observations
par	array of parameter results after regression
m_dat	number of data points or observations
n_par	number of parameters to seek (if n_par != 1 or 2, then par[0] = intercept and par[1] = slope)

6.8.3.12 void eduGuess (double * *P*, double * *q*, double * *par*, int *k*, int *m_dat*, void * *data*)

Function will formed an educated guess for the next set of parameters in the GSTA analysis.

This function takes partial pressure and adsorption observations, P and q, and tries to give a decent initial guess to what the GSTA parameters, par, will be for the next iteration.

Parameters

P	partial pressure observations in the data (kPa)
q	adsorption observations in the data (any units)
par	parameter array for the GSTA isotherm
k	index of the current number of parameters being considered
m_dat	number of pressure-adsorption observations in the isotherm
data	pointer to the GSTA_OPT_DATA data structure

6.8.3.13 double gstaFunc (double *p*, const double * *K*, double *qmax*, int *n_par*)

Function evaluates the result of the GSTA isotherm model.

This function will evaluate the GSTA model and return the adsorbed amount given the current partial pressure p and the equilibrium parameters K.

Parameters

p	current partial pressure (kPa)
K	array of equilibrium parameters (1/kPa^n)
qmax	the theorectical maximum capacity for the isotherm
n_par	the number of equilibrium parameters

6.8.3.14 double gstaObjFunc (double * *t*, double * *y*, double * *par*, int *m_dat*, void * *data*)

Function to evaulate the GSTA objective function value.

The objective function seeks to penalize the relative fittness of the model based on the number of parameters it took to minimize the euclidean norms. By penalizing the fittness of the model in this fashion, we can find the best solution to the system that required the least number of equilibrium parameters.

6.8.3.15 void eval_GSTA (const double * *par*, int *m_dat*, const void * *data*, double * *fvec*, int * *info*)

Function to evaluate the GSTA model and feed into the lmfit routine.

This function will formulate the residuals that go into the Levenberg-Marquardt's Algorithm for non-linear least squares regression. The form of this function is specific to how we interface with the lmfit routines.

6.8.3.16 int gsta_optimize (const char * *fileName*)

Function to perform the GSTA optimization routine.

This function is callable from the UI and is used to find the optimum parameters of the GSTA isotherm model given a particular set of isotherm data for single-component adsorption equilibria.

Parameters

fileName	name of the input file that holds the isotherm data

Note

The input file for the GSTA optimization routine is a text file holding the necessary information and data needed to run the routine. That input file has a very specific format that is detailed below.
Number of Isotherm Curves
Theoretical Maximum Adsorption Capacity (if unknown, provide 0)
Temperature of the ith Isotherm (K)
Number of Data points for the ith Isotherm
Partial Pressure (kPa) [tab] Corresponding Adsorbed Amount (any units)
(2nd Line down is repeated for all isotherms you are optimizing on...)

Example:

2
21.0
298.15
4
0.000165483 2.77

0.000306379 2.75
0.00044922 5.00
0.000939259 10.40
313.15
4
0.000589636 2.75
0.001063584 3.70
0.001351836 4.2
0.001543464 4.6
The above example would be for 2 sets of isotherms at 298.15 and 313.15 K, respectively. Maximum adsorption capacity is given as 21 (which in this has units of wt%). Each isotherm has 4 data points, which are given in a list as p (kPa) and q (wt%) pairs. Units of adsorption don't matter as long as they are consistent. If you give maximum capacity in mol/kg, then the q's in the lists must also be in mol/kg.

6.9 lark.h File Reference

Linear Algebra Residual Kernels.

```
#include "macaw.h"
#include <float.h>
```

Classes

- struct ARNOLDI_DATA

  Data structure for the construction of the Krylov subspaces for a linear system.
- struct GMRESLP_DATA

  Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning.
- struct GMRESRP_DATA

  Data structure for the Restarted GMRES algorithm with Right Preconditioning.
- struct PCG_DATA

  Data structure for implementation of the PCG algorithms for symmetric linear systems.
- struct BiCGSTAB_DATA

  Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems.
- struct CGS_DATA

  Data structure for the implementation of the CGS algorithm for non-symmetric linear systems.
- struct OPTRANS_DATA

  Data structure for implementation of linear operator transposition.
- struct GCR_DATA

  Data structure for the implementation of the GCR algorithm for non-symmetric linear systems.
- struct GMRESR_DATA

  Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GMRESR)
- struct KMS_DATA

  Data structure for the implemenation of the Krylov Multi-Space (KMS) Method.
- struct QR_DATA

  Data structure for the implementation of a QR solver given some invertable linear operator.
- struct PICARD_DATA

  Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems.
- struct BACKTRACK_DATA

  Data structure for the implementation of Backtracking Linesearch.
- struct PJFNK_DATA

  Data structure for the implementation of the PJFNK algorithm for non-linear systems.
- struct NUM_JAC_DATA

  Data structure to form a numerical jacobian matrix with finite differences.

Macros

- #define MIN_TOL 1e-15

  Minimum Allowable Tolerance for linear and non-linear problems.

Enumerations

- enum krylov_method {
  GMRESLP, PCG, BiCGSTAB, CGS,
  FOM, GMRESRP, GCR, GMRESR,
  KMS, QR }

  Enum of definitions for linear solver types in PJFNK.

Functions

- int update_arnoldi_solution (Matrix< double > &x, Matrix< double > &x0, ARNOLDI_DATA *arnoldi_dat)

  Function to update the linear vector x based on the Arnoldi Krylov subspace.
- int arnoldi (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &r0, ARNOLDI_DATA *arnoldi_dat, const void *matvec_data, const void *precon_data)

  Function to factor a linear operator into an orthonormal basis and upper Hessenberg matrix.
- int gmresLeftPreconditioned (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &b, GMRESLP_DATA *gmreslp_dat, const void *matvec_data, const void *precon_data)

  Function to iteratively solve a non-symmetric, indefinite linear system with GMRESLP.
- int fom (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &b, GMRESLP_DATA *gmreslp_dat, const void *matvec_data, const void *precon_data)

  Function to directly solve a non-symmetric, indefinite linear system with FOM.
- int gmresRightPreconditioned (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &b, GMRESRP_DATA *gmresrp_dat, const void *matvec_data, const void *precon_data)

  Function to iteratively solve a non-symmetric, indefinite linear system with GMRESRP.
- int pcg (int(*matvec)(const Matrix< double > &p, Matrix< double > &Ap, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &z, const void *data), Matrix< double > &b, PCG_DATA *pcg_dat, const void *matvec_data, const void *precon_data)

  Function to iteratively solve a symmetric, definite linear system with PCG.
- int bicgstab (int(*matvec)(const Matrix< double > &p, Matrix< double > &Ap, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &z, const void *data), Matrix< double > &b, BiCGSTAB_DATA *bicg_dat, const void *matvec_data, const void *precon_data)

  Function to iteratively solve a non-symmetric, definite linear system with BiCGSTAB.
- int cgs (int(*matvec)(const Matrix< double > &p, Matrix< double > &Ap, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &z, const void *data), Matrix< double > &b, CGS_DATA *cgs_dat, const void *matvec_data, const void *precon_data)

  Function to iteratively solve a non-symmetric, definite linear system with CGS.
- int operatorTranspose (int(*matvec)(const Matrix< double > &v, Matrix< double > &Av, const void *data), Matrix< double > &r, Matrix< double > &u, OPTRANS_DATA *transpose_dat, const void *matvec_data)

  *Function that is used to perform transposition of a linear operator and results in a new vector A^T*r=u.*
- int gcr (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &Mr, const void *data), Matrix< double > &b, GCR_DATA *gcr_dat, const void *matvec_data, const void *precon_data)

  Function to iteratively solve a non-symmetric, definite linear system with GCR.

- int gmresrPreconditioner (const Matrix< double > &r, Matrix< double > &Mr, const void *data)

  Function used in conjunction with GMRESR to apply GMRESRP iterations as a preconditioner.
- int gmresr (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), int(*terminal_precon)(const Matrix< double > &r, Matrix< double > &Mr, const void *data), Matrix< double > &b, GMRESR_DATA *gmresr_dat, const void *matvec_data, const void *term_precon_data)

  Function to iteratively solve a non-symmetric, indefinite linear system with GMRESR.
- int kmsPreconditioner (const Matrix< double > &r, Matrix< double > &Mr, const void *data)

  Preconditioner function for the Krylov Multi-Space.
- int krylovMultiSpace (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), int(*terminal_precon)(const Matrix< double > &r, Matrix< double > &Mr, const void *data), Matrix< double > &b, KMS_DATA *kms_dat, const void *matvec_data, const void *term_precon_data)

  Function to iteratively solve a non-symmetric, indefinite linear system with KMS.
- int QRsolve (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), Matrix< double > &b, QR_DATA *qr_dat, const void *matvec_data)

  Function to solve a dense linear operator system using QR factorization.
- int picard (int(*res)(const Matrix< double > &x, Matrix< double > &r, const void *data), int(*evalx)(const Matrix< double > &x0, Matrix< double > &x, const void *data), Matrix< double > &x, PICARD_DATA *picard_dat, const void *res_data, const void *evalx_data)

  Function to iteratively solve a non-linear system using the Picard or Fixed-Point method.
- int jacvec (const Matrix< double > &v, Matrix< double > &Jv, const void *data)

  Function to form a linear operator of a Jacobian matrix used along with the PJFNK method.
- int backtrackLineSearch (int(*feval)(const Matrix< double > &x, Matrix< double > &F, const void *data), Matrix< double > &Fkp1, Matrix< double > &xkp1, Matrix< double > &pk, double normFk, BACKTRAC↩ K_DATA *backtrack_dat, const void *feval_data)

  Function to perform a Backtracking Line Search operation to smooth out convergence of PJFNK.
- int pjfnk (int(*res)(const Matrix< double > &x, Matrix< double > &F, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &p, const void *data), Matrix< double > &x, PJFNK_DATA *pjfnk↩ _dat, const void *res_data, const void *precon_data)

  Function to perform the PJFNK algorithm to solve a non-linear system of equations.
- int NumericalJacobian (int(*Func)(const Matrix< double > &x, Matrix< double > &F, const void *user_↩ data), const Matrix< double > &x, Matrix< double > &J, int Nx, int Nf, NUM_JAC_DATA *jac_dat, const void *user_data)

  Function to form a full numerical Jacobian matrix from a given non-linear function.
- int LARK_TESTS ()

  Function that runs a variety of tests on all the functions in LARK.

6.9.1 Detailed Description

Linear Algebra Residual Kernels.

lark.cpp

The functions contained within are designed to solve generic linear and non-linear square systems of equations given a function argument and data from the user. Optionally, the user can also provide a function to return a preconditioning result that will be applied to the system.

Having the user define how the preconditioning is carried out provides two major advantages: (1) we do not need to store and large, sparse preconditioning matrices and instead only store the preconditioned vector result and (2) this allows the user to use any kind of preconditioner they see fit for their problem.

The Arnoldi function is typically not called by the user, but can be if desired. It accepts the function arguments and a residual vector to form an orthonormal basis of the Krylov subspace using the Modified Gram-Schmidt process (aka Arnoldi Iteration). This function is called by GMRES to iteratively solve a linear system of equations. Note that

you can use this function to directly solve the linear system as long as that system is not too large. Construction of the basis is expensive, which is why this is used as a sub-function of an iterative method.

The Restarted GMRES function will accept function arguments for a linear system and attempt to solve said system iteratively by constructing an orthonormal basis from the Krylov function. Note that this GMRES function does support restarting and will use restarting by default if the linear system is too large.

Also included is a GMRES algorithm without restarting. This will directly solve the linear system within residual tolerance using a Full Orthogonal basis set of that system. It is equivalent to calling the Krylov method with the k parameter equal to N (i.e. the number of equations). This method is nick-named the Full Othogonalization Method (FOM), although the true FOM algorithm in literature is slightly different.

The PJFNK function will accept function arguments for a square, non-linear system of equations and attempt to solve it iteratively using both the GMRES and Krylov functions with Newton's method to convert the non-linear system into a linear system.

Also built here is a PCG implementation for solving symmetric linear systems. Can also be called by PJFNK if we know that the linear system (i.e. the Jacobian) is symmetric. This algorithm is significantly more efficient than GMRES, but is only valid if the system of equations is symmetric.

Other linear solvers implemented in this work are the BiCGSTAB and CGS algorithms for non-symmetric, positive definite matrices. These algorithms are significantly more computationally efficient than GMRES or FOM. However, they can both break down if the linear system is poorly conditioned. In general, you only want to use these methods if you have preconditioning available and your linear system is very, very large. Otherwise, you will be better suited to using GMRES or FOM.

There is also an implementation of the Generalized Conjugate Residual (GCR) method with and without restarting. This is a GMRES-like method that should give the exact solution within N iterations, where N is the original size of the matrix. Built ontop of the GCR method is a GMRESR (or GMRES Recursive) algorithm that uses GCR as the base method and performs GMRESRP iterations as a preconditioner at each iteration of GCR. This is the only linear solver that has built-in preconditioning. As a result, it may be slower than other algorithms for simple problems, but generally will have much better convergence behavior and will almost always give better residual reduction, even for hard to solve problems.

We have also developed a novel/experimental iterative method based on the idea of recursively preconditioning a Krylov Subspace with more Krylov Subspaces. We have called with algorithm the Krylov Multi-Space (KMS) method. This algorithm is based on publications from Vorst and Vuik (1991) and Saad (1993). The idea is too use the FGMRES algorithm developed by Saad (1993) and precondition it with more FGMRES steps, i.e., nesting the iterations as Vorst and Vuik (1991) had proposed. In this way, we have created a generalized Krylov Subspace method that has it's own variable preconditioner that can be adjusted depending on the user's desired complexity and convergence rate. If the levels of recursion requested is zero, then this algorithm is exactly equal to GMRES with right preconditioning. If the level is one, then it is FGMRES with a GMRES preconditioner. However, we allow the levels of recursion to reach up to 5, thus allowing us to precondition the preconitioners with more GMRES steps. This can result is significantly faster convergence rates, but is typically only necessary for very large or difficult to solve problems.

NOTE: There are three GMRES implementations: (i) gmresLP, (ii) fom, and (iii) gmresRP. GMRESLP is a restarted GMRES implementation that is left preconditioned and only checks the residual on the outer loops. This may be less efficient than GMRESRP, which can check both outer and inner loop residuals. However, GMRESRP has to use right preconditioning, which also slightly changes the convergence behavior of the linear system. GMRES with left preconditioning and without restarting will just build the full subspace by default, thus solving the system exactly, but may require too much memory. You can do a GMRESRP unrestarted by specifying that the restart parameter be equal to the size of the problem.

Basic Implementation Details:

Linear Solvers -> Solve Ax=b for x
Non-Linear Solvers -> Solve F(x)=0 for x
All implementations require system size to be 2 or greater

Author

Austin Ladshaw

Date

10/14/2014

Copyright

6.9.2 Macro Definition Documentation

6.9.2.1 #define MIN_TOL 1e-15

Minimum Allowable Tolerance for linear and non-linear problems.

6.9.3 Enumeration Type Documentation

6.9.3.1 enum krylov_method

Enum of definitions for linear solver types in PJFNK.

Enum delineates the available Krylov Subspace methods that can be used to solve the linear sub-problem at each non-linear iteration in a Newton method.

Enumerator

GMRESLP
PCG
BiCGSTAB
CGS
FOM
GMRESRP
GCR
GMRESR
KMS
QR

6.9.4 Function Documentation

6.9.4.1 int update_arnoldi_solution (Matrix< double > & *x*, Matrix< double > & *x0*, ARNOLDI_DATA * *arnoldi_dat*)

Function to update the linear vector x based on the Arnoldi Krylov subspace.

This function will update a solution vector x based on the previous solution x0 given the orthonormal basis and upper Hessenberg matrix formed in the Arnoldi algorithm. Updating is automatically called by the GMRESLP function. It is expected that the Arnoldi algorithm has already been called prior to calling this function.

Parameters

x	matrix that will hold the new updated solution to the linear system
x0	matrix that holds the previous solution to the linear system
arnoldi_dat	pointer to the ARNOLDI_DATA data structure

6.9.4.2 int arnoldi (int(∗)(const Matrix< double > &v, Matrix< double > &w, const void ∗data) *matvec,* int(∗)(const Matrix< double > &b, Matrix< double > &p, const void ∗data) *precon,* Matrix< double > & *r0,* ARNOLDI_DATA ∗ *arnoldi_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to factor a linear operator into an orthonormal basis and upper Hessenberg matrix.

This function performs the Arnoldi algorithm to factor a linear operator into an orthonormal basis and upper Hessenberg matrix. Each orthonormal vector is formed using a Modified Gram-Schmidt procedure. When used in conjunction with GMRESLP, user may supply a preconditioning operator to improve convergence of the linear system. However, this function can be used by itself to factor the user's linear operator.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
r0	user supplied vector to serve as the first basis vector in the orthonormal basis
arnoldi_dat	pointer to the ARNOLDI_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.3 int gmresLeftPreconditioned (int(∗)(const Matrix< double > &v, Matrix< double > &w, const void ∗data) *matvec,* int(∗)(const Matrix< double > &b, Matrix< double > &p, const void ∗data) *precon,* Matrix< double > & *b,* GMRESLP_DATA ∗ *gmreslp_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to iteratively solve a non-symmetric, indefinite linear system with GMRESLP.

This function iteratively solves a non-symmetric, indefinite linear system using the Generalized Minimum RE↩ Sidual method with Left Preconditioning (GMRESLP). It calls the Arnoldi algorithm to factor a linear operator into an

orthonormal basis and upper Hessenberg matrix, then uses that factorization to form an approximation to the linear system. Because this algorithm uses left-side preconditioning, it can only check the linear residuals at the outer iterations.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmreslp_dat	pointer to the GMRESLP_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.4 int fom (int(∗)(const Matrix< double > &v, Matrix< double > &w, const void ∗data) *matvec,* int(∗)(const Matrix< double > &b, Matrix< double > &p, const void ∗data) *precon,* Matrix< double > & *b,* GMRESLP_DATA ∗ *gmreslp_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to directly solve a non-symmetric, indefinite linear system with FOM.

This function directly solves a non-symmetric, indefinite linear system using the Full Orthogonalization Method (F↩OM). This algorithm is exactly equivalent to GMRESLP without restarting. Therefore, it uses the GMRESLP_DATA structure and calls the GMRESLP algorithm without using restarts. As a result, it never checks linear residuals. However, this should give the exact solution upon completion, assuming the linear operator is not singular.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmreslp_dat	pointer to the GMRESLP_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and

anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.5 int gmresRightPreconditioned (int(∗)(const Matrix< double > &v, Matrix< double > &w, const void ∗data) *matvec,* int(∗)(const Matrix< double > &b, Matrix< double > &p, const void ∗data) *precon,* Matrix< double > & *b,* GMRESRP_DATA ∗ *gmresrp_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to iteratively solve a non-symmetric, indefinite linear system with GMRESRP.

This function iteratively solves a non-symmetric, indefinite linear system using the Generalized Minimum RESidual method with Right Preconditioning (GMRESRP). Because this algorithm uses right preconditioning, it is able to check the linear residuals at both the outer and inner iterations. This may be much for efficient compared to G↩ MRESLP. In order to check inner residuals, this algorithm has to perform it's own internal Modified Gram-Schmidt procedure and will not call the Arnoldi algorithm.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmresrp_dat	pointer to the GMRESRP_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.6 int pcg (int(∗)(const Matrix< double > &p, Matrix< double > &Ap, const void ∗data) *matvec,* int(∗)(const Matrix< double > &r, Matrix< double > &z, const void ∗data) *precon,* Matrix< double > & *b,* PCG_DATA ∗ *pcg_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to iteratively solve a symmetric, definite linear system with PCG.

This function iteratively solves a symmetric, definite linear system using the Preconditioned Conjugate Gradient (PCG) method. The PCG algorithm is optimal in terms of efficiency and residual reduction, but only if the linear system is symmetric. PCG will fail if the linear operator is non-symmetric!

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
pcg_dat	pointer to the PCG_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.7 int bicgstab (int(∗)(const Matrix< double > &p, Matrix< double > &Ap, const void ∗data) *matvec,* int(∗)(const Matrix< double > &r, Matrix< double > &z, const void ∗data) *precon,* Matrix< double > & *b,* BiCGSTAB_DATA ∗ *bicg_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to iteratively solve a non-symmetric, definite linear system with BiCGSTAB.

This function iteratively solves a non-symmetric, definite linear system using the Bi-Conjugate Gradient STABilized (BiCGSTAB) method. This is a highly efficient algorithm for solving non-symmetric problems, but will occassionally breakdown and fail. Most common failures are caused by poor preconditioning. Works very well for grid-based linear systems.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
bicg_dat	pointer to the BiCGSTAB_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.8 int cgs (int(∗)(const Matrix< double > &p, Matrix< double > &Ap, const void ∗data) *matvec,* int(∗)(const Matrix< double > &r, Matrix< double > &z, const void ∗data) *precon,* Matrix< double > & *b,* CGS_DATA ∗ *cgs_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to iteratively solve a non-symmetric, definite linear system with CGS.

This function iteratively solves a non-symmetric, definite linear system using the Conjugate Gradient Squared (CGS) method. This is an extremely efficient algorithm for solving non-symmetric problems, but will often breakdown and fail. Most common failures are caused by poor or no preconditioning. Works very will for grid-based linear systems.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
cgs_dat	pointer to the CGS_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.9 int operatorTranspose (int(∗)(const Matrix< double > &v, Matrix< double > &Av, const void ∗data) *matvec,* Matrix< double > & *r,* Matrix< double > & *u,* OPTRANS_DATA ∗ *transpose_dat,* const void ∗ *matvec_data*)

Function that is used to perform transposition of a linear operator and results in a new vector A^T∗r=u.

This function takes a user supplied linear operator and forms the result of that operator transposed and multiplied by a given vector r (A^T∗r=u). Transposition is accomplised by reordering the transpose operator and multiplying the non-transposed operator by a complete set of orthonormal vectors. The end result gives the ith component of the vector u for each operation (u_i = r^T∗A∗i). Here, i is a vector made from the ith column of the identity matrix. If the linear system if sufficiently large, then this operation may take some time.

Parameters

matvec	user supplied linear operator given as an int function
r	vector to be multiplied by the transpose of the operator
u	vector to store the result of the operator transposition (u=A^T∗r)
transpose_dat	pointer to the OPTRANS_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

6.9.4.10 int gcr (int(∗)(const Matrix< double > &x, Matrix< double > &Ax, const void ∗data) *matvec,* int(∗)(const Matrix< double > &r, Matrix< double > &Mr, const void ∗data) *precon,* Matrix< double > & *b,* GCR_DATA ∗ *gcr_dat,* const void ∗ *matvec_data,* const void ∗ *precon_data*)

Function to iteratively solve a non-symmetric, definite linear system with GCR.

This function iteratively solves a non-symmetric, definite linear system using the Generalized Conjugate Residual (GCR) method. Similar to GMRESRP, this algorithm will construct a growing orthonormal basis set that will eventually form the exact solution to the linear system. However, this algorithm is less efficient than GMRESRP and can suffer breakdowns if the linear system is indefinite.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gcr_dat	pointer to the GCR_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.11 int gmresrPreconditioner (const Matrix< double > & *r,* Matrix< double > & *Mr,* const void ∗ *data*)

Function used in conjunction with GMRESR to apply GMRESRP iterations as a preconditioner.

This function is required to take the form of the user supplied preconditioning functions for other iterative methods. However, it cannot be used in conjunction with any other Krylov method. It is only called by the GMRESR function when the preconditioner needs to be applied.

Parameters

r	vector supplied to the preconditioner to operate on
Mr	vector to hold the result of the preconditioning operation
data	void pointer to the GMRESR_DATA data structure

6.9.4.12 int gmresr (int(∗)(const Matrix< double > &x, Matrix< double > &Ax, const void ∗data) *matvec,* int(∗)(const Matrix< double > &r, Matrix< double > &Mr, const void ∗data) *terminal_precon,* Matrix< double > & *b,* GMRESR_DATA ∗ *gmresr_dat,* const void ∗ *matvec_data,* const void ∗ *term_precon_data*)

Function to iteratively solve a non-symmetric, indefinite linear system with GMRESR.

This function iteratively solves a non-symmetric, indefinite linear system using the Generalized Minimum RE↩ Sidual Recursive (GMRESR) method. This algorithm actually uses GCR at the outer iterations, but stabilizes GCR with GMRESRP inner iterations to implicitly form a variable preconditioner to the linear system. As such, this is one of only two methods that inherently includes preconditioning (the other is KMS), without any user supplied preconditioning operator. However, this algorithms is signficantly more computationally expensive than GCR or GMRESRP separately. It should only be used for solving very large or very hard to solve linear systems.

Parameters

matvec	user supplied linear operator given as an int function
terminal_precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmresr_dat	pointer to the GMRESR_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
term_precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (*matvec) (const Matrix<double>& v, Matrix<double> &Av, const void *data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (*terminal_precon) (const Matrix<double>& b, Matrix<double> &Mb, const void *data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.13 int kmsPreconditioner (const Matrix< double > & *r*, Matrix< double > & *Mr*, const void * *data*)

Preconditioner function for the Krylov Multi-Space.

This function is required to take the form of the user supplied preconditioning functions for other iterative methods. However, it cannot be used in conjunction with any other Krylov method. It is only called by the KMS function when the preconditioner needs to be applied.

Parameters

r	vector supplied to the preconditioner to operate on
Mr	vector to hold the result of the preconditioning operation
data	void pointer to the KMS_DATA data structure

6.9.4.14 int krylovMultiSpace (int(*)(const Matrix< double > &x, Matrix< double > &Ax, const void *data) *matvec*, int(*)(const Matrix< double > &r, Matrix< double > &Mr, const void *data) *terminal_precon*, Matrix< double > & *b*, KMS_DATA * *kms_dat*, const void * *matvec_data*, const void * *term_precon_data*)

Function to iteratively solve a non-symmetric, indefinite linear system with KMS.

This function iteratively solves a non-symmetric, indefinite linear system using the Krylov Multi-Space (KMS) method. This algorithm uses GMRESRP at both outer and inner iterations to implicitly form a variable preconditioner to the linear system. As such, this is one of only two methods that inherently includes preconditioning, without any user supplied preconditioning operator (the other being GMRESR). The advantage to this method over GMRESR is that this method is GMRES at its core, and will therefore never breakdown or need to be stabilized. Additionally, you can call this method and set it's max_level parameter (see KMS_DATA) to 0, which will make this algorithm exactly equal to GMRESRP. If the max_level is set to 1, then this algorithm is exactly FGMRES (Saad, 1993) with the GMRES algorithm as a preconditioner. However, you can set max_level higher to precondition the preconditioners with more preconditioners. Thus creating a method with any desired complexity or rate of convergence.

Parameters

matvec	user supplied linear operator given as an int function
terminal_precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
kms_dat	pointer to the KMS_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
term_precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---

int (∗terminal_precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.15 int QRsolve (int(∗)(const Matrix< double > &x, Matrix< double > &Ax, const void ∗data) *matvec,* Matrix< double > & *b,* QR_DATA ∗ *qr_dat,* const void ∗ *matvec_data*)

Function to solve a dense linear operator system using QR factorization.

This function is used to solve a dense linear system using QR factorization. It should only be used if iterative methods are unstable or if the linear system is very dense. There will likely be memory limitations to using this method, since it is assumed that the matrix/operator is dense. This method may also be less efficient because it has to extract the matrix elements from the linear operator. So if the linear operator is large, then the setup cost for this method is high.

Factorization is carried out using Householder Reflections. Each reflection matrix is iteratively applied to the operator and the vector b to convert the linear system to upper triangular. Then, the system is solved using backwards substitution.

Parameters

matvec	user supplied linear operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
qr_dat	pointer to the QR_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

6.9.4.16 int picard (int(∗)(const Matrix< double > &x, Matrix< double > &r, const void ∗data) *res,* int(∗)(const Matrix< double > &x0, Matrix< double > &x, const void ∗data) *evalx,* Matrix< double > & *x,* PICARD_DATA ∗ *picard_dat,* const void ∗ *res_data,* const void ∗ *evalx_data*)

Function to iteratively solve a non-linear system using the Picard or Fixed-Point method.

This function iteratively solves a non-linear system using the Picard method. User supplies a residual function and a weak solution form function. The weak form function is used to approximate the next solution vector for the non-linear system and the residual function is used to determine convergence. User also supplies an initial guess to the non-linear system as a matix x, which will also be used to store the solution. This algorithm is very simple and may not be sufficient to solve complex non-linear systems.

Parameters

res	user supplied function for the non-linear residuals of the system
evalx	user supplied function for the weak form to estimate the next solution
x	user supplied matrix holding the initial guess to the non-linear system
picard_dat	pointer to the PICARD_DATA data structure
res_data	user supplied void pointer to a data structure used for residual evaluations
evalx_data	user supplied void pointer to a data structure used for evaluation of weak form

Note

int (∗res) (const Matrix<double>& x, Matrix<double> &F, const void ∗data)

This is a user supplied function for the non-linear residuals. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix x representing the current non-linear variables. Those variables are used to evaluate the users functions and return the residuals in the matrix F. The void pointer data is a data structure provided by the user to hold information the function may need in order to form the residuals.

---

int (∗evalx) (const Matrix<double>& x0, Matrix<double> &x, const void ∗data)

This is a user supplied function to approximate the next solution vector x based on the previous solution vector x0. The x0 matrix is passed to this function and must be used to edit the entries of x based on the weak form of the problem. The user is free to define any weak form approximation. Void pointer data is the users data structure that may be used to pass additional information into this function in order to evaluate the weak form.
Example Residual: F(x) = x^2 + x - 1 Goal is to make this function equal zero
Example Weak Form: x = 1 - x0^2 Rearrage residual to form a weak solution

---

6.9.4.17 int jacvec (const Matrix< double > & *v*, Matrix< double > & *Jv*, const void ∗ *data*)

Function to form a linear operator of a Jacobian matrix used along with the PJFNK method.

This function is used in conjunction with the PJFNK routine to form a linear operator that a Krylov method can operate on. This linear operator is formed from the current residual vector of the non-linear iteration in PJFNK using a finite difference approximation.
Jacobian Linear Operator: J∗v = (F(x_k + eps∗v) - F(x_k)) / eps

Parameters

v	vector to be multiplied by the Jacobian matrix
Jv	storage vector for the result of the Jacobi-vector product
data	void pointer to the PJFNK_DATA data structure holding solver information

6.9.4.18 int backtrackLineSearch (int(∗)(const Matrix< double > &x, Matrix< double > &F, const void ∗data) *feval,* Matrix< double > & *Fkp1,* Matrix< double > & *xkp1,* Matrix< double > & *pk,* double *normFk,* BACKTRACK_DATA ∗ *backtrack_dat,* const void ∗ *feval_data*)

Function to perform a Backtracking Line Search operation to smooth out convergence of PJFNK.

This function performs a simple backtracking line search operation on the residuals from the PJFNK method. The step size of the non-linear iteration is checked against a level of tolerance for residual reduction, then adjusted down if necessary. This method always starts out with the maximum allowable step size. If the largest step size is fine, then the algorithm does nothing. Otherwise, it iteratively adjusts the step size down, until a suitable step is found. In the case that no suitable step is found, this algorithm will report failure to the PJFNK method and PJFNK will decide whether to continue trying to find a global minimum or report that it is stuck in a local minimum.

Parameters

feval	user supplied residual function for the non-linear system
Fkp1	vector holding the residuals for the next non-linear step
xkp1	vector holding the solution for the next non-linear step
pk	vector holding the current non-linear search direction
normFk	value of the current non-linear residual
backtrack_dat	pointer to the BACKTRACK_DATA data structure
feval_data	user supplied void pointer to the data structure needed for residual evaluation

Note

int (∗feval) (const Matrix<double>& x, Matrix<double> &F, const void ∗data)

This is a user supplied function for the non-linear residuals. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix x representing the current non-linear variables. Those variables are used to evaluate the users functions and return the residuals in the matrix F. The void pointer data is a data structure provided by the user to hold information the function may need in order to form the residuals.

6.9.4.19 int pjfnk (int(∗)(const Matrix< double > &x, Matrix< double > &F, const void ∗data) *res,* int(∗)(const Matrix< double > &r, Matrix< double > &p, const void ∗data) *precon,* Matrix< double > & *x,* PJFNK_DATA ∗ *pjfnk_dat,* const void ∗ *res_data,* const void ∗ *precon_data*)

Function to perform the PJFNK algorithm to solve a non-linear system of equations.

This function solves a non-linear system of equations using the Preconditioned Jacobian- Free Newton-Krylov (P↩ JFNK) algorithm. Each non-linear step of this method results in a linear sub-problem that is solved iteratively with one of the Krylov methods in the krylov_method enum. User must supplied a residual function that computes the non-linear residuals of the system given the current state of the variables x. Additionally, the user must also supplied an initial guess to the non-linear system. Optionally, the user may supply a preconditioning function for the linear sub-problem.
Basic Steps: (i) Calc F(x_k), (ii) Solve J(x_k)∗s_k=-F(x_k) for s_k, (iii) Form x_kp1 = x_k + s_k

Parameters

res	user supplied residual function for the non-linear system
precon	user supplied preconditioning function for the linear sub-problems
x	user supplied initial guess and storage location of the solution
pjfnk_dat	pointer to the PJFNK_DATA data structure
res_data	user supplied void pointer to data structure used in residual function
precon_data	user supplied void pointer to data structure used in preconditioning function

Note

int (*res) (const Matrix<double>& x, Matrix<double> &F, const void *data)

This is a user supplied function for the non-linear residuals. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix x representing the current non-linear variables. Those variables are used to evaluate the users functions and return the residuals in the matrix F. The void pointer data is a data structure provided by the user to hold information the function may need in order to form the residuals.

---

int (*precon) (const Matrix<double>& b, Matrix<double> &Mb, const void *data)

This is a user supplied function for a preconditioning operator. It has the same form as the linear operators from the Krylov methods and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the jacvec linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.9.4.20 int NumericalJacobian (int(*)(const Matrix< double > &x, Matrix< double > &F, const void *user_data) *Func*, const Matrix< double > & *x*, Matrix< double > & *J*, int *Nx*, int *Nf*, NUM_JAC_DATA * *jac_dat*, const void * *user_data*)

Function to form a full numerical Jacobian matrix from a given non-linear function.

This function uses finite differences to form a full rank Jacobian matrix for a user supplied non-linear function. The Jacobian matrix will be formed at the current state of the non-linear variables x and stored in a full matrix J. Integers Nx and Nf are used to determine the size of the Jacobian matrix.

Parameters

Func	user supplied function for evaluation of the non-linear system
x	matrix holding the current value of the non-linear variables
J	matrix that will store the numerical Jacobian result
Nx	number of non-linear variables in the system
Nf	number of non-linear functions in the system
jac_dat	pointer to the NUM_JAC_DATA data structure
user_data	user supplied void pointer to a data structure used in the non-linear function

6.9.4.21 int LARK_TESTS ( )

Function that runs a variety of tests on all the functions in LARK.

This function runs a variety of tests on the linear and non-linear methods developed in LARK. It can be called from the UI.

6.10 macaw.h File Reference

MAtrix CAlculation Workspace.

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include <exception>
#include "error.h"
```

Classes

- class Matrix< T >

  Templated C++ Matrix Class Object (click Matrix to go to function definitions)

Macros

- #define M_PI 3.14159265358979323846264338327950288

  Value of PI with double precision.

Functions

- int MACAW_TESTS ()

  Function to run the MACAW tests.

6.10.1 Detailed Description

MAtrix CAlculation Workspace.

macaw.cpp

This is a small C++ library that faciltates the use and construction of real matrices using vector objects. The Matrix class is templated so that users are able to work with matrices of any type including, but not limited to: (i) doubles, (ii) ints, (iii) floats, and (iv) even other matrices! Routines and functions are defined for Dense matrix operations. As a result, we typically only use Column Matrices (or Vectors) when doing any actual simulations. However, the development of this class was integral to the development and testing of the Sparse matrix operators in lark.h.

While the primary goal of this object was to define how to operate on real matrices, we could extend this idea to complex matrices as well. For this, we could develop objects that represent imaginary and complex numbers and then create a Matrix of those objects. For this reason, the matrix operations here are all templated to abstract away the specificity of the type of matrix being operated on.

Author

Austin Ladshaw

Date

01/07/2014

Copyright

6.10.2 Macro Definition Documentation

6.10.2.1 #define M_PI 3.14159265358979323846264338327950288

Value of PI with double precision.

6.10.3 Function Documentation

6.10.3.1 int MACAW_TESTS ( )

Function to run the MACAW tests.

This function is callable from the UI and is used to run several algorithm tests for the Matrix objects. This test should never report any errors.

6.11 magpie.h File Reference

Multicomponent Adsorption Generalized Procedure for Isothermal Equilibria.

```
#include "lmcurve.h"
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include "error.h"
#include "lark.h"
```

Classes

- struct GSTA_DATA

  GSTA Data Structure.
- struct mSPD_DATA

  MSPD Data Structure.
- struct GPAST_DATA

  GPAST Data Structure.
- struct SYSTEM_DATA

  System Data Structure.
- struct MAGPIE_DATA

  MAGPIE Data Structure.

Macros

- #define DBL_EPSILON 2.2204460492503131e-016

  Machine precision value used for approximating gradients.
- #define Z 10.0

  Surface coordination number used in the MSPD activity model.
- #define A 3.13E+09

  Corresponding van der Waals standard area for our coordination number (cm^2/mol)
- #define V 18.92

  Corresponding van der Waals standard volume for our coordination number (cm^3/mol)
- #define Po 100.0

  Standard State Pressure - Units: kPa.
- #define R 8.3144621

  Gas Constant - Units: J/(K∗mol) = kB ∗ Na.
- #define Na 6.0221413E+23

  Avagadro's Number - Units: molecules/mol.
- #define kB 1.3806488E-23

  Boltzmann's Constant - Units: J/K.
- #define shapeFactor(v_i) (((Z - 2) ∗ v_i) / (Z ∗ V)) + (2 / Z)

  This macro replaces all instances of shapeFactor(#) with the following single line calculation.
- #define lnKo(H, S, T) -(H / (R ∗ T)) + (S / R)

  This macro calculates the natural log of the dimensionless isotherm parameter.
- #define He(qm, K1, m) (qm ∗ K1) / (m ∗ Po)

  This macro calculates the Henry's Coefficient for the ith component.

Functions

- double qo (double po, const void ∗data, int i)

  Function computes the result of the GSTA isotherm for the ith species.
- double dq_dp (double p, const void ∗data, int i)

  Function computes the derivative of the GSTA model with respect to partial pressure.
- double q_p (double p, const void ∗data, int i)

  Function computes the ratio between the adsorbed amount and partial pressure for the GSTA isotherm.
- double PI (double po, const void ∗data, int i)

  Function computes the spreading pressure integral of the ith species.
- double Qst (double po, const void ∗data, int i)

  Function computes the heat of adsorption based on the ith species GSTA parameters.
- double eMax (const void ∗data, int i)

  Function to approximate the maximum lateral energy term for the ith species.
- double lnact_mSPD (const double ∗par, const void ∗data, int i, volatile double PI)

  Function to evaluate the MSPD activity coefficient for the ith species.
- double grad_mSPD (const double ∗par, const void ∗data, int i)

  Function to approximate the derivative of the MSPD activity model with spreading pressure.
- double qT (const double ∗par, const void ∗data)

  Function to calculate the total adsorbed amount (mol/kg) for the mixed surface phase.
- void initialGuess_mSPD (double ∗par, const void ∗data)

  Function to provide an initial guess to the unknown parameters being solved for in GPAST.
- void eval_po_PI (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function used with lmfit to evaluate the reference state pressure of a species based on spreading pressure.
- void eval_po_qo (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

Function used with lmfit to evaluate the reference state pressure of a species based on that species isotherm.

- void eval_po (const double *par, int m_dat, const void *data, double *fvec, int *info)

  Function used with lmfit to evaluate the reference state pressure of a species based on a sub-system.

- void eval_eta (const double *par, int m_dat, const void *data, double *fvec, int *info)

  Function used with lmfit to evaluate the binary interaction parameters for each unique species pair.

- void eval_GPAST (const double *par, int m_dat, const void *data, double *fvec, int *info)

  Function used with lmfit to solve the GPAST system of equations.

- int MAGPIE (const void *data)

  Function to call all sub-routines to solve a MAGPIE/GPAST problem at a given temperature and pressure.

- int MAGPIE_SCENARIOS (const char *inputFileName, const char *sceneFileName)

  Function to perform a series of MAGPIE simulations based on given input files.

6.11.1 Detailed Description

Multicomponent Adsorption Generalized Procedure for Isothermal Equilibria.

magpie.cpp

This file contains all functions and routines associated with predicting isothermal adsorption equilibria from only single component isotherm information. The basis of the model is the Adsorbed Solution Theory developed by Myers and Prausnitz (1965). Added to that base model is a procedure by which we can predict the non-idealities present at the surface phase by solving a closed system of equations involving the activity model.

For more details on this procedure, check out our publication in AIChE where we give a fully feature explaination of our Generalized Predictive Adsorbed Solution Theory (GPAST).

Reference: Ladshaw, A., Yiacoumi, S., and Tsouris, C., "A generalized procedure for the prediction of multicomponent adsorption equilibria", AIChE J., vol. 61, No. 8, p. 2600-2610, 2015.

MAGPIE represents a special case of the more general GPAST procedure, wherin the isotherm for each species is respresent by the GSTA isotherm (see gsta_opt.h) and the activity model for non-ideality at the adsorbent surface is a Modified Spreading Pressure Dependent (MSPD) model. See the above paper reference for more details.

Author

Austin Ladshaw

Date

12/17/2013

Copyright

6.11.2 Macro Definition Documentation

6.11.2.1 #define DBL_EPSILON 2.2204460492503131e-016

Machine precision value used for approximating gradients.

6.11.2.2 #define Z 10.0

Surface coordination number used in the MSPD activity model.

6.11.2.3 #define A 3.13E+09

Corresponding van der Waals standard area for our coordination number (cm^2/mol)

6.11.2.4 #define V 18.92

Corresponding van der Waals standard volume for our coordination number (cm^3/mol)

6.11.2.5 #define Po 100.0

Standard State Pressure - Units: kPa.

6.11.2.6 #define R 8.3144621

Gas Constant - Units: J/(K∗mol) = kB ∗ Na.

6.11.2.7 #define Na 6.0221413E+23

Avagadro's Number - Units: molecules/mol.

6.11.2.8 #define kB 1.3806488E-23

Boltzmann's Constant - Units: J/K.

6.11.2.9 #define shapeFactor(*v_i*) (((Z - 2) ∗ v_i) / (Z ∗ V)) + (2 / Z)

This macro replaces all instances of shapeFactor(#) with the following single line calculation.

6.11.2.10 #define lnKo(*H, S, T*) -(H / (R ∗ T)) + (S / R)

This macro calculates the natural log of the dimensionless isotherm parameter.

6.11.2.11 #define He(*qm, K1, m*) (qm ∗ K1) / (m ∗ Po)

This macro calculates the Henry's Coefficient for the ith component.

6.11.3 Function Documentation

6.11.3.1 double qo (double *po,* const void ∗ *data,* int *i*)

Function computes the result of the GSTA isotherm for the ith species.

This function just computes the result of the GSTA isotherm model for the ith species given the partial pressure po.

Parameters

po	partial pressure in kPa at which to evaluate the GSTA model
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.11.3.2 double dq_dp (double *p*, const void * *data*, int *i*)

Function computes the derivative of the GSTA model with respect to partial pressure.

This function just computes the result of the derivative of GSTA isotherm model for the ith species at the given the partial pressure p.

Parameters

p	partial pressure in kPa at which to evaluate the GSTA model
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.11.3.3 double q_p (double *p*, const void * *data*, int *i*)

Function computes the ratio between the adsorbed amount and partial pressure for the GSTA isotherm.

This function just computes the ratio between the adsorbed amount q (mol/kg) and the partial pressure p (kPa) at the given partial pressure. If p == 0, then this function returns the Henry's Law constant for the isotherm of the ith species.

Parameters

p	partial pressure in kPa at which to evaluate the GSTA model
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.11.3.4 double PI (double *po*, const void * *data*, int *i*)

Function computes the spreading pressure integral of the ith species.

This function uses an analytical solution to the spreading pressure integral with the GSTA isotherm to evaluate and return the value computed by that integral equation.

Parameters

po	partial pressure in kPa at which to evaluate the lumped spreading pressure
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.11.3.5 double Qst (double *po*, const void * *data*, int *i*)

Function computes the heat of adsorption based on the ith species GSTA parameters.

This function computes the isosteric heat of adsorption (J/mol) for the GSTA parameters of the ith species.

Parameters

po	partial pressure in kPa at which to evaluate the heat of adsorption
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.11.3.6 double eMax (const void * *data*, int *i*)

Function to approximate the maximum lateral energy term for the ith species.

The function attempts to approximate the maximum lateral energy term for the ith species. This is not a true maximum, but a cheaper estimate. Value being computed is used to shift the geometric mean and formulate the average cross-lateral energy term between species i and j.

6.11.3.7 double lnact_mSPD (const double * *par*, const void * *data*, int *i*, volatile double *PI*)

Function to evaluate the MSPD activity coefficient for the ith species.

This function will return the natural log of the ith species activity coefficient using the Modified Spreading Pressure Dependent (MSPD) activity model. The par argument holds the variable values being solved for by GPAST and their contents will change depending on whether we are doing a forward or reverse evaluation. This function should not be called by the user and will only be called when needed in the GPAST routine.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure
i	ith species that we want to calculate the activity coefficient for
PI	lumped spreading pressure term used in gradient estimations

6.11.3.8 double grad_mSPD (const double * *par*, const void * *data*, int *i*)

Function to approximate the derivative of the MSPD activity model with spreading pressure.

This function returns a 2nd order, finite different approximation of the derivative of the MSPD activity model with the spreading pressure. The par argument will either hold the current iterates estimate of spreading pressure or should be passed as null. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure
i	ith species for which we will approximate the activty model gradient

6.11.3.9 double qT (const double * *par*, const void * *data*)

Function to calculate the total adsorbed amount (mol/kg) for the mixed surface phase.

This function will uses the obtained system parameters from par and estimate the total amount of gases adsorbed to the surface in mol/kg. The user does not need to call this function, since this result will be stored in the SYSTEM_DATA structure.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure

6.11.3.10 void initialGuess_mSPD (double * *par*, const void * *data*)

Function to provide an initial guess to the unknown parameters being solved for in GPAST.

This function intends to provide an initial guess for the unknown values being solved for in the GPAST system. Depending on what type of solve is requested, this algorithm will provide a guess for the adsorbed or gas phase composition.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure

6.11.3.11 void eval_po_PI (const double * *par*, int *m_dat*, const void * *data*, double * *fvec*, int * *info*)

Function used with lmfit to evaluate the reference state pressure of a species based on spreading pressure.

This function is used inside of the MSPD activity model to calculate the reference state pressure of a particular species at a given spreading pressure for the system. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.11.3.12 void eval_po_qo (const double * *par*, int *m_dat*, const void * *data*, double * *fvec*, int * *info*)

Function used with lmfit to evaluate the reference state pressure of a species based on that species isotherm.

This function is used to evaluate the partial pressure or reference state pressure for a particular species given single-component adsorbed amount. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations

Parameters

data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.11.3.13 void eval_po (const double * *par*, int *m_dat*, const void * *data*, double * *fvec*, int * *info*)

Function used with lmfit to evaluate the reference state pressure of a species based on a sub-system.

This function is used to approximate reference state pressures based on the spreading pressure of a sub-system in GPAST. The sub-system will be one of the unique binary systems that exist in the overall mixed gas system. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.11.3.14 void eval_eta (const double * *par*, int *m_dat*, const void * *data*, double * *fvec*, int * *info*)

Function used with lmfit to evaluate the binary interaction parameters for each unique species pair.

This function is used to estimate the binary interaction parameters for all species pairs in a given sub-system. Those parameters are then stored for later used when evaluating the activity coefficients for the overall mixture. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.11.3.15 void eval_GPAST (const double * *par*, int *m_dat*, const void * *data*, double * *fvec*, int * *info*)

Function used with lmfit to solve the GPAST system of equations.

This function is used after having calculated and stored all necessary information to solve a closed form GPAST system of equations. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations

Parameters

data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.11.3.16 int MAGPIE (const void * *data*)

Function to call all sub-routines to solve a MAGPIE/GPAST problem at a given temperature and pressure.

This is the function that a typical user will want to incorporate into their own codes when evaluating adsorption of a gas mixture. Prior to calling this function, all required structures and information in the MAGPIE_DATA structure must have been properly initialized. After this function has completed it's operations, it will return an integer used to denote a success or failure of the routine. Integers 0, 1, 2, and 3 all denote success. Anything else is considered a failure.

To setup the MAGPIE_DATA structure correctly, you must reserve space for all vector objects based on the number of gas species in the mixture. In general, you only need to reserve space for the adsorbing species. However, you can also reserve space for non-adsorbing species, but you MUST give a gas/adsorbed mole fraction of the non-adsorbing species 0.0 so that the routine knows to ignore them (very important)!

After setting up the memory for the vector objects, you can intialize information specific to the simulation you want to request. The number of species (N), total pressure (PT) and gas temperature (T) must always be given. You can neglect the non-idealities of the surface phase by setting the Ideal bool to true. This will result in faster calculations, because MAGPIE will just revert down to the Ideal Adsorbed Solution Theory (IAST).

The Recover bool will denote whether we are doing a forward or reverse GPAST evaluation. Forward evaluation is for solving for the composition of the adsorbed phase given the composition of the gas phase (Recover = false). Reverse evaluation is for solve for the composition of the gas phase given the composition of the adsorbed phase (Recover = true).

For a reverse evaluation (Recover = true) you will also need to stipulate whether or not there is a carrier gas (Carrier = true or false). A carrier gas is considered any non-adsorbing species that may be present in the gas phase and contributing to the total pressure in the system.

The parameters that must be initialized for all species include all GSTA_DATA parameters and the van der Waals volume parameter (v) in the mSPD_DATA structure. For non-adsorbing species, you can ignore these parameters, but need to set the sites (m) from GSTA_DATA to 1. GPAST cannot run any evaluations without these parameters being set properly AND set in the same order for all species (i.e., make sure that gpast_dat[i].qmax corresponds to mspd_dat[i].v and so on).

Lastly, you need to give either the gas phase or adsorbed phase mole fractions, depending on whether you are going to run a forward or reverse evaluation, respectively. For a forward evaluation, provide the gas mole fractions (y) in GPAST_DATA for each species (non-adsorbing species should have this value set to 0.0). For a reverse evaluation, provide the adsorbed mole fractions (x) in GPAST_DATA for each species, as well as the total adsorbed amount (qT) in SYSTEM_DATA. Again, non-adsorbing species should have their respective phase mole fractions set to 0.0 to exclude them from the simulation. Additionally, if there are non-adsorbing species present, then the Carrier bool in SYSTEM_DATA must be set to true.

Parameters

data	void pointer for the MAGPIE_DATA data structure holding all necessary information

6.11.3.17 int MAGPIE_SCENARIOS (const char * *inputFileName,* const char * *sceneFileName*)

Function to perform a series of MAGPIE simulations based on given input files.

This function is callable from the UI and is used to perform a series of isothermal equilibria evaluations using the MAGPIE routines. There are two input files that must be provided: (i) inputFileName - containing parameter information for the species and (ii) sceneFileName - containing information for each MAGPIE simulation. Each of these files have a specific structure (see below). NOTE: this may change in future versions.

inputFileName Text File Structure:

Integer for Number of Adsorbing Species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat above for all n sites in species i)
(repeat above for all species i)

Example Input File:

5
17.1
5.8797
1
-20351.9 -81.8369
16.2
5.14934
1
-16662.7 -74.4766
19.7
9.27339
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
13.25
4.59144
1
-13418.5 -84.888
18.0
10.0348
1
-20640.4 -72.6119
(The above input file gives the parameter information for 5 adsorbing species)

sceneFileName Text File Structure:

Integer Flag to mark Forward (0) or { Reverse (1) evaluations }
Number of Simulations to Run
Total Pressure (kPa) [tab] Temperature (K) { [tab] Total Adsorption (mol/kg) [tab] Carrier Gas Flag (0=false, 1=true) }
Gas/Adsorbed Mole Fractions for each species in the order given in prior file (tab separated)
(repeat above for all simulations desired)
NOTE: only provide the Total Adsorption and Carrier Flag if doing Reverse evaluations!

Example Scenario File 1:

0
4
0.65 303.15
0.364 0.318 0.318
3.25 303.15
0.371 0.32 0.309
6.85 303.15
0.388 0.299 0.313
13.42 303.15
0.349 0.326 0.325
(The above scenario file is for 4 forward evaluations/simulations for a 3-adsorbing species system)

Example Scenario File 2:

1
4
0.65 303.15 5.4 0
0.364 0.318 0.318
3.25 303.15 7.7 0
0.371 0.32 0.309
6.85 303.15 9.8 0
0.388 0.299 0.313
13.42 303.15 10.4 0
0.349 0.326 0.325
(The above scenario file is for 4 reverse evaluations/simulations for a 3-adsorbing species system and no carrier gas)

6.12 mola.h File Reference

Molecule Object Library from Atoms.

```
#include <ctype.h>
#include "eel.h"
```

Classes

- class Molecule

  C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

Macros

- #define M_PI 3.14159
- #define SphereVolume(r) ((4.0/3.0)∗M_PI∗r∗r∗r)
- #define SphereArea(r) (4.0∗M_PI∗r∗r)

Enumerations

- enum valid_phase {
  SOLID, LIQUID, AQUEOUS, GAS,
  PLASMA, ADSORBED, OTHER }

Functions

- int MOLA_TESTS ()

  Function to run the MOLA tests.

6.12.1 Detailed Description

Molecule Object Library from Atoms.

mola.cpp

This file contains a C++ Class for creating Molecule objects from the Atom objects that were defined in eel.h. Molecules can be created and registered from basic information or can be registered from a growing list of pre-registered molecules that are accessible by name/formula.

Registered Molecules are are known and defined prior to runtime. They have a charge, energy characteristics, phase, name, and formula that they are recongized by. The formula is used to create the atoms that they are made from. If some information is incomplete, it must be specified as to what information is missing (i.e. denote whether the formation energies are known).

Formation energies are used to determine stability/dissociation/acidity equilibrium constants during runtime. If the formation energies are unknown, then the equilibrium constants must be given to a reaction object on when it is initialized.

The molecule formula's are given as strings which are parsed in the constructor to determine what atoms from the EEL files will be registered and used. Note, you will be able to build molecules from an input file, but the library molecules here are ready to be used in applications and require no more input other that the molecule's formula.

List of Currently Registered Molecules

Ag (s)
Ag + (aq)
AgBr (s)
AgCl (s)
AgI (s)
Ag2S (s)
AgOH (aq)
Ag(OH)2 - (aq)
AgCl (aq)
AgCl2 - (aq)
Al (s)
Al 3+ (aq)
AlOH 2+ (aq)
Al(OH)2 + (aq)
Al(OH)3 (aq)
Al(OH)4 - (aq)
Al2O3 (s)
AlOOH (s)
Al(OH)3 (s)
Al2Si2(OH)4 (s)
As (s)
AsO4 3- (aq)
Ba 2+ (aq)
BaSO4 (s)
BaCO3 (s)
Be 2+ (aq)

Be(OH)2 (s)
Be3(OH)3 3+ (aq)
B(OH)4 - (aq)
Br2 (l)
Br2 (aq)
Br - (aq)
BrO - (aq)
CO3 2- (aq)
Cl - (aq)
CaCl2 (aq)
CaAl2Si2O8 (s)
C (s)
CO2 (g)
CH4 (g)
CH4 (aq)
CH3OH (aq)
CN - (aq)
CH3COOH (aq)
CH3COO - (aq)
C2H5OH (aq)
Ca 2+ (aq)
CaOH + (aq)
Ca(OH)2 (aq)
Ca(OH)2 (s)
CaCO3 (s)
CaMg(CO3)2 (s)
CaSiO3 (s)
CaSO4 (s)
CaSO4(H2O)2 (s)
Ca5(PO4)3OH (s)
Cd 2+ (aq)
Cd(OH) + (aq)
Cd(OH)3 - (aq)
Cd(OH)4 2- (aq)
Cd(OH)2 (aq)
CdO (s)
Cd(OH)2 (s)
CdCl + (aq)
CdCl2 (aq)
CdCl3 - (aq)
CdCO3 (s)
Cl2 (g)
Cl2 (aq)
ClO - (aq)
ClO2 (aq)
ClO2 - (aq)
ClO3 - (aq)
ClO4 (aq)
Co (s)
Co 2+ (aq)
Co 3+ (aq)
CoOH + (aq)
Co(OH)2 (aq)
Co(OH)3 - (aq)
Co(OH)2 (s)
CoO (s)
Co3O4 (s)
Cr (s)

Cr 2+ (aq)
Cr 3+ (aq)
CrOH 2+ (aq)
Cr(OH)2 + (aq)
Cr(OH)3 (aq)
Cr(OH)4 - (aq)
Cr2O3 (s)
CrO4 2- (aq)
Cr2O7 2- (aq)
Cu (s)
Cu + (aq)
Cu 2+ (aq)
CuOH + (aq)
Cu(OH)2 (aq)
Cu(OH)3 - (aq)
Cu(OH)4 2- (aq)
CuS (s)
Cu2S (s)
CuO (s)
CuCO3Cu(OH)2 (s)
(CuCO3)2Cu(OH)2 (s)
F2 (g)
F - (aq)
Fe (s)
Fe 2+ (aq)
FeOH + (aq)
Fe(OH)2 (aq)
Fe(OH)3 - (aq)
Fe 3+ (aq)
FeOH 2+ (aq)
Fe(OH)2 + (aq)
Fe(OH)3 (aq)
Fe(OH)4 - (aq)
Fe2(OH)2 4+ (aq)
FeS2 (s)
FeO (s)
Fe(OH)2 (s)
Fe2O3 (s)
Fe3O4 (s)
FeOOH (s)
Fe(OH)3 (s)
FeCO3 (s)
Fe2SiO4 (s)
H2O (l)
H + (aq)
H2CO3 (aq)
HCO3 - (aq)
HNO3 (aq)
HCl (aq)
H3AsO4 (aq)
H2AsO4 - (aq)
HAsO4 2- (aq)
H2AsO3 - (aq)
H3BO3 (aq)
HBrO (aq)
HCOOH (aq)
HCOO - (aq)
HCN (aq)

HClO (aq)
HCoO2 - (aq)
HCrO4 - (aq)
HCuO2 - (aq)
HF (aq)
HF2 - (aq)
H2 (g)
H2 (aq)
H2O2 (aq)
HO2 - (aq)
H2O (g)
Hg (l)
Hg2 2+ (aq)
Hg 2+ (aq)
HgOH + (aq)
Hg(OH)2 (aq)
Hg(OH)3 - (aq)
Hg2Cl2 (s)
HgO (s)
HgS (s)
HgI2 (s)
HgCl + (aq)
HgCl2 (aq)
HgCl3 - (aq)
HgCl4 2- (aq)
HgOH + (aq)
Hg(OH)2 (aq)
HgO2 - (aq)
HIO (aq)
HIO3 (aq)
HNO2 (aq)
HPO4 2- (aq)
H2PO4 - (aq)
H3PO4 (aq)
H2S (g)
H2S (aq)
HS - (aq)
HSO3 - (aq)
H2SO3 (aq)
HSO4 - (aq)
H2SO4 (aq)
HSeO3 - (aq)
H2SeO3 (aq)
HSeO4 - (aq)
H4SiO4 (aq)
HV2O5 - (aq)
H4VO4 + (aq)
H3VO4 (aq)
H2VO4 - (aq)
HVO4 2- (aq)
H4VO4(C2O4)2 3- (aq)
H4VO4C2O4 - (aq)
H2V10O28 4- (aq)
HV10O28 5- (aq)
HV2O7 3- (aq)
I2 (s)
I2 (aq)
I - (aq)

I3 - (aq)
IO - (aq)
IO3 - (aq)
KAl3Si3O10(OH)2 (s)
K + (aq)
Mg(OH)2 (aq)
Mg5Al2Si3O10(OH)8 (s)
Mg (s)
Mg 2+ (aq)
MgOH + (aq)
Mg(OH)2 (s)
Mn (s)
Mn 2+ (aq)
Mn(OH)2 (s)
Mn3O4 (s)
MnOOH (s)
MnO2 (s)
MnCO3 (s)
MnS (s)
MnSiO3 (s)
NaHCO3 (aq)
NaCO3 - (aq)
Na + (aq)
NaCl (aq)
NaOH (aq)
NO3 - (aq)
NH3 (aq)
NaAlSiO3O8 (s)
NH2CH2COOH (aq)
NH2CH2COO - (aq)
N2 (g)
N2O (g)
NH3 (g)
NH4 + (aq)
NO2 - (aq)
Ni 2+ (aq)
NiOH + (aq)
Ni(OH)2 (aq)
Ni(OH)3 - (aq)
NiO (s)
NiS (s)
OH - (aq)
O2 (g)
O2 (aq)
O3 (g)
P (s)
PO4 3- (aq)
Pb (s)
Pb 2+ (aq)
PbOH + (aq)
Pb(OH)2 (aq)
Pb(OH)3 - (aq)
Pb(OH)4 2- (aq)
Pb(OH)2 (s)
PbO (s)
PbO2 (s)
Pb3O4 (s)
PbS (s)

PbSO4 (s)
PbCO3 (s)
S (s)
SO2 (g)
SO3 (g)
S 2- (aq)
SO3 2- (aq)
SO4 2- (aq)
Se (s)
SeO3 2- (aq)
SeO4 2- (aq)
Si (s)
SiO2 (s)
Sr 2+ (aq)
SrOH + (aq)
SrCO3 (s)
SrSO4 (s)
UO2 2+ (aq)
UO2NO3 + (aq)
UO2(NO3)2 (aq)
UO2OH + (aq)
UO2(OH)2 (aq)
UO2(OH)3 - (aq)
UO2(OH)4 2- (aq)
(UO2)2OH 3+ (aq)
(UO2)2(OH)2 2+ (aq)
(UO2)3(OH)4 2+ (aq)
(UO2)3(OH)5 + (aq)
(UO2)3(OH)7 - (aq)
(UO2)4(OH)7 + (aq)
UO2CO3 (aq)
UO2(CO3)2 2- (aq)
UO2(CO3)3 4- (aq)
UO2Cl + (aq)
UO2Cl2 (aq)
UO2Cl3 - (aq)
UO2SO4 (aq)
UO2(SO4)2 2- (aq)
VO 2+ (aq)
VOOH + (aq)
VO(OH)2 (s)
V2O4 (s)
(VO)2(OH)2 + (aq)
VOF + (aq)
VOF2 (aq)
VOF3 - (aq)
VOF4 2- (aq)
VOCl + (aq)
VOSO4 (aq)
VO(C2O4)2 2- (aq)
VOOHC2O4 - (aq)
VOCH3COO + (aq)
VO(CH3COO)2 (aq)
VOCO3 (aq)
VOOHCO3 - (aq)
V4O9 2- (aq)
VO2 + (aq)
VO4 3- (aq)

V2O5 (s)
V10O28 6- (aq)
V2O7 4- (aq)
V4O12 4- (aq)
VO2SO4 - (aq)
VO2OHCO3 2- (aq)
VO2(CO3)2 3- (aq)
Zn (s)
Zn 2+ (aq)
ZnOH + (aq)
Zn(OH)2 (aq)
Zn(OH)3 - (aq)
Zn(OH)4 2- (aq)
Zn(OH)2 (s)
ZnCl + (aq)
ZnCl2 (aq)
ZnCl3 - (aq)
ZnCl4 2- (aq)
ZnCO3 (s)
Those registered molecules follow a strict naming convention by which they can be recognized (see below)...

Naming Convention

Plus (+) and minus (-) charges are denoted by the numeric value of the charge followed by a + or - sign, respectively (e.g. UO2(CO3)3 4- (aq))

The phase is always denoted last and will be marked as (l) for liquid, (s) for solid, (aq) for aqueous, and (g) for gas (see above).

When registering a molecule that is not in the library, you must also provide a linear formula during construction or registration. This is needed so that the string parsing is easier to handle when the molecule subsequently registers the necessary atoms. (e.g. UO2(CO3)3 = UO2C3O9 or UO11C3).

Author

Austin Ladshaw

Date

02/24/2014

Copyright

6.12.2 Macro Definition Documentation

6.12.2.1 #define M_PI 3.14159

6.12.2.2 #define SphereVolume(*r*) ((4.0/3.0)∗M_PI∗r∗r∗r)

6.12.2.3 #define SphereArea(*r*) (4.0∗M_PI∗r∗r)

6.12.3 Enumeration Type Documentation

6.12.3.1 enum valid_phase

Enumerator

SOLID

LIQUID
AQUEOUS
GAS
PLASMA
ADSORBED
OTHER

6.12.4 Function Documentation

6.12.4.1 int MOLA_TESTS ( )

Function to run the MOLA tests.

This function is callable from the UI and is used to run several algorithm tests for the Molecule objects. This test should never report any errors.

6.13 monkfish.h File Reference

Multi-fiber wOven Nest Kernel For Interparticle Sorption History.

```
#include "dogfish.h"
```

Classes

- struct MONKFISH_PARAM
  Data structure for species specific information and parameters.
- struct MONKFISH_DATA
  Primary data structure for running MONKFISH.

Functions

- double default_porosity (int i, int l, const void ∗user_data)
  Default porosity function for MONKFISH.
- double default_density (int i, int l, const void ∗user_data)
  Default density function for MONKFISH.
- double default_interparticle_diffusion (int i, int l, const void ∗user_data)
  Default interparticle diffusion function.
- double default_monk_adsorption (int i, int l, const void ∗user_data)
  Default adsorption strength function.
- double default_monk_equilibrium (int i, int l, const void ∗user_data)
  Default equilibirium adsorption function in mg/g.
- double default_monkfish_retardation (int i, int l, const void ∗user_data)
  Default retardation coefficient function.
- double default_exterior_concentration (int i, const void ∗user_data)
  Default exterior concentratio function.
- double default_film_transfer (int i, const void ∗user_data)
  Default film mass transfer function.

- int setup_MONKFISH_DATA (FILE *file, double(*eval_porosity)(int i, int l, const void *user_data), double(*eval_density)(int i, int l, const void *user_data), double(*eval_ext_diff)(int i, int l, const void *user_data), double(*eval_adsorb)(int i, int l, const void *user_data), double(*eval_retard)(int i, int l, const void *user_data), double(*eval_ext_conc)(int i, const void *user_data), double(*eval_ext_film)(int i, const void *user_data), double(*dog_diffusion)(int i, int l, const void *user_data), double(*dog_ext_film)(int i, const void *user_data), double(*dog_surf_conc)(int i, const void *user_data), const void *user_data, MONKFISH_DATA *monk_dat)

  Setup function to allocate memory and setup function pointers for the MONKFISH simulation.
- int MONKFISH_TESTS ()

  Function to run tests on the MONKFISH algorithms.

6.13.1 Detailed Description

Multi-fiber wOven Nest Kernel For Interparticle Sorption History.

monkfish.cpp

This file contains structures and functions associated with modeling the sorption characteristics of woven fiber bundles used to recover uranium from seawater. It is coupled with the DOGFISH kernel that determines the sorption of individual fibers. This kernel will resolve the interparticle diffusion between bundles of individual fibers in a woven ball-like domain.

Warning

Functions and methods in this file are still under construction.

Author

Austin Ladshaw

Date

04/14/2015

Copyright

6.13.2 Function Documentation

6.13.2.1 double default_porosity (int *i*, int *l*, const void * *user_data*)

Default porosity function for MONKFISH.

This function assumes a linear relationship between the maximum porosity at the center of the woven fibers and the minimum porosity at the edge of the woven fiber bundle.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.13.2.2 double default_density (int *i*, int *l*, const void * *user_data*)

Default density function for MONKFISH.

This function calls the porosity function and uses the single fiber density to provide an estimate of the bulk fiber density locally in the woven fiber bundle.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.13.2.3 double default_interparticle_diffusion (int *i*, int *l*, const void * *user_data*)

Default interparticle diffusion function.

This function assumes that the interparticle diffusivity is a contant and returns that diffusivity multiplied by the domain porosity to form the effective diffusion coefficient in the domain.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.13.2.4 double default_monk_adsorption (int *i*, int *l*, const void * *user_data*)

Default adsorption strength function.

This function will either use the default equilibrium function or the DOGFISH simulation result to produce the approximate adsorption strength using perturbation theory.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.13.2.5 double default_monk_equilibrium (int *i*, int *l*, const void * *user_data*)

Default equilibirium adsorption function in mg/g.

This function uses the exterior species' concentration (mol/L), the species' molecular weight (g/mol), and the bulk fiber density (g/L) to calculate the adsorption equilibrium in mg/g. It assumes that the exterior concentration represents the moles of species per liter of solution that is being sorbed.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.13.2.6 double default_monkfish_retardation (int *i,* int *l,* const void * *user_data*)

Default retardation coefficient function.

This function calls the porosity, density, and adsorption functions to evaluate the retardation coefficient of the diffusing material.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.13.2.7 double default_exterior_concentration (int *i,* const void * *user_data*)

Default exterior concentratio function.

This function assumes that the exterior concentration for sorption is just equal to the value of exterior_concentration given in MONKFISH_PARAM.

Parameters

i	index for the ith adsorbing species
user_data	pointer to the MONKFISH_DATA structure

6.13.2.8 double default_film_transfer (int *i,* const void * *user_data*)

Default film mass transfer function.

This function assumes that the film mass transfer coefficient is just equal to the value of the film_transfer_coeff in MONKFISH_PARAM.

Parameters

i	index for the ith adsorbing species
user_data	pointer to the MONKFISH_DATA structure

6.13.2.9 int setup_MONKFISH_DATA (FILE * *file,* double(*)(int i, int l, const void *user_data) *eval_porosity,* double(*)(int i, int l, const void *user_data) *eval_density,* double(*)(int i, int l, const void *user_data) *eval_ext_diff,* double(*)(int i, int l, const void *user_data) *eval_adsorb,* double(*)(int i, int l, const void *user_data) *eval_retard,* double(*)(int i, const void *user_data) *eval_ext_conc,* double(*)(int i, const void *user_data) *eval_ext_film,* double(*)(int i, int l, const void *user_data) *dog_diffusion,* double(*)(int i, const void *user_data) *dog_ext_film,* double(*)(int i, const void *user_data) *dog_surf_conc,* const void * *user_data,* MONKFISH_DATA * *monk_dat*)

Setup function to allocate memory and setup function pointers for the MONKFISH simulation.

This function will allocate memory and setup the MONKFISH problem. To specify use of the default functions in MONKFISH, pass NULL args for all function pointers and the user_data data structure. Otherwise, pass in your own custom arguments. The MONKFISH_DATA pointer must always be passed to this function.

Parameters

file	pointer to the output file to print out results
eval_porosity	function pointer for the bulk domain porosity function
eval_density	function pointer for the bulk domain density function
eval_ext_diff	function pointer for the interparticle diffusion function
eval_adsorb	function pointer for the adsorption strength function
eval_retard	function pointer for the retardation coefficient function
eval_ext_conc	function pointer for the external concentration function
eval_ext_film	function pointer for the external film mass transfer function
dog_diffusion	function pointer for the DOGFISH diffusion function (see dogfish.h)
dog_ext_film	function pointer for the DOGFISH film mass transfer (see dogfish.h)
dog_surf_conc	function pointer for the DOGFISH surface concentration (see dogfish.h)
user_data	pointer for the user's own data structure (only if using custom functions)
monk_dat	pointer for the MONKFISH_DATA structure

6.13.2.10 int MONKFISH_TESTS ( )

Function to run tests on the MONKFISH algorithms.

This function currently does nothing and is not callable from the UI.

6.14 sandbox.h File Reference

Coding Test Area.

```
#include "flock.h"
#include "school.h"
```

Functions

- int RUN_SANDBOX ()

  Function to run the methods implemented in the Sandbox.

6.14.1 Detailed Description

Coding Test Area.

sandbox.cpp

This file contains a series of simple tests for routines used in other files and algorithms. Before any code or methods are used, they are tested here to make sure that they are useful. The tests in the sandbox are callable from the UI to make it easier to alter existing sandbox code and run tests on new proposed methods or algorithms.

Warning

Functions and methods in this file are not meant to be used anywhere else.

Author

Austin Ladshaw

Date

04/11/2015

Copyright

6.14.2 Function Documentation

6.14.2.1 int RUN_SANDBOX ( )

Function to run the methods implemented in the Sandbox.

This function is callable from the UI and is used to observe results from the tests of newly developed algorithms. Edit header and source files here to test out your own routines or functions. Then you can run those functions by rebuilding the Ecosystem executable and running the sandbox tests.

6.15 school.h File Reference

Seawater Codes from a Highly Object-Oriented Library.

```
#include "eel.h"
#include "mola.h"
#include "shark.h"
#include "dogfish.h"
#include "monkfish.h"
#include "yaml_wrapper.h"
```

6.15.1 Detailed Description

Seawater Codes from a Highly Object-Oriented Library.

This file contains include statements for all files used in the aqueous adsorption problems, primarily targeted at Seawater simulations. Include this file into any other project or source code that needs the methods below.

Files Included in SCHOOL

eel.h mola.h shark.h dogfish.h monkfish.h yaml_wrapper.h

Note

(1) shark.h also includes methods from macaw.h and lark.h
(2) dogfish.h also includes methods from finch.h

Author

Austin Ladshaw

Date

02/23/2015

Copyright

6.16 scopsowl.h File Reference

Simultaneously Coupled Objects for Pore and Surface diffusion Operations With Linear systems.

```
#include "egret.h"
#include "skua.h"
```

Classes

- struct SCOPSOWL_PARAM_DATA

  Data structure for the species' parameters in SCOPSOWL.
- struct SCOPSOWL_DATA

  Primary data structure for SCOPSOWL simulations.

Macros

- #define SCOPSOWL_HPP_
- #define Dp(Dm, ep) (ep∗ep∗Dm)

  Estimate of Pore Diffusivity (cm^2/s)
- #define Dk(rp, T, MW) (9700.0∗rp∗pow((T/MW),0.5))

  Estimate of Knudsen Diffusivity (cm^2/s)
- #define avgDp(Dp, Dk) (pow(((1/Dp)+(1/Dk)),-1.0))

  Estimate of Average Pore Diffusion (cm^2/s)

Functions

- void print2file_species_header (FILE *Output, SCOPSOWL_DATA *owl_dat, int i)

  Function to print out the main header for the output file.
- void print2file_SCOPSOWL_time_header (FILE *Output, SCOPSOWL_DATA *owl_dat, int i)

  Function to print out the time and space header for the output file.
- void print2file_SCOPSOWL_header (SCOPSOWL_DATA *owl_dat)

  Function to call the species and time header functions.
- void print2file_SCOPSOWL_result_old (SCOPSOWL_DATA *owl_dat)

  Function to print out the old time results to the output file.
- void print2file_SCOPSOWL_result_new (SCOPSOWL_DATA *owl_dat)

  Function to print out the new time results to the output file.
- double default_adsorption (int i, int l, const void *user_data)

  Default function for evaluating adsorption and adsorption strength.
- double default_retardation (int i, int l, const void *user_data)

  Default function for evaluating retardation coefficient.
- double default_pore_diffusion (int i, int l, const void *user_data)

  Default function for evaluating pore diffusivity.
- double default_surf_diffusion (int i, int l, const void *user_data)

  Default function for evaluating surface diffusion for HOMOGENEOUS pellets.
- double zero_surf_diffusion (int i, int l, const void *user_data)

  Zero function for evaluating no surface diffusion in HOMOGENEOUS pellets.
- double default_effective_diffusion (int i, int l, const void *user_data)

  Default function for evaluating effective diffusivity for HOMOGENEOUS pellets.
- double const_pore_diffusion (int i, int l, const void *user_data)

  Constant pore diffusion function for homogeneous or heterogeneous pellets.
- double default_filmMassTransfer (int i, const void *user_data)

  Default function for evaluating the film mass transfer coefficient.
- double const_filmMassTransfer (int i, const void *user_data)

  Constant film mass transfer coefficient function.
- int setup_SCOPSOWL_DATA (FILE *file, double(*eval_sorption)(int i, int l, const void *user_data), double(*eval_retardation)(int i, int l, const void *user_data), double(*eval_pore_diff)(int i, int l, const void *user_data), double(*eval_filmMT)(int i, const void *user_data), double(*eval_surface_diff)(int i, int l, const void *user_data), const void *user_data, MIXED_GAS *gas_data, SCOPSOWL_DATA *owl_data)

  Setup function to allocate memory and setup function pointers for the SCOPSOWL simulation.
- int SCOPSOWL_Executioner (SCOPSOWL_DATA *owl_dat)

  SCOPSOWL executioner function to solve a time step.
- int set_SCOPSOWL_ICs (SCOPSOWL_DATA *owl_dat)

  Function to set the initial conditions for a SCOPSOWL simulation.
- int set_SCOPSOWL_timestep (SCOPSOWL_DATA *owl_dat)

  Function to set the timestep of the SCOPSOWL simulation.
- int SCOPSOWL_preprocesses (SCOPSOWL_DATA *owl_dat)

  Function to perform all preprocess SCOPSOWL operations.
- int set_SCOPSOWL_params (const void *user_data)

  Function to set the values of all non-linear system parameters during simulation.
- int SCOPSOWL_postprocesses (SCOPSOWL_DATA *owl_dat)

  Function to perform all postprocess SCOPSOWL operations.
- int SCOPSOWL_reset (SCOPSOWL_DATA *owl_dat)

  Function to reset all stateful information to prepare for next simulation.
- int SCOPSOWL (SCOPSOWL_DATA *owl_dat)

  Function to progress the SCOPSOWL simulation through time till complete.

- int SCOPSOWL_SCENARIOS (const char *scene, const char *sorbent, const char *comp, const char *sorbate)

  Function to perform a SCOPSOWL simulation based on a set of parameters given in input files.
- int SCOPSOWL_TESTS ()

  Function to run a SCOPSOWL test simulation.

6.16.1 Detailed Description

Simultaneously Coupled Objects for Pore and Surface diffusion Operations With Linear systems.

scopsowl.cpp

This file contains structures and functions associated with modeling adsorption in commercial, bi-porous adsorbents such as zeolites and mordenites. The pore diffusion and mass transfer equations are coupled with adsorption and surface diffusion through smaller crystals embedded in a binder matrix. However, you can also direct this simulation to treat the adsorbent as homogeneous (instead of heterogeneous) in order to model an even greater variety of gaseous adsorption kinetic problems. This object is coupled with either MAGPIE, SKUA, or BOTH depending on the type of simulation requested.

Author

Austin Ladshaw

Date

01/29/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.16.2 Macro Definition Documentation

6.16.2.1 #define SCOPSOWL_HPP_

6.16.2.2 #define Dp(*Dm, ep*) (ep*ep*Dm)

Estimate of Pore Diffusivity (cm^2/s)

6.16.2.3 #define Dk(*rp, T, MW*) (9700.0*rp*pow((T/MW),0.5))

Estimate of Knudsen Diffusivity (cm^2/s)

6.16.2.4 #define avgDp(**Dp, Dk**) (pow(((1/Dp)+(1/Dk)),-1.0))

Estimate of Average Pore Diffusion (cm^2/s)

6.16.3 Function Documentation

6.16.3.1 void print2file_species_header (FILE * *Output,* SCOPSOWL_DATA * *owl_dat,* int *i*)

Function to print out the main header for the output file.

6.16.3.2 void print2file_SCOPSOWL_time_header (FILE * *Output,* SCOPSOWL_DATA * *owl_dat,* int *i*)

Function to print out the time and space header for the output file.

6.16.3.3 void print2file_SCOPSOWL_header (SCOPSOWL_DATA * *owl_dat*)

Function to call the species and time header functions.

6.16.3.4 void print2file_SCOPSOWL_result_old (SCOPSOWL_DATA * *owl_dat*)

Function to print out the old time results to the output file.

6.16.3.5 void print2file_SCOPSOWL_result_new (SCOPSOWL_DATA * *owl_dat*)

Function to print out the new time results to the output file.

6.16.3.6 double default_adsorption (int *i,* int *l,* const void * *user_data*)

Default function for evaluating adsorption and adsorption strength.

This function is called in the preprocesses and postprocesses to estimate the strength of adsorption in the macro-scale problem from perturbations. It will use perturbations in either the MAGPIE simulation or SKUA simulation, depending on the type of problem the user is solving.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.7 double default_retardation (int *i,* int *l,* const void * *user_data*)

Default function for evaluating retardation coefficient.

This function is called in the preprocesses and postprocesses to estimate the retardation coefficient for the simulation. It is recalculated at every time step to keep track of all changing conditions in the simulation.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.8 double default_pore_diffusion (int *i*, int *l*, const void * *user_data*)

Default function for evaluating pore diffusivity.

This function is called during the evaluation of non-linear residuals to more accurately represent non-linearities in the pore diffusion behavior. The pore diffusion is calculated based on kinetic theory of gases (see egret.h) and is adjusted according to the Knudsen Diffusion model and the porosity of the binder material.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.9 double default_surf_diffusion (int *i*, int *l*, const void * *user_data*)

Default function for evaluating surface diffusion for HOMOGENEOUS pellets.

This function is ONLY used if the pellet is determined to be homogeneous. Otherwise, this is replaced by the surface diffusion function for the SKUA simulation. The diffusivity is calculated based on the Arrhenius rate expression and then adjusted by the outside partial pressure of the adsorbing species.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.10 double zero_surf_diffusion (int *i*, int *l*, const void * *user_data*)

Zero function for evaluating no surface diffusion in HOMOGENEOUS pellets.

This function is ONLY used if the pellet is determined to be homogeneous and we want to specify that there is no surface diffusion.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.11 double default_effective_diffusion (int *i*, int *l*, const void * *user_data*)

Default function for evaluating effective diffusivity for HOMOGENEOUS pellets.

This function is ONLY used if the pellet is determined to be homogeneous. Otherwise, this is replaced by the pore diffusion function. The effective diffusivity is determined by the combination of pore diffusivity and surface diffusivity with adsorption strength in an homogeneous pellet.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.12 double const_pore_diffusion (int *i*, int *l*, const void * *user_data*)

Constant pore diffusion function for homogeneous or heterogeneous pellets.

This function should be used if the user wants to specify a constant pore diffusivity. The value of pore diffusion is then set equal to the value of pore_diffusion in the SCOPSOWL_PARAM_DATA structure.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.13 double default_filmMassTransfer (int *i*, const void * *user_data*)

Default function for evaluating the film mass transfer coefficient.

This function is called during the setup of the boundary conditions and is used to estimate the film mass transfer coefficient for the macro-scale problem. The coefficient is calculated according to the kinetic theory of gases (see egret.h).

Parameters

i	index for the ith species in the system
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.14 double const_filmMassTransfer (int *i*, const void * *user_data*)

Constant film mass transfer coefficient function.

This function is used when the user wants to specify a constant value for film mass transfer. The value of that coefficient is then set equal to the value of film_transfer in the SCOPSOWL_PARAM_DATA structure.

Parameters

i	index for the ith species in the system
user_data	pointer for the SCOSPOWL_DATA structure

6.16.3.15 int setup_SCOPSOWL_DATA (FILE * *file*, double(*)(int i, int l, const void *user_data) *eval_sorption*, double(*)(int i, int l, const void *user_data) *eval_retardation*, double(*)(int i, int l, const void *user_data) *eval_pore_diff*, double(*)(int i, const void *user_data) *eval_filmMT*, double(*)(int i, int l, const void *user_data) *eval_surface_diff*, const void * *user_data*, MIXED_GAS * *gas_data*, SCOPSOWL_DATA * *owl_data*)

Setup function to allocate memory and setup function pointers for the SCOPSOWL simulation.

This function sets up the memory and function pointers used in SCOPSOWL simulations. User can provide NULL in place of functions for the function pointers and the setup will automatically use just the default settings. However, the user is required to pass the necessary data structure pointers for MIXED_GAS and SCOPSOWL_DATA.

Parameters

file	pointer to the output file to print out results
eval_sorption	pointer to the adsorption evaluation function
eval_retardation	pointer to the retardation evaluation function
eval_pore_diff	pointer to the pore diffusion function
eval_filmMT	pointer to the film mass transfer function
eval_surface_diff	pointer to the surface diffusion function (required)
user_data	pointer to the user's data structure used for the parameter functions
gas_data	pointer to the MIXED_GAS structure used to evaluate kinetic gas theory
owl_data	pointer to the SCOPSOWL_DATA structure

6.16.3.16 int SCOPSOWL_Executioner (SCOPSOWL_DATA * *owl_dat*)

SCOPSOWL executioner function to solve a time step.

This function will call the preprocess, solver, and postprocess functions to evaluate a single time step in a simulation. All simulation conditions must be set prior to calling this function. This function will typically be the one called from other simulations that will involve a SCOPSOWL evaluation to resolve kinetic coupling.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.17 int set_SCOPSOWL_ICs (SCOPSOWL_DATA * *owl_dat*)

Function to set the initial conditions for a SCOPSOWL simulation.

This function will setup the initial conditions of the simulation based on the initial temperature, pressure, and adsorbed molefractions. It assumes that the initial conditions are constant throughout the domain of the problem. This function should only be called once during a simulation.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.18 int set_SCOPSOWL_timestep (SCOPSOWL_DATA * *owl_dat*)

Function to set the timestep of the SCOPSOWL simulation.

This function is used to set the next time step to be used in the SCOPSOWL simulation. A constant time step based on the size of the pellet discretization will be used. Users may want to use a custom time step to ensure that coupled-multi-scale systems are all in sync.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.19 int SCOPSOWL_preprocesses (SCOPSOWL_DATA * *owl_dat*)

Function to perform all preprocess SCOPSOWL operations.

This function will update the boundary conditions and simulation conditions based on the current temperature, pressure, and gas phase composition, which may all vary in time. Since this function is called by the SCOPSOW↩ L_Executioner, it does not need to be called explicitly by the user.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.20 int set_SCOPSOWL_params (const void * *user_data*)

Function to set the values of all non-linear system parameters during simulation.

This is the function override for the FINCH setparams function (see finch.h). It will update the values of non-linear parameters in the residuals so that all variables in a species' system are fully coupled.

Parameters

user_data	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.21 int SCOPSOWL_postprocesses (SCOPSOWL_DATA * *owl_dat*)

Function to perform all postprocess SCOPSOWL operations.

This function will update the retardation coefficients based on newly obtained simulation results for the current time step and calculate the average and total amount of adsorption of each species in the domain. Additionally, this function will call the print functions to store simulation results in the output file.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.22 int SCOPSOWL_reset (SCOPSOWL_DATA * *owl_dat*)

Function to reset all stateful information to prepare for next simulation.

This function will update the stateful information used in SCOPSOWL to prepare the system for the next time step in the simulation. However, because updating the states erases the old state, the user must be absolutely sure that

the simulation is ready to be updated. For just running standard simulations, this is not an issue, but in coupling with other simulations it is very important.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.23 int SCOPSOWL (SCOPSOWL_DATA * *owl_dat*)

Function to progress the SCOPSOWL simulation through time till complete.

This function will call the initial conditions, then progressively call the executioner, time step, and reset functions to propagate the simulation in time. As such, this function is primarily used when running a SCOPSOWL simulation by itself and not when coupling it to an other problem.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.16.3.24 int SCOPSOWL_SCENARIOS (const char * *scene,* const char * *sorbent,* const char * *comp,* const char * *sorbate*)

Function to perform a SCOPSOWL simulation based on a set of parameters given in input files.

This is the primary function to be called when running a stand-alone SCOPSOWL simulation. Parameters and system information for the simulation are given in a series of input files that come in as character arrays. These inputs are all required to call this function.

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File

Note

Each input file has a particular format that must be strictly adhered to in order for the simulation to be carried out correctly. The format for each input file, and an example, is provided below...

Scenario Input Format

System Temperature (K) [tab] Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Simulation Time (hrs) [tab] Print Out Time (hrs)
BC Type (0 = Neumann, 1 = Dirichlet)
Number of Gas Species
Initial Total Adsorption (mol/kg)
Name of ith Species [tab] Adsorbable? (0 = false, 1 = true) [tab] Gas Phase Molefraction [tab] Initial Sorbed Molefraction
(repeat above for all species)

Example Scenario Input

353.15 101.35 0.36
4.0 0.05
0
5
0.0
N2 0 0.7634 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0191 0.0
Above example is for a 5-component mixture of N2, O2, Ar, CO2, and H2O, but we are only considering the H2O as adsorbable.

Adsorbent Input File

Heterogeneous Pellet? (0 = false, 1 = true) [tab] Surface Diffusion Included? (0 = false, 1 = true)
Macro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (cm) [tab] Pellet Density (kg/L) [tab] Porosity (vol. void / vol. binder) [tab] Pore Radius (cm)
(Below is only needed if pellet is Heterogeneous)
Micro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
Crystal Radius (um) [tab] Binder Fraction (vol. binder / vol. pellet)

Example Adsorbent Input

1 1
2
0.118 1.69 0.272 3.5E-6
2
2.0 0.175
Above example is for a heterogeneous adsorbent with surface diffusion. The pellet and crystals are both considered spherical. Pellet radius is 0.118 cm, density is 1.69 kg/L, porosity is 0.272, and pore size is 3.5e-6 cm. The pellet is made up of 17.5 % binder material and contains crystals roughly 2.0 um in radius.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846

0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is a continuation of the Scenario Input example wherein each grouping represents parameters that are associated with N2, O2, Ar, CO2, and H2O, respectively. The order is VERY important!

Adsorbate Input File

{ Type of Surface Diffusion Function (0 = constant, 1 = simple Darken, 2 = theoretical Darken) }
(NOTE: The above option is only given IF the pellet was specified as Heterogeneous!)
Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0
0.8814 0.0
267.999 0.0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
1.28 540.1
374.99 0.01
3.01
1.27
2
-46597.5 -53.6994
-125024 -221.073
Above example would be for a simulation involving two adsorbable species using a constant surface diffusion function. Each adsorbable species has it's own set of kinetic and equilibrium parameters that must be given in the same order as the species appeared in the Scenario Input. Note: we do not need to supply this information for non-adsorbable species.

6.16.3.25 int SCOPSOWL_TESTS ( )

Function to run a SCOPSOWL test simulation.

This function runs a test of the SCOPSOWL physics and prints out results to a text file. It is callable from the UI.

6.17 scopsowl_opt.h File Reference

Optimization Routine for Surface Diffusivities in SCOPSOWL.

```
#include "scopsowl.h"
```

Classes

- struct SCOPSOWL_OPT_DATA

  Data structure for the SCOPSOWL optmization routine.

Functions

- int SCOPSOWL_OPT_set_y (SCOPSOWL_OPT_DATA *owl_opt)

  Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.
- int initial_guess_SCOPSOWL (SCOPSOWL_OPT_DATA *owl_opt)

  Function to set up an initial guess for the surface diffusivity parameter in SCOPSOWL.
- void eval_SCOPSOWL_Uptake (const double *par, int m_dat, const void *data, double *fvec, int *info)

  Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.
- int SCOPSOWL_OPTIMIZE (const char *scene, const char *sorbent, const char *comp, const char *sorbate, const char *data)

  Function called to perform the optimization routine given a specific set of information and data.

6.17.1 Detailed Description

Optimization Routine for Surface Diffusivities in SCOPSOWL.

scopsowl_opt.cpp

This file contains structures and functions associated with performing non-linear least-squares optimization of the SCOPSOWL simulation results against actual kinetic adsorption data. The optimization routine here allows you to run data comparisons and optimizations in three forms: (i) Rough optimizations - cheaper operations, but less accurate, (ii) Exact optmizations - much more expensive, but greater accuracy, and (iii) data/model comparisons - no optimization, just using system parameters to compare simulation results agains a set of data.

Depending on the level of optimization desired, this routine could take several minutes or several hours. The optimization/comparisons are printed out in two files: (i) a parameter file, which contains the simulation partial pressures and temperatures and the optimized diffusivities with the euclidean norm of the fitting and (ii) a comparison file that shows the model value and data value at each time step for each kinetic curve.

The optimized diffusion parameters are given for each individual kinetic data curve. Each data curve will have a different pairing of partial pressure and temperature. Because of this, you will get a list of different diffusivities for each data curve. To get the optimum kinetic parameters from this list of diffusivities, you must fit the diffusion parameter values to the following diffusion function model...
D_opt = D_ref * exp(-E / (R*T)) * pow(p , (T_ref/T) - B)
where D_ref is the Reference Diffusivity (um^2/hr), E is the activation energy for adsorption (J/mol), R is the gas law constant (J/K/mol), T is the system temperature (K), p is the partial pressure of the adsorbing species (kPa), T_ref is the Reference Temperature (K), and B is the Affinity constant. This algorithm does not automatically produce these parameters for you, but gives you everything you need to produce them yourself.
This routine allows you to optimize multiple kinetic curves at one time. However, all data must be for the same adsorbent-adsorbate system. In other words, the adsorbent and adsorbate pair must be the same for each kinetic curve analyzed. Also, each experiment must have been done in a thin bed or continuous flow system where the adsorbents were exposed to a nearly constant outside partial pressure for all time steps and the gas velocity of that system is assumed constant for all experiments. This experimental setup is very typical for studying adsorption kinetics for gas-solid systems.

Author

Austin Ladshaw

Date

05/14/2015

Copyright

6.17.2 Function Documentation

6.17.2.1 int SCOPSOWL_OPT_set_y (SCOPSOWL_OPT_DATA * *owl_opt*)

Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.

This function takes the current mole fraction of the adsorbing gas and calculates the gas mole fractions of the other gases in the sytem based on the standard inlet gas composition given in the scenario file.

6.17.2.2 int initial_guess_SCOPSOWL (SCOPSOWL_OPT_DATA * *owl_opt*)

Function to set up an initial guess for the surface diffusivity parameter in SCOPSOWL.

This function performs the Rough optimization on the surface diffusivity based on the idea of reducing or eliminating function bias between data and simulation. A positive function bias means that the simulation curve is "higher" than the data curve and a negative function bias means that the simulation curve is "lower" than the data curve. We use this information to incrementally adjust the rate of surface diffusion until this bias is near zero. When bias is near zero, the simulation is nearly optimized, but further refinement may be necessary to find the true minimum solution.

6.17.2.3 void eval_SCOPSOWL_Uptake (const double * *par,* int *m_dat,* const void * *data,* double * *fvec,* int * *info*)

Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.

This function will run the SCOPSOWL simulation at a given value of surface diffusivity and produce residuals that feed into the Levenberg-Marquardt's algorithm for non-linear least-squares regression. The form of this function is specific to the format required by the lmfit routine.

Parameters

par	array of parameters that are to be optimized
m_dat	number of data points or functions to evaluate
data	user supplied data structure holding information necessary to form the residuals
fvec	array of residuals computed at the current parameter values
info	integer pointer denoting whether or not the user requests to end a particular simulation

6.17.2.4 int SCOPSOWL_OPTIMIZE (const char * *scene,* const char * *sorbent,* const char * *comp,* const char * *sorbate,* const char * *data*)

Function called to perform the optimization routine given a specific set of information and data.

This is the function that is callable by the UI. The user must provide 5 input files to the routine in order to establish simulation conditions, adsborbent properties, component properties, adsorbate equilibrium parameters, and the set of data that we are comparing the simulations to. Each input file has a very specific structure and order to the information that it contains. The structure here is DIFFERENT than the structure for just running standard SCOP↩ SOWL simulations (see scopsowl.h).

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File
data	Kinetic Adsorption Data File

Note

Much of the structure of these input files are "similar" to that of the input files used in SCOPSOWL_SCENA↩ RIOS (see scopsowl.h), but with some notable differences. Below gives the format for each input file with an example. Make sure your input files follow this format before calling this routine from the UI.

Scenario Input File

Optimization? (0 = false, 1 = true) [tab] Rough Optimization? (0 = false, 1 = true)
Surf. Diff. (0 = constant, 1 = simple Darken, 2 = theoretical Darken) [tab] BC Type (0 = Neumann, 1 = Dirichlet)
Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Number of Gaseous Species
Initial Adsorption Total (mol/kg)
Name [tab] Adsorbable? (0 = false, 1 = true) [tab] Inlet Gas Mole Fraction [tab] Initial Adsorbed Mole Fraction
(NOTE: The above line is repeated for all species in gas phase. Also, this algorithm only allows you to consider one adsorbable gas component. Inlet gas mole fractions must be non-zero for all non-adsorbing gases and must sum to 1.)

Example Scenario Input

1 0
0 0
101.35 0.36
5
0.0
N2 0 0.7825 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0 0.0
Above example is for running optimizations on data collected with a gas stream at 0.36 cm/s with 5 gas species in the mixture, only H2O of which is adsorbing. The "base line" or "inlet gas" without H2O has a composition of N2 at 0.7825, O2 at 0.2081, Ar at 0.009, and CO2 at 0.0004.

Adsorbent Input File

Heterogeneous Pellet? (0 = false, 1 = true)
Macro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (cm) [tab] Pellet Density (kg/L) [tab] Porosity (vol. void / vol. binder) [tab] Pore Radius (cm)
(Below is only needed if pellet is Heterogeneous)
Micro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
Crystal Radius (um) [tab] Binder Fraction (vol. binder / vol. pellet)

Example Adsorbent Input

1
2
0.118 1.69 0.272 3.5E-6
2
2.0 0.175
Above example is nearly identical to the file given in the SCOPSOWL_SCENARIO example (see scopsowl.h). However, here we do not give an integer flag denoting whether or not we are considering surface diffusion as a mechanism. This is because we automatically assume that surface diffusion is a mechanism in the system, since that is the unknown parameter that we are performing the optimizations for.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846
0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is exactly the same as in the SCOPSOWL_SCENARIO example (see scopsowl.h). There is no difference in the input file formats for this input. Keep in mind that the order is VERY important! All species information must be in the same order that the species appeared in the Scenario input file.

Adsorbate Input File

Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0 0
0 0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
Above example gives the equilibrium parameters associated with the H2O-MS3A single component adsorption system. Note that the kinetic parameters (Ref. Diff., Act. Energy, Ref. Temp., and Affinity) were all given a value of zero. These values are irrelavent if we are running an optimization because they will be replaced with a single estimate for the diffusivity that is being optimization for. However, if we wanted to run this routine with comparisons and not do any optmization, then you would need to provide non-zero values for these parameters (at least for Ref. Diff.).

Data Input File

Number of Kinetic Data Curves
Number of data points in the ith curve
Temperature (K) [tab] Partial Pressure (kPa) [tab] Equilibrium Adsorption (mol/kg) all of ith curve
Time point 1 (hrs) [tab] Adsorption 1 (mol/kg) of ith curve
Time point 1 (hrs) [tab] Adsorption 2 (mol/kg) of ith curve
... (Repeat for all time-adsorption data points)
(Repeat above for all curves i)

Example Data Input

40
2990
298.15 0.000310922 2.9
0 0
0.166666667 0.001834419
0.333611111 0.004880247
0.5 0.008306803
...
2789
298.15 0.00055189 5
0 0
0.166944444 0.003350185
0.333611111 0.007418267
0.5 0.009930906
0.666666667 0.014597236
0.833611111 0.021377373
....
Above is a partial example for a data set of 40 kinetic curves. The first curve contains 2990 data points and has temperature of 298.15 K, partial pressure of 0.000310922 kPa, and an equilibrium adsorption of 2.9. Each first time point should start from 0 hours and each initial adsorption should correspond to the value of initial adsorption indicated in the Scenario input file. Then, this structure is repeated for all adsorptio curves.

6.18 shark.h File Reference

Speciation-object Hierarchy for Adsorption Reactions and Kinetics.

```
#include "mola.h"
#include "macaw.h"
#include "lark.h"
#include "yaml_wrapper.h"
#include "dogfish.h"
```

Classes

- class MasterSpeciesList
  Master Species List Object.
- class Reaction
  Reaction Object.
- class MassBalance

Mass Balance Object.

- class UnsteadyReaction

  Unsteady Reaction Object (inherits from Reaction)
- class AdsorptionReaction

  Adsorption Reaction Object.
- class UnsteadyAdsorption

  Unsteady Adsorption Reaction Object.
- class MultiligandAdsorption

  Multi-ligand Adsorption Reaction Object.
- class ChemisorptionReaction

  Chemisorption Reaction Object.
- class MultiligandChemisorption

  Multi-ligand Chemisorption Reaction Object.
- struct SHARK_DATA

  Data structure for SHARK simulations.

Macros

- #define Rstd 8.3144621

  Gas Law Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)
- #define Na 6.0221413E+23

  Avagadro's Number - Units: molecules/mol.
- #define kB 1.3806488E-23

  Boltzmann's Constant - Units: J/K or C∗V/K.
- #define e 1.6021766208E-19

  Elementary Electric Charge - Units: C.
- #define Faraday 96485.33289

  Faraday's Constant - C/mol.
- #define VolumeSTD 15.17

  Standard Segment Volume - cm^3/mol.
- #define AreaSTD 2.5E5

  Standard Segment Area - m^2/mol.
- #define CoordSTD 10

  Standard Coordination Number.
- #define LengthFactor(z, r, s) (((z/2.0)∗(r-s)) - (r-1.0))

  Calculation of the Length Factor Parameter in UNIQUAC.
- #define VacuumPermittivity 8.8541878176E-12

  Vacuum Permittivity Constant - F/m or C/V/m.
- #define WaterRelPerm 80.1

  Approximate Relative Permittivity for water - Unitless.
- #define AbsPerm(Rel) (Rel∗VacuumPermittivity)

  Calculation of Absolute Permittivity of a medium - F/m or C/V/m.

Typedefs

- typedef struct SHARK_DATA SHARK_DATA

  Data structure for SHARK simulations.

Enumerations

- enum valid_mb { BATCH, CSTR, PFR }

  Enumeration for the list of valid activity models for non-ideal solutions.
- enum valid_act {
  IDEAL, DAVIES, DEBYE_HUCKEL, SIT,
  PITZER }

  Enumeration for the list of valid activity models for non-ideal solutions.
- enum valid_surf_act { IDEAL_ADS, FLORY_HUGGINS, UNIQUAC_ACT }

  Enumeration for the list of valid surface activity models for non-ideal adsorption.

Functions

- void print2file_shark_info (SHARK_DATA *shark_dat)

  Function to print out simulation conditions and options to the output file.
- void print2file_shark_header (SHARK_DATA *shark_dat)

  Function to print out the head of species and time stamps to the output file.
- void print2file_shark_results_new (SHARK_DATA *shark_dat)

  Function to print out the simulation results for the current time step.
- void print2file_shark_results_old (SHARK_DATA *shark_dat)

  Function to print out the simulation results for the previous time step.
- double calculate_ionic_strength (const Matrix< double > &x, MasterSpeciesList &MasterList)

  Function to calculate the ionic strength of the solution.
- int FloryHuggins (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for simple non-ideal adsorption (for adsorption reaction object)
- int FloryHuggins_unsteady (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for simple non-ideal adsorption (for unsteady adsorption object)
- int FloryHuggins_multiligand (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for simple non-ideal adsorption (for multiligand adsorption object)
- int FloryHuggins_chemi (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for simple non-ideal adsorption (for chemisorption reaction object)
- int FloryHuggins_multichemi (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for simple non-ideal adsorption (for multiligand chemisorption object)
- int UNIQUAC (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for adsorption reaction object)
- int UNIQUAC_unsteady (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for unsteady adsorption object)
- int UNIQUAC_multiligand (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand adsorption object)
- int UNIQUAC_chemi (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for chemisorption reaction object)
- int UNIQUAC_multichemi (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand chemisorption object)
- int ideal_solution (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Activity function for Ideal Solution.
- int Davies_equation (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Activity function for Davies Equation.
- int DebyeHuckel_equation (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Activity function for Debye-Huckel Equation.
- int surf_act_choice (const std::string &input)

  First test of SIT Model.

- int act_choice (const std::string &input)

  Function takes a given string and returns a flag denoting which activity model was choosen.
- int reactor_choice (const std::string &input)

  Function takes a give string and returns a flag denoting which type of reactor was choosen for the system.
- bool linesearch_choice (const std::string &input)

  Function returns a bool to determine the form of line search requested.
- int linearsolve_choice (const std::string &input)

  Function returns the linear solver flag for the PJFNK method.
- int Convert2LogConcentration (const Matrix< double > &x, Matrix< double > &logx)

  Function to convert the given values of variables (x) to the log of those variables (logx)
- int Convert2Concentration (const Matrix< double > &logx, Matrix< double > &x)

  Function to convert the given log values of variables (logx) to the values of those variables (x)
- int read_scenario (SHARK_DATA *shark_dat)

  Function to go through the yaml object for the scenario document.
- int read_multiligand_scenario (SHARK_DATA *shark_dat)

  Function to go through the yaml object to setup memory space for multiligand objects.
- int read_multichemi_scenario (SHARK_DATA *shark_dat)

  Function to go through the yaml object to setup memory space for multiligand chemisorption objects.
- int read_options (SHARK_DATA *shark_dat)

  Function to go through the yaml object for the solver options document.
- int read_species (SHARK_DATA *shark_dat)

  Function to go through the yaml object for the master species document.
- int read_massbalance (SHARK_DATA *shark_dat)

  Function to go through the yaml object for the mass balance document.
- int read_equilrxn (SHARK_DATA *shark_dat)

  Function to go through the yaml object for the equilibrium reaction document.
- int read_unsteadyrxn (SHARK_DATA *shark_dat)

  Function to go through the yaml object for the unsteady reaction document.
- int read_adsorbobjects (SHARK_DATA *shark_dat)

  Function to go through the yaml object for each Adsorption Object.
- int read_unsteadyadsorbobjects (SHARK_DATA *shark_dat)

  Function to go through the yaml object for each Unsteady Adsorption Object.
- int read_multiligandobjects (SHARK_DATA *shark_dat)

  Function to go through the yaml object for each MultiligandAdsorption Object.
- int read_chemisorbobjects (SHARK_DATA *shark_dat)

  Function to go through the yaml object for each Chemisorption Object.
- int read_multichemiobjects (SHARK_DATA *shark_dat)

  Function to go through the yaml object for each MultiligandChemisorption Object.
- int setup_SHARK_DATA (FILE *file, int(*residual)(const Matrix< double > &x, Matrix< double > &res, const void *data), int(*activity)(const Matrix< double > &x, Matrix< double > &gama, const void *data), int(*precond)(const Matrix< double > &r, Matrix< double > &p, const void *data), SHARK_DATA *dat, const void *activity_data, const void *residual_data, const void *precon_data, const void *other_data)

  Function to setup the memory and pointers for the SHARK_DATA structure for the current simulation.
- int shark_add_customResidual (int i, double(*other_res)(const Matrix< double > &x, SHARK_DATA *shark_dat, const void *other_data), SHARK_DATA *shark_dat)

  Function to add user defined custom residual functions to the OtherList vector object in SHARK_DATA.
- int shark_parameter_check (SHARK_DATA *shark_dat)

  Function to check the Reaction and UnsteadyReaction objects for missing info.
- int shark_energy_calculations (SHARK_DATA *shark_dat)

  Function to calculate all Reaction and UnsteadyReaction energies.
- int shark_temperature_calculations (SHARK_DATA *shark_dat)

Function to calculate all Reaction and UnsteadyReaction parameters as a function of temperature.

- int shark_pH_finder (SHARK_DATA *shark_dat)

  Function will search MasterSpeciesList for existance of H + (aq) and OH - (aq) molecules.
- int shark_guess (SHARK_DATA *shark_dat)

  Function provides a rough initial guess for the values of all non-linear variables.
- int shark_initial_conditions (SHARK_DATA *shark_dat)

  Function to establish the initial conditions of the shark simulation.
- int shark_executioner (SHARK_DATA *shark_dat)

  Function to execute a shark simulation at a single time step or pH value.
- int shark_timestep_const (SHARK_DATA *shark_dat)

  Function to set up all time steps in the simulation to a specified constant.
- int shark_timestep_adapt (SHARK_DATA *shark_dat)

  Function to set up all time steps in the simulation based on success or failure to converge.
- int shark_preprocesses (SHARK_DATA *shark_dat)

  Function to call other functions for calculation of parameters and setting of time steps.
- int shark_solver (SHARK_DATA *shark_dat)

  Function to call the PJFNK solver routine given the current SHARK_DATA information.
- int shark_postprocesses (SHARK_DATA *shark_dat)

  Function to convert PJFNK solutions to concentration values and print to the output file.
- int shark_reset (SHARK_DATA *shark_dat)

  Function to reset the values of all stateful information in SHARK_DATA.
- int shark_residual (const Matrix< double > &x, Matrix< double > &F, const void *data)

  Default residual function for shark evaluations.
- int SHARK (SHARK_DATA *shark_dat)

  Function to call all above functions to perform a shark simulation.
- int SHARK_SCENARIO (const char *yaml_input)

  Function to perform a shark simulation based on the conditions in a yaml formatted input file.
- int SHARK_TESTS ()

  Function to perform a series of shark calculation tests.
- int SHARK_TESTS_OLD ()

  Function to perform a series of shark calculation tests (older version)

6.18.1 Detailed Description

Speciation-object Hierarchy for Adsorption Reactions and Kinetics.

shark.cpp

This file contains structures and functions associated with solving speciation and kinetic problems in aqueous systems. The primary aim for the development of these algorithms was to solve speciation and adsorption problems for the recovery of uranium resources from seawater. Seawater is an extradorinarily complex medium in which to work, which is why these algorithms are being constructed in a piece-wise, object-oriented fashion. This allows us to displace much of the complexity of the problem by breaking it down into smaller, more managable pieces.

Each piece of SHARK contributes to a residual function when solving the overall speciation, reaction, kinetic chemical problem. These residuals are then fed into the PJFNK solver function in lark.h. The variables of the system are the log(C) concentration values of each species in the system. We solve for log(C) concentrations, rather than just C, because the PJFNK method is an unbounded solution algorithm. So to prevent the algorithm from producing negative values for concentration, we reformulate all residuals in terms of the log(C) values. In this way, regardless of the value found for log(C), the concentration C will always be greater than 0.

Currenty, SHARK supports standard aqueous speciation problems with simple kinetic models based on an unsteady form of the standard reaction stoichiometry. As more methods and algorithms are completed, the SHARK simulations will be capable of doing much, much more.

Warning

Much of this is still underconstruction and many methods or interfaces may change. Use with caution.

Author

Austin Ladshaw

Date

05/27/2015

6.18.2 Macro Definition Documentation

6.18.2.1 #define Rstd 8.3144621

Gas Law Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)

6.18.2.2 #define Na 6.0221413E+23

Avagadro's Number - Units: molecules/mol.

6.18.2.3 #define kB 1.3806488E-23

Boltzmann's Constant - Units: J/K or C∗V/K.

6.18.2.4 #define e 1.6021766208E-19

Elementary Electric Charge - Units: C.

6.18.2.5 #define Faraday 96485.33289

Faraday's Constant - C/mol.

6.18.2.6 #define VolumeSTD 15.17

Standard Segment Volume - cm^3/mol.

6.18.2.7 #define AreaSTD 2.5E5

Standard Segment Area - m^2/mol.

6.18.2.8 #define CoordSTD 10

Standard Coordination Number.

6.18.2.9 #define LengthFactor(z, r, s) (((z/2.0)*(r-s)) - (r-1.0))

Calculation of the Length Factor Parameter in UNIQUAC.

6.18.2.10 #define VacuumPermittivity 8.8541878176E-12

Vacuum Permittivity Constant - F/m or C/V/m.

6.18.2.11 #define WaterRelPerm 80.1

Approximate Relative Permittivity for water - Unitless.

6.18.2.12 #define AbsPerm(Rel) (Rel*VacuumPermittivity)

Calculation of Absolute Permittivity of a medium - F/m or C/V/m.

6.18.3 Typedef Documentation

6.18.3.1 typedef struct SHARK_DATA SHARK_DATA

Data structure for SHARK simulations.

C-style object holding data and function pointers associated with solving aqueous speciation and reaction kinetics. This object couples all other objects available in shark.h in order to provide residual calculations for each individual function that makes up the overall system model. Those residuals are brought together inside the residual function and fed into the lark.h PJFNK solver routine. That solver then attempts to find a solution to all non-linear variables simultaneously. Any function or data pointers in this structure can be overriden to change how you interface with and solve the problem. Users may also provide a set of custom residual functions through the "OtherList" vector object. Those residual function must all have the same format.

6.18.4 Enumeration Type Documentation

6.18.4.1 enum valid_mb

Enumeration for the list of valid activity models for non-ideal solutions.

Note

The SIT and PITZER models are not currently supported.

Enumerator

BATCH

CSTR

PFR

6.18.4.2 enum valid_act

Enumeration for the list of valid activity models for non-ideal solutions.

Note

The SIT and PITZER models are not currently supported.

Enumerator

IDEAL
DAVIES
DEBYE_HUCKEL
SIT
PITZER

6.18.4.3 enum valid_surf_act

Enumeration for the list of valid surface activity models for non-ideal adsorption.

Note

We had to create an IDEAL_ADS option to replace the IDEAL enum already in use for non-ideal solution or aqueous phases. (ADS => adsorption)

Enumerator

IDEAL_ADS
FLORY_HUGGINS
UNIQUAC_ACT

6.18.5 Function Documentation

6.18.5.1 void print2file_shark_info (SHARK_DATA * *shark_dat*)

Function to print out simulation conditions and options to the output file.

6.18.5.2 void print2file_shark_header (SHARK_DATA * *shark_dat*)

Function to print out the head of species and time stamps to the output file.

6.18.5.3 void print2file_shark_results_new (SHARK_DATA * *shark_dat*)

Function to print out the simulation results for the current time step.

6.18.5.4 void print2file_shark_results_old (SHARK_DATA * *shark_dat*)

Function to print out the simulation results for the previous time step.

6.18.5.5 double calculate_ionic_strength (const Matrix< double > & *x*, MasterSpeciesList & *MasterList*)

Function to calculate the ionic strength of the solution.

This function calculates the ionic strength of a system given the concentrations of the species present in solution, as well as any other relavent information from SHARK_DATA such as charge.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
MasterList	reference to the MasterSpeciesList object holding species information

6.18.5.6 int FloryHuggins (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for simple non-ideal adsorption (for adsorption reaction object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. NOTE: Only for AdsorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the AdsorptionReaction object holding parameter information

6.18.5.7 int FloryHuggins_unsteady (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for simple non-ideal adsorption (for unsteady adsorption object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. NOTE: Only for UnsteadyAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the UnsteadyAdsorption object holding parameter information

6.18.5.8 int FloryHuggins_multiligand (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for simple non-ideal adsorption (for multiligand adsorption object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. NOTE: Only for MultiligandAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandAdsorption object holding parameter information

6.18.5.9 int FloryHuggins_chemi (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for simple non-ideal adsorption (for chemisorption reaction object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. NOTE: Only for ChemisorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the ChemisorptionReaction object holding parameter information

6.18.5.10 int FloryHuggins_multichemi (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for simple non-ideal adsorption (for multiligand chemisorption object)

This is a simple surface activity model to be used with the Chemisorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Chemisorption Object itself as the const void *data structure. NOTE: Only for MultiligandChemisorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandChemisorption object holding parameter information

6.18.5.11 int UNIQUAC (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for adsorption reaction object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for AdsorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the AdsorptionReaction object holding parameter information

6.18.5.12 int UNIQUAC_unsteady (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for unsteady adsorption object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for UnsteadyAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the UnsteadyAdsorption object holding parameter information

6.18.5.13 int UNIQUAC_multiligand (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand adsorption object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for MultiligandAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandAdsorption object holding parameter information

6.18.5.14 int UNIQUAC_chemi (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for chemisorption reaction object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void *data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for ChemisorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the ChemisorptionReaction object holding parameter information

6.18.5.15 int UNIQUAC_multichemi (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand chemisorption object)

This is a more complex surface activity model to be used with the Chemisorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative

concentrations of each surface species. Therefore, we will pass the Chemisorption Object itself as the const void *data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for MultiligandChemisorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandChemisorption object holding parameter information

6.18.5.16 int ideal_solution (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Activity function for Ideal Solution.

This is one of the default activity models available. It assumes the system behaves ideally and sets the activity coefficients to 1 for all species.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

6.18.5.17 int Davies_equation (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Activity function for Davies Equation.

This is one of the default activity models available. It uses the Davies semi-empirical model to calculate average activities of each species in solution. This model is typically valid for systems involving high ionic strengths upto 0.5 M (mol/L).

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

6.18.5.18 int DebyeHuckel_equation (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Activity function for Debye-Huckel Equation.

This is one of the default activity models available. It uses the Debye-Huckel limiting model to calculate average activities of each species in solution. This model is typically valid for systems involving low ionic strengths and is only good for solutions between 0 and 0.01 M.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

6.18.5.19 int surf_act_choice (const std::string & *input*)

First test of SIT Model.

Function takes a given string and returns a flag denoting which surface activity model was choosen This function returns an integer flag that will be one of the valid surface activity model flags from the valid_surf_act enum. If the input string was not recognized, then it defaults to returning the IDEAL_ADS flag.

Parameters

input	string for the name of the surface activity model

6.18.5.20 int act_choice (const std::string & *input*)

Function takes a given string and returns a flag denoting which activity model was choosen.

This function returns an integer flag that will be one of the valid activity model flags from the valid_act enum. If the input string was not recognized, then it defaults to returning the IDEAL flag.

Parameters

input	string for the name of the activity model

6.18.5.21 int reactor_choice (const std::string & *input*)

Function takes a give string and returns a flag denoting which type of reactor was choosen for the system.

This function returns an integer flag that will be one of the valid reactor type flags from the valid_mb enum. If the input string was not recognized, then it defaults to returning the BATCH flag.

Parameters

input	string for the name of the activity model

6.18.5.22 bool linesearch_choice (const std::string & *input*)

Function returns a bool to determine the form of line search requested.

This function returns true if the user requests a bouncing line search algorithm and false if the user wants a standard line search. If the input string is unrecognized, then it returns false.

Parameters

input	string for the line search method option

6.18.5.23 int linearsolve_choice (const std::string & *input*)

Function returns the linear solver flag for the PJFNK method.

This function takes in a string argument and returns the integer flag for the appropriate linear solver in PJFNK. If the input string was unrecognized, then it returns the GMRESRP flag.

Parameters

input	string for the linear solver method option

6.18.5.24 int Convert2LogConcentration (const Matrix< double > & *x*, Matrix< double > & *logx*)

Function to convert the given values of variables (x) to the log of those variables (logx)

This function returns an integer flag to denote success of failure. It takes a constant matrix argument x and replaces the elements of the matrix logx with the base 10 log of those x values. This is used mainly to convert a set of concentrations (x) to their respective log(C) values (logx).

Parameters

x	matrix of values to take the base 10 log of
logx	matrix whose entries are to be changed to base 10 log(x)

6.18.5.25 int Convert2Concentration (const Matrix< double > & *logx*, Matrix< double > & *x*)

Function to convert the given log values of variables (logx) to the values of those variables (x)

This function returns an integer flag to denote success of failure. It takes a constant matrix argument logx and replaces the elements of the matrix x with 10^logx. This is used mainly to convert a set of log(C) values (logx) to their respective concentration values (x).

Parameters

logx	matrix of values to apply as the power of 10 (i.e., 10^logx)
x	matrix whose entries are to be changed to the result of 10^logx

6.18.5.26 int read_scenario (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for the scenario document.

This function checks the yaml object for the expected keys and values of the scenario document to setup the shark simulation for the input given in the input file.

6.18.5.27 int read_multiligand_scenario (SHARK_DATA * *shark_dat*)

Function to go through the yaml object to setup memory space for multiligand objects.

This function checks the yaml object for the expected keys and values of the multiligand scenario documents to setup the shark simulation for the input given in the input file.

6.18.5.28 int read_multichemi_scenario (SHARK_DATA * *shark_dat*)

Function to go through the yaml object to setup memory space for multiligand chemisorption objects.

This function checks the yaml object for the expected keys and values of the multiligand chemisorption scenario documents to setup the shark simulation for the input given in the input file.

6.18.5.29 int read_options (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for the solver options document.

This function checks the yaml object for the expected keys and values of the solver options document to setup the shark simulation for the input given in the input file.

6.18.5.30 int read_species (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for the master species document.

This function checks the yaml object for the expected keys and values of the master species document to setup the shark simulation for the input given in the input file.

6.18.5.31 int read_massbalance (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for the mass balance document.

This function checks the yaml object for the expected keys and values of the mass balance document to setup the shark simulation for the input given in the input file.

6.18.5.32 int read_equilrxn (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for the equilibrium reaction document.

This function checks the yaml object for the expected keys and values of the equilibrium reaction document to setup the shark simulation for the input given in the input file.

6.18.5.33 int read_unsteadyrxn (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for the unsteady reaction document.

This function checks the yaml object for the expected keys and values of the unsteady reaction document to setup the shark simulation for the input given in the input file.

6.18.5.34 int read_adsorbobjects (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for each Adsorption Object.

This function checks the yaml object for the expected keys and values of the adsorption object documents to setup the shark simulation for the input given in the input file.

Note

Each adsorption object will have its own document header by the name of that object

6.18.5.35 int read_unsteadyadsorbobjects (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for each Unsteady Adsorption Object.

This function checks the yaml object for the expected keys and values of the unsteady adsorption object documents to setup the shark simulation for the input given in the input file.

Note

Each unsteady adsorption object will have its own document header by the name of that object

6.18.5.36 int read_multiligandobjects (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for each MultiligandAdsorption Object.

This function checks the yaml object for the expected keys and values of the multiligand object documents to setup the shark simulation for the input given in the input file.

Note

Each ligand object will have its own document header by the name of that object

6.18.5.37 int read_chemisorbobjects (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for each Chemisorption Object.

This function checks the yaml object for the expected keys and values of the chemisorption object documents to setup the shark simulation for the input given in the input file.

Note

Each adsorption object will have its own document header by the name of that object

6.18.5.38 int read_multichemiobjects (SHARK_DATA * *shark_dat*)

Function to go through the yaml object for each MultiligandChemisorption Object.

This function checks the yaml object for the expected keys and values of the multiligand chemisorption object documents to setup the shark simulation for the input given in the input file.

Note

Each ligand object will have its own document header by the name of that object

6.18.5.39 int setup_SHARK_DATA (FILE * *file*, int(*)(const Matrix< double > &x, Matrix< double > &res, const void *data) *residual*, int(*)(const Matrix< double > &x, Matrix< double > &gama, const void *data) *activity*, int(*)(const Matrix< double > &r, Matrix< double > &p, const void *data) *precond*, SHARK_DATA * *dat*, const void * *activity_data*, const void * *residual_data*, const void * *precon_data*, const void * *other_data*)

Function to setup the memory and pointers for the SHARK_DATA structure for the current simulation.

This function will be called after reading the scenario file and is used to setup the memory and other pointers for the user requested simulation. This function must be called before running a simulation or trying to read in the remander of the yaml formatted input file. Options may be overriden manually after calling this function.

Parameters

file	pointer for the output file where shark results will be stored
residual	pointer to the residual function that will be fed into the PJFNK solver
activity	pointer to the activity function that will determine the activity coefficients
precond	pointer to the linear preconditioning operation to be applied to the Jacobian
dat	pointer to the SHARK_DATA data structure
activity_data	optional pointer for data needed in activity functions
residual_data	optional pointer for data needed in residual functions
precon_data	optional pointer for data needed in preconditioning functions
other_data	optional pointer for data needed in the evaluation of user defined residual functions

6.18.5.40 int shark_add_customResidual (int *i,* double(∗)(const Matrix< double > &x, SHARK_DATA ∗shark_dat, const void ∗other_data) *other_res,* SHARK_DATA ∗ *shark_dat*)

Function to add user defined custom residual functions to the OtherList vector object in SHARK_DATA.

This function will need to be used if the user wants to include custom residuals into the system via the OtherList object in SHARK_DATA. For each i residual you want to add, you must call this function passing your residual function and the SHARK_DATA structure pointer. The order that those functions are executed in are determined by the integer i.

Parameters

i	index that the other_res function will appear at in the OtherList object
other_res	function pointer for the user's custom residual function
shark_dat	pointer to the SHARK_DATA data structure

6.18.5.41 int shark_parameter_check (SHARK_DATA ∗ *shark_dat*)

Function to check the Reaction and UnsteadyReaction objects for missing info.

This function checks the Reaction and UnsteadyReaction objects for missing information. If information is missing, this function will return an error that will cause the program to force quit.

6.18.5.42 int shark_energy_calculations (SHARK_DATA ∗ *shark_dat*)

Function to calculate all Reaction and UnsteadyReaction energies.

This function will call the calculate energy functions for Reaction and UnsteadyReaction objects.

6.18.5.43 int shark_temperature_calculations (SHARK_DATA ∗ *shark_dat*)

Function to calculate all Reaction and UnsteadyReaction parameters as a function of temperature.

This function will call all temperature dependent functions in Reaction and UnsteadyReaction to calculate equilibrium and reaction rate parameters as a function of system temperature.

6.18.5.44 int shark_pH_finder (SHARK_DATA ∗ *shark_dat*)

Function will search MasterSpeciesList for existance of H + (aq) and OH - (aq) molecules.

This function searches all molecules in the MasterSpeciesList object for the H + (aq) and OH - (aq) molecules. If they are found, then it sets the pH_index and pOH_index of the SHARK_DATA structure and indicates that the system contains these variables.

6.18.5.45 int shark_guess (SHARK_DATA ∗ *shark_dat*)

Function provides a rough initial guess for the values of all non-linear variables.

This function constructs an rough initial guess for the values of all non-linear variables in the system. The guess is based primarily off of trying to statisfy all mass balance constraints, initial conditions, and pH constraints if any apply.

6.18.5.46 int shark_initial_conditions (SHARK_DATA * *shark_dat*)

Function to establish the initial conditions of the shark simulation.

This function will establish the initial conditions for a transient problem by solving the speciation of the system while holding the transient/unsteady variables constant at their respective initial values. However, if the system we are trying to solve is steady, then this function just calls the shark_guess function.

6.18.5.47 int shark_executioner (SHARK_DATA * *shark_dat*)

Function to execute a shark simulation at a single time step or pH value.

This function calls the preprocess, solver, and postprocess functions in order. If a particular solve did not converge, then it will retry the solver routine until it runs out of tries or attains convergence.

6.18.5.48 int shark_timestep_const (SHARK_DATA * *shark_dat*)

Function to set up all time steps in the simulation to a specified constant.

This function will set all time steps for the current simulation to a constant that is specified in the input file. The time step will not be changed unless the simulation fails, then it will be reduced in order to try to get the system to converge.

6.18.5.49 int shark_timestep_adapt (SHARK_DATA * *shark_dat*)

Function to set up all time steps in the simulation based on success or failure to converge.

This function will set all time steps for the current simulation based on some factor multiple of the prior time step used and whether or not the previous solution step was successful. If the previous step converged, then the new time step will be 1.5x the old time step. If it failed, then the simulation will be retried with a new time step of 0.5x the old time step.

6.18.5.50 int shark_preprocesses (SHARK_DATA * *shark_dat*)

Function to call other functions for calculation of parameters and setting of time steps.

This function will call the shark_temperature_calculations function and the appropriate time step function. If the user requests a constant time step, it will call the shark_timestep_const function. Otherwise, it calls the shark_↩ timestep_adapt function.

6.18.5.51 int shark_solver (SHARK_DATA * *shark_dat*)

Function to call the PJFNK solver routine given the current SHARK_DATA information.

This function will perform the necessary steps before and after calling the PJFNK solver routine. Based on the simulation flags, the solver function will perform an intial guess for unsteady variables, call the PJFNK method, and the printout a console message about the performance. If a terminal failure occurs during the solver, it will print out the current state of residuals, variables, and the Jacobian matrix to the console. Analyzing this information could provide clues as to why failure occured.

6.18.5.52 int shark_postprocesses (SHARK_DATA * *shark_dat*)

Function to convert PJFNK solutions to concentration values and print to the output file.

This function will convert the non-linear variables to their respective concentration values, then print the solve information out to the output file.

6.18.5.53 int shark_reset (SHARK_DATA * *shark_dat*)

Function to reset the values of all stateful information in SHARK_DATA.

This function will reset all stateful matrix data in the SHARK_DATA structure in preparation of the next time step simulation.

6.18.5.54 int shark_residual (const Matrix< double > & *x*, Matrix< double > & *F*, const void * *data*)

Default residual function for shark evaluations.

This function calls each individual object's residual function to formulate the overall residual function used in the P↩ JFNK solver routine. It will also call the activity function. The order in which these function calls occurs is as follows: (i) activities, (ii) Reaction, (iii) UnsteadyReaction, (iv) MassBalance, (v) OtherList, and (vi) MasterSpeciesList. If a constant pH is specified, then the MasterSpeciesList residual call is replaced with a constraint on the H + (aq) variable (if one exists).

6.18.5.55 int SHARK (SHARK_DATA * *shark_dat*)

Function to call all above functions to perform a shark simulation.

This function is called after reading in all inputs, setting all constants, and calling the setup function. It will call all the necessary functions and subroutines iteratively until the desired simulation is complete.

6.18.5.56 int SHARK_SCENARIO (const char * *yaml_input*)

Function to perform a shark simulation based on the conditions in a yaml formatted input file.

This is the primary function used to run shark simulations from the UI. It requires that ths user provide one input file that is formatted with yaml keys, symbols, and spacing so that it can be recognized by the parser. This style of input file is much easier to use and understand than the input files used for SCOPSOWL or SKUA. Below shows an example of a typical input file. Note that the # symbol is used in the input file to comment out lines of text that the parser does not need to read.

Example Yaml Input for SHARK

#This will serve as a test input file for shark to demo how to structure the document
#In practice, this section should be listed first, but it doesn't really matter
#DO NOT USE TABS IN THESE INPUT FILES
#— Starts a document ... Ends a document
#All keys must be proceeded by a :
#All lists/header must be preceeded by a -
#Spacing of the keys will indicate which list/header they belong to
Scenario:
—

- vars_fun:
  numvar: 25
  num_ssr: 15
  num_mbe: 7
  num_usr: 2
  num_other: 0 #Not required or used in current version

- sys_data:
  act_fun: davies
  const_pH: false
  pH: 7 #Only required if we are specifying a const_pH
  temp: 298.15 #Units must be in Kelvin
  dielec: 78.325 #Units must be in (1/Kelvin)
  rel_perm: 80.1 #Unitless number
  res_alk: 0 #Units must be in mol/L (Residual Alkalinity)
  volume: 1.0 #Units must be in L

- run_time:
  steady: false #NOTE: All time must be represented in hours
  specs_curve: false #Only needed if steady = true, and will default to false
  dt: 0.001 #Only required if steady = false
  time_adapt: true #Only needed if steady = false, and will default to false
  sim_time: 96.0 #Only required if steady = false
  t_out: 0.01 #Only required if steady = false
  ...

#The following header is entirely optional, but is used to set solver options
SolverOptions:
—
line_search: true #Default = true, and is recommended to be true
search_type: standard
linear_solve: gmresrp #Note: FOM will be fastest for small problems
restart: 25 #Note: restart only used if using GMRES or GCR type solvers
nl_maxit: 50
nl_abstol: 1e-5
nl_reltol: 1e-8
lin_reltol: 1e-10 #Min Tol = 1e-15
lin_abstol: 1e-10 #Min Tol = 1e-15
nl_print: true
l_print: true
...

#After the Scenario read, shark will call the setup_function, then read info below
MasterSpecies:
—
#Header names are specific
#Keys are chosen by user, but must span numbers 0 through numvar-1
#Keys will denote the ordering of the variables
#Note: Currently, the number of reg molecules is very limited

- reg:
  0: Cl - (aq)
  1: NaHCO3 (aq)
  2: NaCO3 - (aq)
  3: Na + (aq)
  4: HNO3 (aq)
  5: NO3 - (aq)
  6: H2CO3 (aq)
  7: HCO3 - (aq)
  8: CO3 2- (aq)
  9: UO2 2+ (aq)
  10: UO2NO3 + (aq)

11: UO2(NO3)2 (aq)
12: UO2OH + (aq)
13: UO2(OH)3 - (aq)
14: (UO2)2(OH)2 2+ (aq)
15: (UO2)3(OH)5 + (aq)
16: UO2CO3 (aq)
17: UO2(CO3)2 2- (aq)
18: UO2(CO3)3 4- (aq)
19: H2O (l)
20: OH - (aq)
21: H + (aq)
#Keys for the sub-headers must follow same rules as keys from above

- unreg:

  - 22:
    formula: A(OH)2 (aq)
    charge: 0
    enthalpy: 0
    entropy: 0
    have_HS: false
    energy: 0
    have_G: false
    phase: Aqueous
    name: Amidoxime
    lin_form: none

  - 23:
    formula: UO2AO2 (aq)
    charge: 0
    enthalpy: 0
    entropy: 0
    have_HS: false
    energy: 0
    have_G: false
    phase: Aqueous
    name: Uranyl-amidoximate
    lin_form: none

  - 24:
    formula: UO2CO3AO2 2- (aq)
    charge: -2
    enthalpy: 0
    entropy: 0
    have_HS: false
    energy: 0
    have_G: false
    phase: Aqueous
    name: Uranyl-carbonate-amidoximate
    lin_form: none
    ...
    #NOTE: Total concentrations must be given in mol/L
    MassBalance:
    —
    #Header names under MassBalance are choosen by the user
    #All other keys will be checked

- water:
  total_conc: 1

  - delta:
    "H2O (l)": 1

- carbonate:
  total_conc: 0.0004175

  - delta:
    "NaHCO3 (aq)": 1
    "NaCO3 - (aq)": 1
    "H2CO3 (aq)": 1
    "HCO3 - (aq)": 1
    "CO3 2- (aq)": 1
    "UO2CO3 (aq)": 1
    "UO2(CO3)2 2- (aq)": 2
    "UO2(CO3)3 4- (aq)": 3
    "UO2CO3AO2 2- (aq)": 1
    #Other mass balances skipped for demo purposes...
    ...
    #Document for equilibrium or steady reactions
    EquilRxn:
    —
    #Headers under EquilRxn separate out each reaction object
    #Keys for these headers only factor into the order of the equations
    #Stoichiometry follows the convention that products are pos(+) and reactants are neg(-)
    #Note: logK is only required if any species in stoichiometry is unregistered
    #Example: below represents - {H2O (l)} –> {H + (aq)} + {OH - (aq)}
    #Note: a valid reaction statement requires at least 1 stoichiometry args
    #Note: You can also provide reaction energies: enthalpy, entropy, and energy

- rxn00:
  logK: -14

  - stoichiometry:
    "H2O (l)": -1
    "OH - (aq)": 1
    "H + (aq)": 1

- rxn01:
  logK: -6.35

  - stoichiometry:
    "H2CO3 (aq)": -1
    "HCO3 - (aq)": 1
    "H + (aq)": 1
    #Other reactions skipped for demo purposes...
    ...
    #Document for unsteady reactions
    UnsteadyRxn:
    —
    #Same basic standards for this doc as the EquilRxn

#Main difference is the inclusion of rate information
#You are required to give at least 1 rate
#You are also required to denote which variable is unsteady
#You must give the initial concentration for the variable in mol/L
#Rate units are in (L/mol)^n/hr
#Note: we also have keys for forward_ref, reverse_ref,
#activation_energy, and temp_affinity.
#These are optional if forward and/or reverse are given
#Note: You can also provide reaction energies: enthalpy, entropy, and energy

- rxn00:
  unsteady_var: UO2AO2 (aq)
  initial_condition: 0
  logK: -1.35
  forward: 4.5e+6
  reverse: 1.00742e+8

  – stoichiometry:
    "UO2 2+ (aq)": -1
    "A(OH)2 (aq)": -1
    "UO2AO2 (aq)": 1
    "H + (aq)": 2

- rxn01:
  unsteady_var: UO2CO3AO2 2- (aq)
  initial_condition: 0
  logK: 3.45
  forward: 2.55e+15
  reverse: 9.04774e+11

  – stoichiometry:
    "UO2 2+ (aq)": -1
    "CO3 2- (aq)": -1
    "A(OH)2 (aq)": -1
    "UO2CO3AO2 2- (aq)": 1
    "H + (aq)": 2
    ...

    Note

    It may be advantageous to look at some other shark input file examples. More input files are provided in the input_files/SHARK directory of the ecosystem project folder. Please refer to your own source file location for more input file examples for SHARK.

6.18.5.57 int SHARK_TESTS ( )

Function to perform a series of shark calculation tests.

This function sets up and solves a test problem for shark. It is callable from the UI.

6.18.5.58 int SHARK_TESTS_OLD ( )

Function to perform a series of shark calculation tests (older version)

This function sets up and solves a test problem for shark. It is NOT callable from the UI.

6.19 skua.h File Reference

Surface Kinetics for Uptake by Adsorption.

```
#include "finch.h"
#include "magpie.h"
#include "egret.h"
```

Classes

- struct SKUA_PARAM

  Data structure for species' parameters in SKUA.
- struct SKUA_DATA

  Data structure for all simulation information in SKUA.

Macros

- #define SKUA_HPP_
- #define D_inf(Dref, Tref, B, p, T) (Dref ∗ pow(p+sqrt(DBL_EPSILON),(Tref/T)-B))

  Empirical correction of diffusivity (um^2/hr)
- #define D_o(Diff, E, T) (Diff ∗ exp(-E/(Rstd∗T)))

  Arrhenius Rate Expression for Diffusivity (um^2/hr)
- #define D_c(Diff, phi) (Diff ∗ (1.0/((1.0+1.1E-6)-phi)))

  Approximate Darken Diffusivity Equation (um^2/hr)

Functions

- void print2file_species_header (FILE ∗Output, SKUA_DATA ∗skua_dat, int i)

  Function to print out the species' headers to output file.
- void print2file_SKUA_time_header (FILE ∗Output, SKUA_DATA ∗skua_dat, int i)

  Function to print out time and space headers to output file.
- void print2file_SKUA_header (SKUA_DATA ∗skua_dat)

  Function calls the other header functions to establish output file structure.
- void print2file_SKUA_results_old (SKUA_DATA ∗skua_dat)

  Function to print out the old time step simulation results to the output file.
- void print2file_SKUA_results_new (SKUA_DATA ∗skua_dat)

  Function to print out the new time step simulation results to the output file.
- double default_Dc (int i, int l, const void ∗data)

  Default function for surface diffusivity.
- double default_kf (int i, const void ∗data)

  Default function for film mass transfer coefficent.
- double const_Dc (int i, int l, const void ∗data)

  Constant surface diffusivity function.
- double simple_darken_Dc (int i, int l, const void ∗data)

  Simple Darken model for surface diffusivity.
- double theoretical_darken_Dc (int i, int l, const void ∗data)

  Theoretical Darken model for surface diffusivity.
- double empirical_kf (int i, const void ∗data)

Empirical function for film mass transfer coefficent.

- double const_kf (int i, const void *data)

  Constant function for film mass transfer coefficent.

- int molefractionCheck (SKUA_DATA *skua_dat)

  Function to check mole fractions in gas and solid phases for errors.

- int setup_SKUA_DATA (FILE *file, double(*eval_Dc)(int i, int l, const void *user_data), double(*eval_Kf)(int i, const void *user_data), const void *user_data, MIXED_GAS *gas_data, SKUA_DATA *skua_dat)

  Function to setup the function pointers and vector objects in memory to setup the SKUA simulation.

- int SKUA_Executioner (SKUA_DATA *skua_dat)

  Function to execute preprocesses, solvers, and postprocesses for a SKUA simulation.

- int set_SKUA_ICs (SKUA_DATA *skua_dat)

  Function to establish the initial conditions of adsorption in the adsorbent.

- int set_SKUA_timestep (SKUA_DATA *skua_dat)

  Function to establish the time step for the current simulation.

- int SKUA_preprocesses (SKUA_DATA *skua_dat)

  Function to perform the necessary preprocess operations before a solve.

- int set_SKUA_params (const void *user_data)

  Function to call the diffusivity function during the solve.

- int SKUA_postprocesses (SKUA_DATA *skua_dat)

  Function to perform the necessary postprocess operations after a solve.

- int SKUA_reset (SKUA_DATA *skua_dat)

  Function to reset the stateful information in SKUA after a simulation.

- int SKUA (SKUA_DATA *skua_dat)

  Function to iteratively call all execution steps to evolve a simulation through time.

- int SKUA_SCENARIOS (const char *scene, const char *sorbent, const char *comp, const char *sorbate)

  Function callable from the UI to perform a SKUA simulation based on user supplied input files.

- int SKUA_TESTS ()

  Function to perform a test of the SKUA functions and routines.

6.19.1 Detailed Description

Surface Kinetics for Uptake by Adsorption.

skua.cpp

This file contains structures and functions associated with solving the surface diffusion partial differential equations for adsorption kinetics in spherical and/or cylindrical adsorbents. For this system, it is assumed that the pore size is so small that all molecules are confined to movement exclusively on the surface area of the adsorbent. The total amount of adsorption for each species is drive by the MAGPIE model for non-ideal mixed gas adsorption. Spatial and temporal varience in adsorption is caused by a combination of different kinetics between adsorbing species and different adsorption affinities for the surface.

The function for surface diffusion involves four parameters, although not all of these parameters are required to be used. Surface diffusion theoretically varies with temperature according to the Arrhenius rate expression, but we also add in an empirical correction term to account for variations in diffusivity with the partial pressure of the species in the gas phase.
D_surf = D_ref * exp(-E / (R*T)) * pow(p , (T_ref/T) - B)
D_ref is the Reference Diffusivity (um^2/hr), E is the activation energy for adsorption (J/mol), R is the gas law constant (J/K/mol), T is the system temperature (K), p is the partial pressure of the adsorbing species (kPa), T_ref is the Reference Temperature (K), and B is the Affinity constant.

Author

Austin Ladshaw

Date

01/26/2015

Copyright

6.19.2 Macro Definition Documentation

6.19.2.1 #define SKUA_HPP_

6.19.2.2 #define D_inf(*Dref, Tref, B, p, T*) (Dref ∗ pow(p+sqrt(**DBL_EPSILON**),(Tref/T)-B))

Empirical correction of diffusivity (um^2/hr)

6.19.2.3 #define D_o(*Diff, E, T*) (Diff ∗ exp(-E/(**Rstd**∗T)))

Arrhenius Rate Expression for Diffusivity (um^2/hr)

6.19.2.4 #define D_c(*Diff, phi*) (Diff ∗ (1.0/((1.0+1.1E-6)-phi)))

Approximate Darken Diffusivity Equation (um^2/hr)

6.19.3 Function Documentation

6.19.3.1 void print2file_species_header (FILE ∗ *Output,* **SKUA_DATA** ∗ *skua_dat,* int *i*)

Function to print out the species' headers to output file.

6.19.3.2 void print2file_SKUA_time_header (FILE ∗ *Output,* **SKUA_DATA** ∗ *skua_dat,* int *i*)

Function to print out time and space headers to output file.

6.19.3.3 void print2file_SKUA_header (**SKUA_DATA** ∗ *skua_dat*)

Function calls the other header functions to establish output file structure.

6.19.3.4 void print2file_SKUA_results_old (**SKUA_DATA** ∗ *skua_dat*)

Function to print out the old time step simulation results to the output file.

6.19.3.5 void print2file_SKUA_results_new (**SKUA_DATA** ∗ *skua_dat*)

Function to print out the new time step simulation results to the output file.

6.19.3.6 double default_Dc (int *i,* int *l,* const void ∗ *data*)

Default function for surface diffusivity.

This is the default function provided by SKUA for the calculation of the surface diffusivity parameter. The diffusivity is calculated based on the Arrhenius rate expression, then corrected for using the empirical correction term with the outside partial pressure of the gas species.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.7 double default_kf (int *i*, const void * *data*)

Default function for film mass transfer coefficent.

This is the default function provided by SKUA for the calculation of the film mass transfer parameter. By default, we are usually going to couple the SKUA model with a pore diffusion model (see scopsowl.h). Therefore, the film mass transfer coefficient would be zero, because we would only consider a Dirichlet boundary condition for this sub-problem.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.8 double const_Dc (int *i*, int *l*, const void * *data*)

Constant surface diffusivity function.

This function allows the user to specify just a single constant value for surface diffusivity. The value of diffusivity applied at all nodes will be the ref_diffusion parameter in SKUA_PARAM.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.9 double simple_darken_Dc (int *i*, int *l*, const void * *data*)

Simple Darken model for surface diffusivity.

This function uses an approximation to Darken's model for surface diffusion. The approximation is exact if the isotherm for adsorption takes the form of the Langmuir model, but is only approximate if the isotherm is heterogeneous. Forming the approximation in this manner is significantly cheaper than forming the true Darken model expression for the GSTA isotherm.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.10 double theoretical_darken_Dc (int *i*, int *l*, const void * *data*)

Theoretical Darken model for surface diffusivity.

This function uses the full theoretical expression of the Darken's diffusion model to calculate the surface diffusivity. This calculation involves formulating the reference state pressures for the adsorbed amount at every node, then calculating derivatives of the adsorption isotherm for each species. It is more accurate than the simple Darken model function, but costs significantly more computational time.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.11 double empirical_kf (int *i*, const void * *data*)

Empirical function for film mass transfer coefficent.

This function provides an empirical estimate of the mass transfer coefficient using the gas velocity, molecular diffusivities, and dimensionless numbers (see egret.h). It is used as the default film mass transfer function IF the boundary condition is specified to be a Neumann type boundary by the user.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.12 double const_kf (int *i*, const void * *data*)

Constant function for film mass transfer coefficent.

This function allows the user to specify a constant value for the film mass transfer coefficient. The value of the film mass transfer coefficient will be the value of film_transfer given in the SKUA_PARAM data structure.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
data	pointer to the SKUA_DATA structure

6.19.3.13 int molefractionCheck (SKUA_DATA * *skua_dat*)

Function to check mole fractions in gas and solid phases for errors.

This function is called after reading input and before calling the primary solution routines. It will force and error and quit the program if their are inconsistencies in the mole fractions it was given. All mole fractions must sum to 1, otherwise there is missing information.

6.19.3.14 int setup_SKUA_DATA (FILE * *file*, double(*)(int i, int l, const void *user_data) *eval_Dc*, double(*)(int i, const void *user_data) *eval_Kf*, const void * *user_data*, MIXED_GAS * *gas_data*, SKUA_DATA * *skua_dat*)

Function to setup the function pointers and vector objects in memory to setup the SKUA simulation.

This function is called to setup the SKUA problem in memory and set function pointers to either defaults or user specified functions. It must be called prior to calling any other SKUA function and will report an error if the object was not setup properly.

Parameters

file	pointer to the output file for SKUA simulations
eval_Dc	pointer to the function to evaluate the surface diffusivity
eval_Kf	pointer to the function to evaluate the film mass transfer coefficient
user_data	pointer to a user defined data structure used in the calculation the the parameters
gas_data	pointer to the MIXED_GAS data structure for egret.h calculations
skua_dat	pointer to the SKUA_DATA data structure

6.19.3.15 int SKUA_Executioner (SKUA_DATA * *skua_dat*)

Function to execute preprocesses, solvers, and postprocesses for a SKUA simulation.

This function calls the preprocess, solver, and postprocess functions to complete a single time step in a SKUA simulation. User's will want to call this function whenever a time step simulation result is needed. This is used primarily when coupling with other models (see scopsowl.h).

6.19.3.16 int set_SKUA_ICs (SKUA_DATA * *skua_dat*)

Function to establish the initial conditions of adsorption in the adsorbent.

This function needs to be called before doing any simulation or execution of a time step, but only once per simulation. It sets the value of adsorption for each adsorbable species to the specified initial values given via qT and xIC in SKUA_DATA.

6.19.3.17 int set_SKUA_timestep (SKUA_DATA * *skua_dat*)

Function to establish the time step for the current simulation.

This function is called to set a time step value for a particular simulation step. By default, the time step is set to (1/4)x space step size. If you need to change the step size, you must do so manually.

6.19.3.18 int SKUA_preprocesses (SKUA_DATA * *skua_dat*)

Function to perform the necessary preprocess operations before a solve.

This function performs preprocess operations prior to calling the solver routine. Those preprocesses include establishing boundary conditions and performing a MAGPIE simulation for the adsorption on the surface (see magpie.h).

6.19.3.19 int set_SKUA_params (const void * *user_data*)

Function to call the diffusivity function during the solve.

This is the function passed into FINCH to be called during the FINCH solver (see finch.h). It will call the diffusion functions set by the user in the setup function above. This is not overridable.

6.19.3.20 int SKUA_postprocesses (SKUA_DATA * *skua_dat*)

Function to perform the necessary postprocess operations after a solve.

This function performs postprocess operations after a solve was completed successfully. Those operations include estimating average total adsorption, average adsorbed mole fractions, and heat of adsorption for each species. Results are then printed to the output file.

6.19.3.21 int SKUA_reset (SKUA_DATA * *skua_dat*)

Function to reset the stateful information in SKUA after a simulation.

This function sets all the old state data to the newly formed state data. It needs to be called after a successful execution of the simulation step and before calling for the next time step to be solved. Do not call out of turn, otherwise information will be lost.

6.19.3.22 int SKUA (SKUA_DATA * *skua_dat*)

Function to iteratively call all execution steps to evolve a simulation through time.

This function is used in conjunction with the scenario call from the UI to numerically solve the adsorption kinetics problem in time. It will call the initial conditions function once, then iteratively call the reset, time step, and executioner functions for SKUA to push the simulation forward in time. This function will be called from the SKUA_SCENARIOS function.

6.19.3.23 int SKUA_SCENARIOS (const char * *scene,* const char * *sorbent,* const char * *comp,* const char * *sorbate*)

Function callable from the UI to perform a SKUA simulation based on user supplied input files.

This is the primary function to be called when running a stand-alone SKUA simulation. Parameters and system information for the simulation are given in a series of input files that come in as character arrays. These inputs are all required to call this function.

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File

Note

Each input file has a particular format that must be strictly adhered to in order for the simulation to be carried out correctly. The format for each input file, and an example, is provided below...

Scenario Input Format

System Temperature (K) [tab] Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Simulation Time (hrs) [tab] Print Out Time (hrs)
BC Type (0 = Neumann, 1 = Dirichlet)
Number of Gas Species
Initial Total Adsorption (mol/kg)

Name of ith Species [tab] Adsorbable? (0 = false, 1 = true) [tab] Gas Phase Molefraction [tab] Initial Sorbed Molefraction
(repeat above for all species)

Example Scenario Input

353.15 101.35 0.36
4.0 0.05
0
5
0.0
N2 0 0.7634 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0191 0.0
Above example is for a 5-component mixture of N2, O2, Ar, CO2, and H2O, but we are only considering the H2O as adsorbable.

Adsorbent Input File

Domain Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (um) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (um)

Example Adsorbent Input

1 6.0
2.0
Above example is for a cylindrical adsorbent with a length of 5 um and radius of 2 um.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846
0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is a continuation of the Scenario Input example wherein each grouping represents parameters that are associated with N2, O2, Ar, CO2, and H2O, respectively. The order is VERY important!

Adsorbate Input File

Type of Surface Diffusion Function (0 = constant, 1 = simple Darken, 2 = theoretical Darken)
Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0
0.8814 0.0
267.999 0.0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
1.28 540.1
374.99 0.01
3.01
1.27
2
-46597.5 -53.6994
-125024 -221.073
Above example would be for a simulation involving two adsorbable species using a constant surface diffusion function. Each adsorbable species has it's own set of kinetic and equilibrium parameters that must be given in the same order as the species appeared in the Scenario Input. Note: we do not need to supply this information for non-adsorbable species.

6.19.3.24 int SKUA_TESTS ( )

Function to perform a test of the SKUA functions and routines.

This function is callable from the UI and will perform a test simulation of the SKUA system of equations. Results from that test are output into a sub-directory called output and named SKUA_Test_Output.txt.

6.20 skua_opt.h File Reference

Optimization Routine for the SKUA Model.

```
#include "skua.h"
```

Classes

- struct SKUA_OPT_DATA

  Data structure for the SKUA Optimization Routine.

Functions

- int SKUA_OPT_set_y (SKUA_OPT_DATA *skua_opt)

  Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.
- int initial_guess_SKUA (SKUA_OPT_DATA *skua_opt)

  Function to set up an initial guess for the surface diffusivity parameter in SKUA.
- void eval_SKUA_Uptake (const double *par, int m_dat, const void *data, double *fvec, int *info)

  Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.
- int SKUA_OPTIMIZE (const char *scene, const char *sorbent, const char *comp, const char *sorbate, const char *data)

  Function called to perform the optimization routine given a specific set of information and data.

6.20.1 Detailed Description

Optimization Routine for the SKUA Model.

skua_opt.cpp

This file contains structures and functions associated with performing non-linear least-squares optimization of the SKUA simulation results against actual kinetic adsorption data. The optimization routine here allows you to run data comparisons and optimizations in three forms: (i) Rough optimizations - cheaper operations, but less accurate, (ii) Exact optmizations - much more expensive, but greater accuracy, and (iii) data/model comparisons - no optimization, just using system parameters to compare simulation results agains a set of data.

Depending on the level of optimization desired, this routine could take several minutes or several hours. The optimization/comparisons are printed out in two files: (i) a parameter file, which contains the simulation partial pressures and temperatures and the optimized diffusivities with the euclidean norm of the fitting and (ii) a comparison file that shows the model value and data value at each time step for each kinetic curve.

The optimized diffusion parameters are given for each individual kinetic data curve. Each data curve will have a different pairing of partial pressure and temperature. Because of this, you will get a list of different diffusivities for each data curve. To get the optimum kinetic parameters from this list of diffusivities, you must fit the diffusion parameter values to the following diffusion function model...
D_opt = D_ref * exp(-E / (R*T)) * pow(p , (T_ref/T) - B)
where D_ref is the Reference Diffusivity (um^2/hr), E is the activation energy for adsorption (J/mol), R is the gas law constant (J/K/mol), T is the system temperature (K), p is the partial pressure of the adsorbing species (kPa), T_ref is the Reference Temperature (K), and B is the Affinity constant. This algorithm does not automatically produce these parameters for you, but gives you everything you need to produce them yourself.
This routine allows you to optimize multiple kinetic curves at one time. However, all data must be for the same adsorbent-adsorbate system. In other words, the adsorbent and adsorbate pair must be the same for each kinetic curve analyzed. Also, each experiment must have been done in a thin bed or continuous flow system where the adsorbents were exposed to a nearly constant outside partial pressure for all time steps and the gas velocity of that system is assumed constant for all experiments. This experimental setup is very typical for studying adsorption kinetics for gas-solid systems.

Author

Austin Ladshaw

Date

05/11/2015

Copyright

6.20.2 Function Documentation

6.20.2.1 int SKUA_OPT_set_y (SKUA_OPT_DATA * *skua_opt*)

Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.

This function takes the current mole fraction of the adsorbing gas and calculates the gas mole fractions of the other gases in the sytem based on the standard inlet gas composition given in the scenario file.

6.20.2.2 int initial_guess_SKUA (SKUA_OPT_DATA * *skua_opt*)

Function to set up an initial guess for the surface diffusivity parameter in SKUA.

This function performs the Rough optimization on the surface diffusivity based on the idea of reducing or eliminating function bias between data and simulation. A positive function bias means that the simulation curve is "higher" than the data curve and a negative function bias means that the simulation curve is "lower" than the data curve. We use this information to incrementally adjust the rate of surface diffusion until this bias is near zero. When bias is near zero, the simulation is nearly optimized, but further refinement may be necessary to find the true minimum solution.

6.20.2.3 void eval_SKUA_Uptake (const double * *par,* int *m_dat,* const void * *data,* double * *fvec,* int * *info*)

Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.

This function will run the SKUA simulation at a given value of surface diffusivity and produce residuals that feed into the Levenberg-Marquardt's algorithm for non-linear least-squares regression. The form of this function is specific to the format required by the lmfit routine.

Parameters

par	array of parameters that are to be optimized
m_dat	number of data points or functions to evaluate
data	user supplied data structure holding information necessary to form the residuals
fvec	array of residuals computed at the current parameter values
info	integer pointer denoting whether or not the user requests to end a particular simulation

6.20.2.4 int SKUA_OPTIMIZE (const char * *scene,* const char * *sorbent,* const char * *comp,* const char * *sorbate,* const char * *data*)

Function called to perform the optimization routine given a specific set of information and data.

This is the function that is callable by the UI. The user must provide 5 input files to the routine in order to establish simulation conditions, adsborbent properties, component properties, adsorbate equilibrium parameters, and the set of data that we are comparing the simulations to. Each input file has a very specific structure and order to the information that it contains. The structure here is DIFFERENT than the structure for just running standard SKUA simulations (see skua.h).

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File
data	Kinetic Adsorption Data File

Note

Much of the structure of these input files are "similar" to that of the input files used in SKUA_SCENARIOS (see skua.h), but with some notable differences. Below gives the format for each input file with an example. Make sure your input files follow this format before calling this routine from the UI.

Scenario Input File

Optimization? (0 = false, 1 = true) [tab] Rough Optimization? (0 = false, 1 = true)
Surf. Diff. (0 = constant, 1 = simple Darken, 2 = theoretical Darken) [tab] BC Type (0 = Neumann, 1 = Dirichlet)
Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Number of Gaseous Species
Initial Adsorption Total (mol/kg)
Name [tab] Adsorbable? (0 = false, 1 = true) [tab] Inlet Gas Mole Fraction [tab] Initial Adsorbed Mole Fraction
(NOTE: The above line is repeated for all species in gas phase. Also, this algorithm only allows you to consider one adsorbable gas component. Inlet gas mole fractions must be non-zero for all non-adsorbing gases and must sum to 1.)

Example Scenario Input

1 0
0 0
101.35 0.36
5
0.0
N2 0 0.7825 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0 0.0
Above example is for running optimizations on data collected with a gas stream at 0.36 cm/s with 5 gas species in the mixture, only H2O of which is adsorbing. The "base line" or "inlet gas" without H2O has a composition of N2 at 0.7825, O2 at 0.2081, Ar at 0.009, and CO2 at 0.0004.

Adsorbent Input File

Domain Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (um) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (um)

Example Adsorbent Input

1 6.0
2.0
Above example is for a cylindrical adsorbent with a length of 5 um and radius of 2 um.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846
0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is exactly the same as in the SCOPSOWL_SCENARIO example (see scopsowl.h). There is no difference in the input file formats for this input. Keep in mind that the order is VERY important! All species information must be in the same order that the species appeared in the Scenario input file.

Adsorbate Input File

Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0 0
0 0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
Above example gives the equilibrium parameters associated with the H2O-MS3A single component adsorption system. Note that the kinetic parameters (Ref. Diff., Act. Energy, Ref. Temp., and Affinity) were all given a value of zero. These values are irrelavent if we are running an optimization because they will be replaced with a single estimate for the diffusivity that is being optimization for. However, if we wanted to run this routine with comparisons and not do any optmization, then you would need to provide non-zero values for these parameters (at least for Ref. Diff.).

Data Input File

Number of Kinetic Data Curves
Number of data points in the ith curve
Temperature (K) [tab] Partial Pressure (kPa) [tab] Equilibrium Adsorption (mol/kg) all of ith curve
Time point 1 (hrs) [tab] Adsorption 1 (mol/kg) of ith curve
Time point 1 (hrs) [tab] Adsorption 2 (mol/kg) of ith curve
... (Repeat for all time-adsorption data points)
(Repeat above for all curves i)

Example Data Input

40
2990
298.15 0.000310922 2.9
0 0
0.166666667 0.001834419
0.333611111 0.004880247
0.5 0.008306803
...
2789
298.15 0.00055189 5
0 0
0.166944444 0.003350185
0.333611111 0.007418267
0.5 0.009930906
0.666666667 0.014597236
0.833611111 0.021377373
....
Above is a partial example for a data set of 40 kinetic curves. The first curve contains 2990 data points and has temperature of 298.15 K, partial pressure of 0.000310922 kPa, and an equilibrium adsorption of 2.9. Each first time point should start from 0 hours and each initial adsorption should correspond to the value of initial adsorption indicated in the Scenario input file. Then, this structure is repeated for all adsorptio curves.

6.21 Trajectory.h File Reference

Single Particle Trajectory Analysis for Magnetic Filtration.

```
#include "macaw.h"
#include <random>
#include <chrono>
```

Classes

- struct TRAJECTORY_DATA

Functions

- double Magnetic_R (const Matrix< double > &dX, const Matrix< double > &dY, int i, double b, double mu_0, double chi_p, double M, double H0, double a)
- double Magnetic_T (const Matrix< double > &dX, const Matrix< double > &dY, int i, double b, double mu_0, double chi_p, double M, double H0, double a)
- double Grav_R (const Matrix< double > &dX, int i, double b, double rho_p, double rho_f)
- double Grav_T (const Matrix< double > &dX, int i, double b, double rho_p, double rho_f)
- double Van_R (const Matrix< double > &dX, const Matrix< double > &dY, int i, double Hamaker, double b, double a)
- double V_RAD (const Matrix< double > &dX, const Matrix< double > &dY, int i, double V0, double rho_f, double a, double eta)
- double V_THETA (const Matrix< double > &dX, const Matrix< double > &dY, int i, double V0, double rho_f, double a, double eta)
- double Brown_RAD (double n_rand, double m_rand, double sigma_n, double sigma_m)

- double Brown_THETA (double s_rand, double t_rand, double sigma_n, double sigma_m)
- int POLAR (Matrix< double > &POL, const Matrix< double > &dX, const Matrix< double > &dY, const void *data, int i)
- double In_PVel_Rad (const Matrix< double > &POL)
- double In_PVel_Theta (const Matrix< double > &POL)
- int In_P_Velocity (const Matrix< double > &POL, Matrix< double > &Vr, Matrix< double > &Vt)
- double PVel_Rad (const Matrix< double > &POL, const Matrix< double > &Vr, int i, double mp, double beta, double t, double sigma_v, double rand_n)
- double PVel_Theta (const Matrix< double > &POL, const Matrix< double > &Vt, int i, double mp, double beta, double t, double sigma_v, double rand_s)
- int P_Velocity (const Matrix< double > &POL, Matrix< double > &Vr, Matrix< double > &Vt, int i, const void *data)
- double RADIAL_FORCE (const Matrix< double > &POL, const Matrix< double > &Vr, int i, double beta, double mp, double dt, double a)
- double TANGENTIAL_FORCE (const Matrix< double > &POL, const Matrix< double > &Vt, const Matrix< double > &dY, int i, double beta, double mp, double dt, double a)
- double Capture_Force (const Matrix< double > &POL, const Matrix< double > &Vr, int i, double beta, double mp, double dt, double a)
- int CARTESIAN (const Matrix< double > &POL, const Matrix< double > &Vr, const Matrix< double > &Vt, Matrix< double > &H, const Matrix< double > &dY, int i, const void *data)
- int DISPLACEMENT (Matrix< double > &dX, Matrix< double > &dY, const Matrix< double > &H, int i)
- int LOCATION (const Matrix< double > &dY, const Matrix< double > &dX, Matrix< double > &X, Matrix< double > &Y, int i)
- double Removal_Efficiency (double Sum_Cap, const void *data)
- int Trajectory_SetupConstants (TRAJECTORY_DATA *dat)
- int Number_Generator (TRAJECTORY_DATA *dat)
- int Run_Trajectory ()

  Run_Trajectory function.

6.21.1 Detailed Description

Single Particle Trajectory Analysis for Magnetic Filtration.

Trajectory.cpp

Alex, Please provide details here... and elsewhere in the file.

Author

Alex Wiechert

Date

08/25/2015

Copyright

6.21.2 Function Documentation

6.21.2.1 double Magnetic_R (const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *b,* double *mu_0,* double *chi_p,* double *M,* double *H0,* double *a*)

6.21.2.2 double Magnetic_T (const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *b,* double *mu_0,* double *chi_p,* double *M,* double *H0,* double *a*)

6.21.2.3 double Grav_R (const Matrix< double > & *dX,* int *i,* double *b,* double *rho_p,* double *rho_f*)

6.21.2.4 double Grav_T (const Matrix< double > & *dX,* int *i,* double *b,* double *rho_p,* double *rho_f*)

6.21.2.5 double Van_R (const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *Hamaker,* double *b,* double *a*)

6.21.2.6 double V_RAD (const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *V0,* double *rho_f,* double *a,* double *eta*)

6.21.2.7 double V_THETA (const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *V0,* double *rho_f,* double *a,* double *eta*)

6.21.2.8 double Brown_RAD (double *n_rand,* double *m_rand,* double *sigma_n,* double *sigma_m*)

6.21.2.9 double Brown_THETA (double *s_rand,* double *t_rand,* double *sigma_n,* double *sigma_m*)

6.21.2.10 int POLAR (Matrix< double > & *POL,* const Matrix< double > & *dX,* const Matrix< double > & *dY,* const void * *data,* int *i*)

6.21.2.11 double In_PVel_Rad (const Matrix< double > & *POL*)

6.21.2.12 double In_PVel_Theta (const Matrix< double > & *POL*)

6.21.2.13 int In_P_Velocity (const Matrix< double > & *POL,* Matrix< double > & *Vr,* Matrix< double > & *Vt*)

6.21.2.14 double PVel_Rad (const Matrix< double > & *POL,* const Matrix< double > & *Vr,* int *i,* double *mp,* double *beta,* double *t,* double *sigma_v,* double *rand_n*)

6.21.2.15 double PVel_Theta (const Matrix< double > & *POL,* const Matrix< double > & *Vt,* int *i,* double *mp,* double *beta,* double *t,* double *sigma_v,* double *rand_s*)

6.21.2.16 int P_Velocity (const Matrix< double > & *POL,* Matrix< double > & *Vr,* Matrix< double > & *Vt,* int *i,* const void * *data*)

6.21.2.17 double RADIAL_FORCE (const Matrix< double > & *POL,* const Matrix< double > & *Vr,* int *i,* double *beta,* double *mp,* double *dt,* double *a*)

6.21.2.18 double TANGENTIAL_FORCE (const Matrix< double > & *POL,* const Matrix< double > & *Vt,* const Matrix< double > & *dY,* int *i,* double *beta,* double *mp,* double *dt,* double *a*)

6.21.2.19 double Capture_Force (const Matrix< double > & *POL,* const Matrix< double > & *Vr,* int *i,* double *beta,* double *mp,* double *dt,* double *a*)

6.21.2.20 int CARTESIAN (const Matrix< double > & *POL,* const Matrix< double > & *Vr,* const Matrix< double > & *Vt,* Matrix< double > & *H,* const Matrix< double > & *dY,* int *i,* const void * *data*)

6.21.2.21 int DISPLACEMENT (Matrix< double > & *dX,* Matrix< double > & *dY,* const Matrix< double > & *H,* int *i*)

6.21.2.22 int LOCATION (const Matrix< double > & *dY,* const Matrix< double > & *dX,* Matrix< double > & *X,* Matrix< double > & *Y,* int *i*)

6.21.2.23 double Removal_Efficiency (double *Sum_Cap,* const void * *data*)

6.21.2.24 int Trajectory_SetupConstants (TRAJECTORY_DATA * *dat*)

6.21.2.25 int Number_Generator (TRAJECTORY_DATA * *dat*)

6.21.2.26 int Run_Trajectory ( )

Run_Trajectory function.

Function to run the Trajectory project.

6.22 ui.h File Reference

User Interface for Ecosystem.

```
#include <fstream>
#include <string>
#include <iostream>
#include "error.h"
#include "yaml_wrapper.h"
#include "flock.h"
#include "school.h"
#include "sandbox.h"
#include "Trajectory.h"
```

Classes

- struct UI_DATA

  Data structure holding the UI arguments.

Macros

- #define UI_HPP_
- #define ECO_VERSION "1.0.0"

  Macro expansion for executable current version number.
- #define ECO_EXECUTABLE "eco"

  Macro expansion for executable current name.

Enumerations

- enum valid_options {
  TEST, EXECUTE, EXIT, CONTINUE,
  HELP, dogfish, eel, egret,
  finch, lark, macaw, mola,
  monkfish, sandbox, scopsowl, shark,
  skua, gsta_opt, magpie, scops_opt,
  skua_opt, trajectory, dove }

  Valid options available upon execution of the code.

Functions

- void aui_help ()

  Function to display help for Advanced User Interface.
- void bui_help ()

  Function to display help for Basic User Interface.
- bool exit (const std::string &input)

  Function returns true if user requests exit.
- bool help (const std::string &input)

  Function returns trun if the user requests help.
- bool version (const std::string &input)

  Function returns true if user requests to know the executable version.
- bool test (const std::string &input)

  Function returns true if user requests to run a test.
- bool exec (const std::string &input)

  Function returns true if the user requests to run a simulation/executable.
- bool path (const std::string &input)

  Function returns true if the user indicates that input files share a common path.
- bool input (const std::string &input)

  Function returns true if the user indicates that the next arguments are input files.
- bool valid_test_string (const std::string &input, UI_DATA *ui_dat)

  Function returns true if the user gave a valid test option.
- bool valid_exec_string (const std::string &input, UI_DATA *ui_dat)

  Function returns true if the user gave a valid execution option.
- int number_files (UI_DATA *ui_dat)

  Function returns the number of expected input files for the user's run option.
- bool valid_addon_options (UI_DATA *ui_dat)

  Function returns true if the user has choosen a valid additional runtime option.
- void display_help (UI_DATA *ui_dat)

  Function to call the appropriate help menu based on type of interface.
- void display_version (UI_DATA *ui_dat)

  Function to display ecosystem version information to the console.
- int invalid_input (int count, int max)

  Function returns a CONTINUE or EXIT when invalid input is given.
- bool valid_input_main (UI_DATA *ui_dat)

  Function returns true if user gave valid input in Basic UI.
- bool valid_input_tests (UI_DATA *ui_dat)

  Function returns true if user gave a valid test function to run.
- bool valid_input_execute (UI_DATA *ui_dat)

  Function returns true if user gave a valid executable function to run.

- int test_loop (UI_DATA *ui_dat)

  Function that loops the Basic UI until a valid test option was selected.
- int exec_loop (UI_DATA *ui_dat)

  Function that loops the Basic UI until a valid executable option was selected.
- int run_test (UI_DATA *ui_dat)

  Function will call the user requested test function.
- int run_exec (UI_DATA *ui_dat)

  Function will call the user requested executable function.
- int run_executable (int argc, const char *argv[])

  Function called by the main and runs both user interfaces for the program.

6.22.1 Detailed Description

User Interface for Ecosystem.

ui.cpp

These routines define how the user will interface with the software

Author

Austin Ladshaw

Date

08/25/2015

Copyright

6.22.2 Macro Definition Documentation

6.22.2.1 #define UI_HPP_

6.22.2.2 #define ECO_VERSION "1.0.0"

Macro expansion for executable current version number.

6.22.2.3 #define ECO_EXECUTABLE "eco"

Macro expansion for executable current name.

6.22.3 Enumeration Type Documentation

6.22.3.1 enum valid_options

Valid options available upon execution of the code.

Enumeration of valid options for executing the ecosystem code. More options become available as the code updates. Some options that appear here may not be viewable in the "help" screen of the executable. Those options are hidden, but are still valid entries.

Enumerator

TEST
EXECUTE
EXIT
CONTINUE
HELP
dogfish
eel
egret
finch
lark
macaw
mola
monkfish
sandbox
scopsowl
shark
skua
gsta_opt
magpie
scops_opt
skua_opt
trajectory
dove

6.22.4 Function Documentation

6.22.4.1 void aui_help ( )

Function to display help for Advanced User Interface.

The Advanved User Interface help screen is accessed by including run option -h or –help when executing the program from command line.

6.22.4.2 void bui_help ( )

Function to display help for Basic User Interface.

The Basic User Interface help screen is accessed by running the executable, then typing "help" at any point during the console prompts. Exception to this occurs when the console prompts you to provide input files for your choosen routine. In this circumstance, the executable always assumes that what the user types in will be an input file.

6.22.4.3 bool exit (const std::string & *input*)

Function returns true if user requests exit.

This function will check the input string for "exit" or "quit" and terminate the executable. Only checked if using the Basic User Interface.

Parameters

input	input string user gives to the console

6.22.4.4 bool help (const std::string & *input*)

Function returns trun if the user requests help.

This function will check the input string for "help", "-h", or "--help" and will tell the executable to display the help menu. The help menu that gets displayed depends on how the executable was run to begin with.

Parameters

input	input string user gives to the console

6.22.4.5 bool version (const std::string & *input*)

Function returns true if user requests to know the executable version.

This function will check the input string for "version", "-v", or "--version" and will tell the executable to display version information about the executable.

Parameters

input	input string user gives to the console

6.22.4.6 bool test (const std::string & *input*)

Function returns true if user requests to run a test.

This function will check the input string for "-t" or "--test" and determine whether or not the user requests to run an ecosystem test function.

Parameters

input	input string user gives to the console

6.22.4.7 bool exec (const std::string & *input*)

Function returns true if the user requests to run a simulation/executable.

This function will check the input string for "-e" or "--execute" and determine whether or not the user requests to run an ecosystem executable function.

Parameters

input	input string the user gives to the console

6.22.4.8 bool path (const std::string & *input*)

Function returns true if the user indicates that input files share a common path.

This function will check the input string for "-p" or "--path" and determine whether or not the user will give a common path to all input files needed for the specified simulation. Only used in Advanced User Interface.

Parameters

input	input string the user gives to the console

6.22.4.9 bool input (const std::string & *input*)

Function returns true if the user indicates that the next arguments are input files.

This function will check the input string for "-i" or "--input" and determine whether or not the user's next arguments are input files for a specific simulation. Only used in Advanced User Interface.

Parameters

input	input string the user gives to the console

6.22.4.10 bool valid_test_string (const std::string & *input,* UI_DATA * *ui_dat*)

Function returns true if the user gave a valid test option.

This function will check the input string given by the user and determine whether that string denotes a valid test. Then, it will mark the option variable in ui_dat with the appropriate option from the valid_options enum.

Parameters

input	input string the user gives to the console
ui_dat	pointer to the data structure for the ui object

6.22.4.11 bool valid_exec_string (const std::string & *input,* UI_DATA * *ui_dat*)

Function returns true if the user gave a valid execution option.

This function will check the input string given by the user and determine whether that string denotes a valid execution option. Then, it will mark the option variable in ui_dat with the appropriate option from the valid_options enum.

Parameters

input	input string the user gives to the console
ui_dat	pointer to the data structure for the ui object

6.22.4.12 int number_files (UI_DATA * *ui_dat*)

Function returns the number of expected input files for the user's run option.

This function will check the option variable in the ui_dat structure to determine the number of input files that is expected to be given. Running different executable functions in ecosystem may require various number of input files.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.13 bool valid_addon_options (UI_DATA * *ui_dat*)

Function returns true if the user has choosen a valid additional runtime option.

This function will check all additional input options in the user_input variable of ui_dat to determine if the user requests any additional options during runtime. Valid additional options are -p or –path and -i or –input.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.14 void display_help (UI_DATA * *ui_dat*)

Function to call the appropriate help menu based on type of interface.

This function looks at the ui_dat structure and the user's OS files to determine what help menu to display and how to display it. There are two different types of help menus that can be displayed: (i) Advanced Help and (ii) Basic Help. Additionally, this function checks the OS file system for the existence of installed help files. If it finds those files, then it instructs the command terminal to read the contents of those files with the "less" command. Otherwise, it will just print the appropriate help menu to the console window.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.15 void display_version (UI_DATA * *ui_dat*)

Function to display ecosystem version information to the console.

This function will check the ui_dat structure to see which type of interface the user is using, then print out the version information for the executable being run.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.16 int invalid_input (int *count,* int *max*)

Function returns a CONTINUE or EXIT when invalid input is given.

This function looks at the current count and the max iterations and determines whether or not to force the executable to terminate. If the user provides too many incorrect options during the Basic User Interface, then the executable will force quit.

Parameters

count	number of times the user has provided a bad option
max	maximum allowable bad options before force quit

6.22.4.17 bool valid_input_main (UI_DATA * *ui_dat*)

Function returns true if user gave valid input in Basic UI.

This function is only called if the user is running the Basic UI. It checks the given console argument stored in user_input of ui_dat for a valid option. If no valid option is given, then this function returns false.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.18 bool valid_input_tests (UI_DATA * *ui_dat*)

Function returns true if user gave a valid test function to run.

This function checks the user_input argument of ui_dat for a valid test option. If no valid test was given, then this function returns false.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.19 bool valid_input_execute (UI_DATA * *ui_dat*)

Function returns true if user gave a valid executable function to run.

This function checks the user_input argument of ui_dat for a valid executable option. If no valid executable was given, then this function returns false.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.20 int test_loop (UI_DATA * *ui_dat*)

Function that loops the Basic UI until a valid test option was selected.

This function loops the Basic UI menu for running a test until a valid test is selected by the user. If a valid test is not selected, and the maximum number of loops has been reached, then this function will cause the program to force quit.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.21 int exec_loop (UI_DATA * *ui_dat*)

Function that loops the Basic UI until a valid executable option was selected.

This function loops the Basic UI menu for running an executable until a valid executable is selected by the user. If a valid executable is not selected, and the maximum number of loops has been reached, then this function will cause the program to force quit.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.22 int run_test (UI_DATA * *ui_dat*)

Function will call the user requested test function.

This function checks the option variable of the ui_dat structure and runs the corresponding test function.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.23 int run_exec (UI_DATA * *ui_dat*)

Function will call the user requested executable function.

This function checks the option variable of the ui_dat structure and runs the corresponding executable function.

Parameters

ui_dat	pointer to the data structure for the ui object

6.22.4.24 int run_executable (int *argc*, const char * *argv[]*)

Function called by the main and runs both user interfaces for the program.

This function is called in the main.cpp file and passes the console arguments given at run time.

Parameters

argc	number of arguments provided by the user at the time of execution
argv	list of C-strings that was provided by the user at the time of execution

6.23 yaml_wrapper.h File Reference

C++ Wrapper for the C-YAML Library.

```
#include "yaml.h"
#include "error.h"
#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <stdexcept>
```

Classes

- class ValueTypePair

  Value-Type Pair object to recognize data type of a string that was read.
- class KeyValueMap

  Key-Value-Type Map object creating a map of the KeyValuePair objects.
- class SubHeader

  Object for the Lowest level of Header for the yaml_wrapper.
- class Header

  Object for headers in a yaml document (inherits from SubHeader)
- class Document

  Object for the various documents in the yaml file.
- class YamlWrapper

  Object for the entire yaml file holding all documents, header, sub-headers, keys, and values.
- class yaml_cpp_class

  Primary object used when reading and digitally storing yaml files.

Typedefs

- typedef enum data_type data_type

  Enum for valid data types in ValueTypePair.
- typedef enum header_state header_state

  Enum for state of the headers in the yaml_wrapper.

Enumerations

- enum data_type {
  STRING, BOOLEAN, DOUBLE, INT,
  UNKNOWN }
- enum header_state { ANCHOR, ALIAS, NONE }

Functions

- std::string allLower (const std::string &input)

  Function to return an all lower case string based on the passed argument.
- bool isEven (int n)

  Function to return true if the given argument is an even number.
- int YAML_WRAPPER_TESTS ()

  Function to run tests on all the objects that yaml_cpp_class.
- int YAML_CPP_TEST (const char *file)

  Function to run a test read for the yaml_cpp_class on a given file.

6.23.1 Detailed Description

C++ Wrapper for the C-YAML Library.

yaml_wrapper.cpp

This file holds objects, structures, and functions associatied with using the C-YAML library. A C++ wrapper has been created for the Kirill Simonov (2006) LibYAML library to more easily store and query information in yaml style input files. The wrapper uses the C-YAML parser to identify the file structure and store the read in information from that document into an object using C++ maps. Those maps are hold information in a series of Key-Value pairs as well as lists of Key-Value pairs. This allows the user to create well organized input files to change the behavior of simulations.

The yaml_wrapper is restricted to the same limitations in the C-YAML source code in terms of how the documents are allowed to be structured for TOKEN based parsing. C-YAML only recognizes specific tokens and will only allow a certain level of Sub-Header mapping. Therefore, this wrapper has the same limitations. Below is an example of acceptable formatting for a C-YAML document.

#Test input file for YAML and SHARK
TestDoc1: &hat
—

- scenario:
  numvar: 25
  act_fun: DAVIES
  steadystate: FALSE
  t_out: 1
  pH: 0

- testblock:
  another: block

  - subblock:
    sub: block
    ...
    TestDoc2: ∗hat
    —

- masterspecies:
  "Cl - (aq)": 0
  "Na + (aq)": 1
  "H2O (l)": 2
  3: NaCl (aq)
  ...
  TestDoc3:
  —
  apple: red
  pear: green

- array: #Block
  banana: yellow
  #List 1 in array

  - list1: &a #also a block
    a: 1 #key : value
    b: 2
    c: 3
    #List 2 in array
  - list2: ∗a
    a: 4
    b: 5
    c: 6
    ...
    TestDoc4:
    —

- anchor: &anchor
  stuff: to do

- alias: ∗anchor
  add: to stuff

- list:

  - anchored: &list_anchor
    info: blah
    atta: boy

  - aliased: ∗list_anchor
    info: bruh
    atta: ber
    ...
    #WARNING: MAKE SURE FILE DOES NOT CONTAIN TABS!!!

TestDoc5:
—

- grab: ∗anchor
  add2: more adds

  - listcopy: ∗a

- block: {1: 2, 3: 4}
  still: in block
  ...

Note

You can view the actual yaml example file in the input_files/SHARK/test_input.yml sub-directory of the project folder.

Author

Austin Ladshaw

Date

07/29/2015

Copyright

6.23.2 Typedef Documentation

6.23.2.1 typedef enum data_type data_type

Enum for valid data types in ValueTypePair.

6.23.2.2 typedef enum header_state header_state

Enum for state of the headers in the yaml_wrapper.

6.23.3 Enumeration Type Documentation

6.23.3.1 enum data_type

Enumerator

STRING

BOOLEAN

DOUBLE

INT

UNKNOWN

6.23.3.2 enum header_state

Enumerator

ANCHOR

ALIAS

NONE

6.23.4 Function Documentation

6.23.4.1 std::string allLower (const std::string & *input*)

Function to return an all lower case string based on the passed argument.

This function will copy the input paramter and convert that copy to all lower case. The copy is then returned and can be checked against valid or allowed strings.

Parameters

input	string to copy and convert to lower case

6.23.4.2 bool isEven (int *n*)

Function to return true if the given argument is an even number.

6.23.4.3 int YAML_WRAPPER_TESTS ( )

Function to run tests on all the objects that yaml_cpp_class.

This test function is currently NOT callable from the UI.

6.23.4.4 int YAML_CPP_TEST (const char * *file*)

Function to run a test read for the yaml_cpp_class on a given file.

This test/executable function is currently NOT callable from the UI.

Index

~AdsorptionReaction
AdsorptionReaction, 12
~Atom
Atom, 26
~ChemisorptionReaction
ChemisorptionReaction, 41
~Document
Document, 50
~Dove
Dove, 63
~Header
Header, 105
~KeyValueMap
KeyValueMap, 110
~MassBalance
MassBalance, 119
~MasterSpeciesList
MasterSpeciesList, 124
~Matrix
Matrix, 130
~Molecule
Molecule, 141
~MultiligandAdsorption
MultiligandAdsorption, 157
~MultiligandChemisorption
MultiligandChemisorption, 167
~PeriodicTable
PeriodicTable, 178
~Reaction
Reaction, 191
~SubHeader
SubHeader, 230
~UnsteadyAdsorption
UnsteadyAdsorption, 244
~UnsteadyReaction
UnsteadyReaction, 255
~ValueTypePair
ValueTypePair, 264
~YamlWrapper
YamlWrapper, 270
~yaml_cpp_class
yaml_cpp_class, 267

A
magpie.h, 331
a
TRAJECTORY_DATA, 237
A_separator
TRAJECTORY_DATA, 237
A_wire
TRAJECTORY_DATA, 237
ABT
dove.h, 281
ADAPTIVE
dove.h, 280
ADSORBED
mola.h, 346
ALIAS
yaml_wrapper.h, 421
ANCHOR
yaml_wrapper.h, 421
AQUEOUS
mola.h, 346
ARNOLDI_DATA, 21
beta, 22
e1, 23
Hkp1, 23
hp1, 22
iter, 22
k, 22
Output, 23
sum, 23
v, 23
Vk, 23
w, 23
yk, 23
abs_tol_bias
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
AbsPerm
shark.h, 374
act_choice
shark.h, 380
act_fun
AdsorptionReaction, 20
MultiligandAdsorption, 163
MultiligandChemisorption, 172
SHARK_DATA, 214
activation_energy
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
UnsteadyReaction, 262
activities
AdsorptionReaction, 20
MultiligandAdsorption, 163
MultiligandChemisorption, 172
activities_old
UnsteadyAdsorption, 251
activity_data
AdsorptionReaction, 20
MultiligandAdsorption, 163
MultiligandChemisorption, 172
SHARK_DATA, 219
activity_new
SHARK_DATA, 218
activity_old
SHARK_DATA, 218
addDocKey
YamlWrapper, 271
addHeadKey

Document, 52
addKey
KeyValueMap, 111
addPair
Document, 51, 52
Header, 106
KeyValueMap, 111
SubHeader, 231
addSubKey
Header, 107
adjoint
Matrix, 132
ads_rxn
AdsorptionReaction, 21
ChemisorptionReaction, 47
UnsteadyAdsorption, 251
adsorb_index
AdsorptionReaction, 20
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
Adsorbable
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
adsorbent_name
AdsorptionReaction, 21
MultiligandAdsorption, 162
MultiligandChemisorption, 172
AdsorptionList
SHARK_DATA, 212
AdsorptionReaction, 8
∼AdsorptionReaction, 12
act_fun, 20
activities, 20
activity_data, 20
ads_rxn, 21
adsorb_index, 20
adsorbent_name, 21
AdsorptionReaction, 12
aqueous_index, 20
area_factors, 20
AreaBasis, 21
calculateActiveFraction, 15
calculateAqueousChargeExchange, 16
calculateAreaFactors, 15
calculateEquilibria, 15
calculateEquilibriumCorrection, 17
calculateLangmuirAdsorption, 16
calculateLangmuirEquParam, 16
calculateLangmuirMaxCapacity, 15
calculatePsi, 16
calculateSurfaceChargeDensity, 15
callSurfaceActivity, 15
charge_density, 21
checkAqueousIndices, 13
Display_Info, 12
Eval_Residual, 17
getActivity, 18
getActivityEnum, 19
getAdsorbIndex, 19
getAdsorbentName, 19
getAqueousIndex, 19
getAreaFactor, 18
getBulkDensity, 18
getChargeDensity, 18
getIonicStrength, 18
getMolarFactor, 17
getNumberRxns, 18
getReaction, 17
getSpecificArea, 18
getSpecificMolality, 18
getSurfaceCharge, 18
getTotalMass, 18
getTotalVolume, 18
getVolumeFactor, 18
IncludeSurfCharge, 21
includeSurfaceCharge, 19
Initialize_Object, 12
ionic_strength, 21
isAreaBasis, 19
List, 19
modifyDeltas, 12
molar_factor, 20
num_rxns, 21
setActivities, 14
setActivityEnum, 13
setActivityModelInfo, 13
setAdsorbIndices, 13
setAdsorbentName, 14
setAqueousIndex, 13
setAqueousIndexAuto, 13
setAreaBasisBool, 14
setAreaFactor, 13
setBasis, 14
setChargeDensity, 15
setChargeDensityValue, 14
setIonicStrength, 15
setIonicStrengthValue, 14
setMolarFactor, 13
setSpecificArea, 14
setSpecificMolality, 14
setSurfaceCharge, 14
setSurfaceChargeBool, 14
setTotalMass, 14
setTotalVolume, 14
setVolumeFactor, 13
specific_area, 20
specific_molality, 20
surface_activity, 19
surface_charge, 20
total_mass, 20
total_volume, 20
volume_factors, 20
affinity
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
Ai

OPTRANS_DATA, 175
alias
SubHeader, 232
alkalinity
MasterSpeciesList, 126
all_pars
GSTA_OPT_DATA, 102
allLower
yaml_wrapper.h, 421
alpha
BACKTRACK_DATA, 30
BiCGSTAB_DATA, 32
CGS_DATA, 36
GCR_DATA, 87
PCG_DATA, 176
anchor_alias_dne
error.h, 293
Ap
PCG_DATA, 177
aqueous_index
AdsorptionReaction, 20
area_factors
AdsorptionReaction, 20
AreaBasis
AdsorptionReaction, 21
AreaSTD
shark.h, 373
arg
GMRESR_DATA, 93
arg_matrix_same
error.h, 293
argc
UI_DATA, 240
argv
UI_DATA, 240
arnoldi
lark.h, 314
arnoldi_dat
GMRESLP_DATA, 91
As
SYSTEM_DATA, 234
assertType
KeyValueMap, 111
ValueTypePair, 264
Atom, 23
∼Atom, 26
Atom, 26
AtomCategory, 28
AtomName, 27
AtomState, 28
AtomSymbol, 28
atomic_number, 29
atomic_radii, 28
atomic_weight, 28
AtomicNumber, 28
AtomicRadii, 27
AtomicWeight, 27
BondingElectrons, 27
Category, 29
DisplayInfo, 28
editAtomicWeight, 26
editElectrons, 26
editNeutrons, 26
editOxidationState, 26
editProtons, 26
editRadii, 27
editValence, 26
Electrons, 27
electrons, 28
Name, 29
NaturalState, 29
Neutrons, 27
neutrons, 28
oxidation_state, 28
OxidationState, 27
Protons, 27
protons, 28
Register, 26
removeElectron, 27
removeNeutron, 27
removeProton, 27
Symbol, 29
valence_e, 28
AtomCategory
Atom, 28
AtomName
Atom, 27
AtomState
Atom, 28
AtomSymbol
Atom, 28
atomic_number
Atom, 29
atomic_radii
Atom, 28
atomic_weight
Atom, 28
AtomicNumber
Atom, 28
AtomicRadii
Atom, 27
AtomicWeight
Atom, 27
atoms
Molecule, 146
aui_help
ui.h, 411
avg_fiber_density
MONKFISH_DATA, 150
avg_norm
SYSTEM_DATA, 234
avg_sorption
MONKFISH_PARAM, 153
avg_sorption_old
MONKFISH_PARAM, 153
avgDp

scopsowl.h, 354
avgPar
gsta_opt.h, 305
avgValue
gsta_opt.h, 305

b
TRAJECTORY_DATA, 237
B0
TRAJECTORY_DATA, 237
BACKTRACK_DATA, 29
alpha, 30
constRho, 30
Fk, 30
fun_call, 30
lambdaMin, 30
normFkp1, 30
rho, 30
xk, 30
BATCH
shark.h, 374
BDF2
dove.h, 280
BOOLEAN
yaml_wrapper.h, 420
backtrack_dat
PJFNK_DATA, 186
backtrackLineSearch
lark.h, 324
BasicUI
UI_DATA, 240
BE
dove.h, 280
begin
Document, 51
Header, 106
KeyValueMap, 111
YamlWrapper, 270, 271
best_par
GSTA_OPT_DATA, 102
bestres
BiCGSTAB_DATA, 33
CGS_DATA, 37
GCR_DATA, 88
GMRESLP_DATA, 90
GMRESRP_DATA, 96
PCG_DATA, 177
PICARD_DATA, 181
bestx
BiCGSTAB_DATA, 33
CGS_DATA, 37
GCR_DATA, 88
GMRESLP_DATA, 91
GMRESRP_DATA, 96
PCG_DATA, 177
PICARD_DATA, 181
PJFNK_DATA, 185
beta
ARNOLDI_DATA, 22
BiCGSTAB_DATA, 32
CGS_DATA, 36
FINCH_DATA, 80
GCR_DATA, 87
PCG_DATA, 176
TRAJECTORY_DATA, 238
BiCGSTAB_DATA, 31
alpha, 32
bestres, 33
bestx, 33
beta, 32
breakdown, 32
iter, 32
maxit, 32
omega, 32
omega_old, 33
Output, 33
p, 34
r, 33
r0, 33
relres, 33
relres_base, 33
res, 33
rho, 32
rho_old, 32
s, 34
t, 34
tol_abs, 33
tol_rel, 33
v, 34
x, 33
y, 34
z, 34
BiCGSTAB
lark.h, 313
bicgstab
lark.h, 318
bicgstab_dat
PJFNK_DATA, 186
binary_diffusion
MIXED_GAS, 138
binder_fraction
SCOPSOWL_DATA, 198
binder_poresize
SCOPSOWL_DATA, 198
binder_porosity
SCOPSOWL_DATA, 198
BondingElectrons
Atom, 27
Bounce
PJFNK_DATA, 185
breakdown
BiCGSTAB_DATA, 32
CGS_DATA, 36
GCR_DATA, 87
Brown_RAD
Trajectory.h, 407
Brown_THETA

Trajectory.h, 407
BT
dove.h, 281
bui_help
ui.h, 411

c
CGS_DATA, 37
GCR_DATA, 88
c_temp
GCR_DATA, 88
CARTESIAN
Trajectory.h, 407
CC_E
FINCH_DATA, 81
CC_I
FINCH_DATA, 80
CE3
egret.h, 289
CGS_DATA, 34
alpha, 36
bestres, 37
bestx, 37
beta, 36
breakdown, 36
c, 37
iter, 36
maxit, 36
Output, 37
p, 37
r, 37
r0, 37
relres, 36
relres_base, 37
res, 36
rho, 36
sigma, 36
tol_abs, 36
tol_rel, 36
u, 37
v, 37
w, 37
x, 37
z, 38
CGS
lark.h, 313
CL_E
FINCH_DATA, 80
CL_I
FINCH_DATA, 80
CONSTANT
dove.h, 280
CONTINUE
ui.h, 411
CR_E
FINCH_DATA, 81
CR_I
FINCH_DATA, 81
CSTR
shark.h, 374
calculate_ionic_strength
shark.h, 375
calculate_properties
egret.h, 291
calculateActiveFraction
AdsorptionReaction, 15
UnsteadyAdsorption, 247
calculateAqueousChargeExchange
AdsorptionReaction, 16
ChemisorptionReaction, 44
UnsteadyAdsorption, 247
calculateAreaFactors
AdsorptionReaction, 15
ChemisorptionReaction, 43
MultiligandAdsorption, 159
MultiligandChemisorption, 169
UnsteadyAdsorption, 246
calculateAvgOxiState
Molecule, 143
calculateElecticPotential
ChemisorptionReaction, 44
MultiligandAdsorption, 160
MultiligandChemisorption, 169
calculateEnergies
Reaction, 192
UnsteadyReaction, 258
calculateEquilibria
AdsorptionReaction, 15
ChemisorptionReaction, 43
MultiligandAdsorption, 159
MultiligandChemisorption, 169
UnsteadyAdsorption, 246
calculateEquilibrium
Reaction, 192
UnsteadyReaction, 258
calculateEquilibriumCorrection
AdsorptionReaction, 17
ChemisorptionReaction, 44
MultiligandAdsorption, 160
MultiligandChemisorption, 170
UnsteadyAdsorption, 248
calculateLangmuirAdsorption
AdsorptionReaction, 16
calculateLangmuirEquParam
AdsorptionReaction, 16
calculateLangmuirMaxCapacity
AdsorptionReaction, 15
calculateMolarArea
Molecule, 143
calculateMolarVolume
Molecule, 143
calculateMolarWeight
Molecule, 143
calculatePsi
AdsorptionReaction, 16
UnsteadyAdsorption, 247
calculateRate

UnsteadyReaction, 258
calculateRates
UnsteadyAdsorption, 246
calculateSurfaceChargeDensity
AdsorptionReaction, 15
ChemisorptionReaction, 43
UnsteadyAdsorption, 247
callSurfaceActivity
AdsorptionReaction, 15
ChemisorptionReaction, 43
MultiligandAdsorption, 160
MultiligandChemisorption, 169
UnsteadyAdsorption, 247
callroutine
FINCH_DATA, 84
CanCalcHS
Reaction, 194
CanCalcG
Reaction, 194
Cap
TRAJECTORY_DATA, 238
Capture_Force
Trajectory.h, 407
Carrier
SYSTEM_DATA, 235
Cartesian
finch.h, 297
Category
Atom, 29
cgs
lark.h, 319
cgs_dat
PJFNK_DATA, 186
changeKey
Document, 51
Header, 106
YamlWrapper, 271
char_length
MIXED_GAS, 138
char_macro
SCOPSOWL_DATA, 198
char_measure
SKUA_DATA, 222
char_micro
SCOPSOWL_DATA, 198
Charge
Molecule, 144
charge
MasterSpeciesList, 126
Molecule, 145
charge_density
AdsorptionReaction, 21
MultiligandAdsorption, 163
MultiligandChemisorption, 173
check_Mass
finch.h, 297
checkAqueousIndices
AdsorptionReaction, 13
MultiligandAdsorption, 158
UnsteadyAdsorption, 244
CheckMass
FINCH_DATA, 79
CheckMolefractions
MIXED_GAS, 137
checkSpeciesEnergies
Reaction, 192
UnsteadyReaction, 258
ChemisorptionList
SHARK_DATA, 212
ChemisorptionReaction, 38
~ChemisorptionReaction, 41
ads_rxn, 47
calculateAqueousChargeExchange, 44
calculateAreaFactors, 43
calculateElecticPotential, 44
calculateEquilibria, 43
calculateEquilibriumCorrection, 44
calculateSurfaceChargeDensity, 43
callSurfaceActivity, 43
ChemisorptionReaction, 41
Delta, 47
Display_Info, 41
Eval_RxnResidual, 44
Eval_SiteBalanceResidual, 45
getActivity, 45
getActivityEnum, 46
getAdsorbIndex, 46
getAdsorbentName, 46
getAreaFactor, 45
getBulkDensity, 46
getChargeDensity, 46
getDelta, 45
getIonicStrength, 46
getLigandIndex, 46
getNumberRxns, 46
getReaction, 45
getSpecificArea, 45
getSpecificMolality, 46
getTotalMass, 46
getTotalVolume, 46
getVolumeFactor, 45
includeSurfaceCharge, 46
Initialize_Object, 41
ligand_index, 47
modifyMBEdeltas, 41
setActivities, 43
setActivityEnum, 42
setActivityModelInfo, 42
setAdsorbIndices, 41
setAdsorbentName, 43
setAreaFactor, 42
setChargeDensity, 43
setChargeDensityValue, 43
setDelta, 42
setDeltas, 42
setIonicStrength, 43

setIonicStrengthValue, 43
setLigandIndex, 42
setSpecificArea, 42
setSpecificMolality, 42
setSurfaceChargeBool, 43
setTotalMass, 42
setTotalVolume, 43
setVolumeFactor, 42
chi_p
TRAJECTORY_DATA, 237
cleanup
yaml_cpp_class, 267
clear
Document, 51
Header, 106
KeyValueMap, 111
SubHeader, 231
YamlWrapper, 271
CN
dove.h, 280
FINCH_DATA, 79
cofactor
Matrix, 131
columnExtend
Matrix, 136
columnExtract
Matrix, 135
columnProjection
Matrix, 134
columnReplace
Matrix, 135
columnShrink
Matrix, 136
columnVectorFill
Matrix, 134
columns
Matrix, 131
CompareFile
SCOPSOWL_OPT_DATA, 204
SKUA_OPT_DATA, 227
ComputeTimeStep
Dove, 68
Conc_new
SHARK_DATA, 218
Conc_old
SHARK_DATA, 217
Console_Output
SHARK_DATA, 217
const_Dc
skua.h, 395
const_filmMassTransfer
scopsowl.h, 357
const_kf
skua.h, 396
const_pH
SHARK_DATA, 217
const_pore_diffusion
scopsowl.h, 357
constRho
BACKTRACK_DATA, 30
ConstantICFill
Matrix, 133
Contains_pOH
SHARK_DATA, 217
Contains_pH
SHARK_DATA, 217
Converged
Dove, 71
PJFNK_DATA, 185
SHARK_DATA, 217
Convert2Concentration
shark.h, 382
Convert2LogConcentration
shark.h, 382
coord
SKUA_DATA, 221
coord_macro
SCOPSOWL_DATA, 196
coord_micro
SCOPSOWL_DATA, 197
CoordSTD
shark.h, 373
copyAnchor2Alias
Document, 52
Header, 107
YamlWrapper, 271
count
UI_DATA, 240
crystal_radius
SCOPSOWL_DATA, 198
Cstd
egret.h, 289
current_equil
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
current_points
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
current_press
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
current_temp
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
current_token
yaml_cpp_class, 268
Cylindrical
finch.h, 297

d
FINCH_DATA, 77
D_c
skua.h, 394
D_ii
egret.h, 290
D_ij
egret.h, 290

D_inf
skua.h, 394
D_o
skua.h, 394
DAVIES
shark.h, 375
DBL_EPSILON
magpie.h, 330
DEBYE_HUCKEL
shark.h, 375
dHo
GSTA_DATA, 100
DISPLACEMENT
Trajectory.h, 408
DIC
FINCH_DATA, 78
DOGFISH_DATA, 53
DirichletBC, 55
end_time, 55
eval_DI, 56
eval_kf, 56
eval_qs, 56
eval_R, 56
fiber_diameter, 56
fiber_length, 56
fiber_specific_area, 56
finch_dat, 56
NonLinear, 55
NumComp, 55
OutputFile, 56
param_dat, 57
Print2Console, 55
Print2File, 55
t_counter, 55
t_print, 55
time, 55
time_old, 55
total_sorption, 56
total_sorption_old, 56
total_steps, 55
user_data, 56
DOGFISH_Executioner
dogfish.h, 275
DOGFISH_PARAM, 57
film_transfer_coeff, 57
initial_sorption, 58
intraparticle_diffusion, 57
sorbed_molefraction, 58
species, 58
surface_concentration, 58
DOGFISH_TESTS
dogfish.h, 276
DOGFISH_postprocesses
dogfish.h, 276
DOGFISH_preprocesses
dogfish.h, 276
DOGFISH_reset
dogfish.h, 276
DOGFISH
dogfish.h, 276
DOUBLE
yaml_wrapper.h, 420
DOVE_HPP_
dove.h, 279
DOVE_TESTS
dove.h, 286
dSo
GSTA_DATA, 100
Data
Matrix, 136
Data_Map
SubHeader, 232
data_type
yaml_wrapper.h, 420
Davies_equation
shark.h, 379
DebyeHuckel_equation
shark.h, 379
default_Dc
skua.h, 394
default_FilmMTCoeff
dogfish.h, 274
default_IntraDiffusion
dogfish.h, 274
default_Retardation
dogfish.h, 274
default_SurfaceConcentration
dogfish.h, 275
default_adsorption
scopsowl.h, 355
default_bcs
finch.h, 300
default_coeff
dove.h, 286
default_density
monkfish.h, 348
default_effective_diffusion
scopsowl.h, 356
default_execution
finch.h, 299
default_exterior_concentration
monkfish.h, 349
default_film_transfer
monkfish.h, 349
default_filmMassTransfer
scopsowl.h, 357
default_func
dove.h, 285
default_ic
finch.h, 299
default_interparticle_diffusion
monkfish.h, 348
default_jacobi
dove.h, 286
default_kf
skua.h, 395

default_monk_adsorption
monkfish.h, 348
default_monk_equilibrium
monkfish.h, 348
default_monkfish_retardation
monkfish.h, 349
default_params
finch.h, 300
default_pore_diffusion
scopsowl.h, 355
default_porosity
monkfish.h, 347
default_postprocess
finch.h, 301
default_precon
finch.h, 300
default_preprocess
finch.h, 299
default_res
finch.h, 300
default_reset
finch.h, 301
default_retardation
scopsowl.h, 355
default_solve
finch.h, 299
default_surf_diffusion
scopsowl.h, 356
default_timestep
finch.h, 299
Delta
ChemisorptionReaction, 47
MassBalance, 122
density
PURE_GAS, 188
determinate
Matrix, 131
diagonalSolve
Matrix, 134
dielectric_const
SHARK_DATA, 216
diffusion_type
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
dim_mis_match
error.h, 293
Dirichlet
FINCH_DATA, 79
dirichletBCFill
Matrix, 134
DirichletBC
DOGFISH_DATA, 55
MONKFISH_DATA, 149
SCOPSOWL_DATA, 198
SKUA_DATA, 222
discretize
FINCH_DATA, 84
Display
Matrix, 132
Display_Info
AdsorptionReaction, 12
ChemisorptionReaction, 41
MassBalance, 119
MultiligandChemisorption, 167
Reaction, 191
UnsteadyAdsorption, 244
UnsteadyReaction, 255
display_help
ui.h, 414
display_version
ui.h, 414
DisplayAll
MasterSpeciesList, 125
DisplayConcentrations
MasterSpeciesList, 125
DisplayContents
Document, 52
Header, 107
SubHeader, 232
yaml_cpp_class, 267
YamlWrapper, 271
DisplayInfo
Atom, 28
MasterSpeciesList, 125
Molecule, 145
DisplayMap
KeyValueMap, 112
DisplayPair
ValueTypePair, 265
DisplayTable
PeriodicTable, 179
Dk
scopsowl.h, 354
Dn
FINCH_DATA, 83
Dnp1
FINCH_DATA, 83
Do
FINCH_DATA, 78
Doc_Map
YamlWrapper, 272
Document, 47
∼Document, 50
addHeadKey, 52
addPair, 51, 52
begin, 51
changeKey, 51
clear, 51
copyAnchor2Alias, 52
DisplayContents, 52
Document, 50
end, 51
getAlias, 52
getAnchoredHeader, 53
getDataMap, 51
getHeadFromSubAlias, 53

getHeadMap, 51
getHeader, 51
getName, 52
getState, 52
Head_Map, 53
isAlias, 53
isAnchor, 53
operator(), 50
operator=, 50
operator[], 50
resetKeys, 51
revalidateAllKeys, 51
setAlias, 52
setName, 52
setNameAliasPair, 52
setState, 52
size, 52
dog_dat
MONKFISH_DATA, 151
dogfish
ui.h, 411
dogfish.h, 272
DOGFISH_Executioner, 275
DOGFISH_TESTS, 276
DOGFISH_postprocesses, 276
DOGFISH_preprocesses, 276
DOGFISH_reset, 276
DOGFISH, 276
default_FilmMTCoeff, 274
default_IntraDiffusion, 274
default_Retardation, 274
default_SurfaceConcentration, 275
print2file_DOGFISH_header, 274
print2file_DOGFISH_result_new, 274
print2file_DOGFISH_result_old, 274
print2file_species_header, 274
set_DOGFISH_ICs, 275
set_DOGFISH_params, 276
set_DOGFISH_timestep, 276
setup_DOGFISH_DATA, 275
domain_diameter
MONKFISH_DATA, 150
Dove, 58
∼Dove, 63
ComputeTimeStep, 68
Converged, 71
Dove, 63
DoveFileOutput, 71
DoveOutput, 71
dt, 70
dt_old, 70
dtmax, 70
dtmin, 70
Eval_Coeff, 68
Eval_Func, 68
Eval_Jacobi, 68
getCurrentTime, 67
getCurrentU, 67
getEndTime, 67
getJacobiMap, 68
getMaxTimeStep, 68
getMinTimeStep, 68
getNewU, 67
getNonlinearRelativeRes, 68
getNonlinearResidual, 68
getNumFunc, 67
getOldTime, 67
getOlderTime, 67
getOldU, 67
getTimeStep, 67
getTimeStepOld, 67
getUserData, 67
hasConverged, 68
int_sub, 71
int_type, 70
Linear, 71
linmax, 71
newton_dat, 72
num_func, 71
Output, 71
precon, 72
Preconditioner, 71
preconditioner, 71
print_header, 66
print_newresult, 66
print_result, 67
registerCoeff, 66
registerFunction, 65
registerJacobi, 66
reset_all, 69
residual, 72
set_LineSearchMethod, 65
set_LinearAbsTol, 64
set_LinearMethod, 65
set_LinearOutput, 65
set_LinearRelTol, 65
set_LinearStatus, 65
set_MaxLinearIterations, 65
set_MaxNonLinearIterations, 65
set_NonlinearAbsTol, 64
set_NonlinearOutput, 65
set_NonlinearRelTol, 64
set_Preconditioning, 65
set_RecursionLevel, 65
set_RestartLimit, 65
set_defaultCoeffs, 64
set_defaultJacobis, 64
set_endtime, 63
set_fileoutput, 64
set_initialcondition, 64
set_integrationtype, 63
set_numfunc, 63
set_output, 64
set_outputfile, 64
set_preconditioner, 64
set_timestep, 63

set_timestepmax, 63
set_timestepmin, 63
set_timestepper, 63
set_tolerance, 64
set_userdata, 64
solve_FE, 69
solve_RK4, 69
solve_RKF, 69
solve_all, 69
solve_timestep, 68
time, 70
time_end, 70
time_old, 70
time_older, 70
timestepper, 71
tolerance, 70
un, 70
unm1, 70
unp1, 70
update_states, 69
update_timestep, 69
user_coeff, 72
user_data, 72
user_func, 71
user_jacobi, 72
validate_linearsteps, 69
validate_precond, 68
dove
ui.h, 411
dove.h, 277
ABT, 281
ADAPTIVE, 280
BDF2, 280
BE, 280
BT, 281
CONSTANT, 280
CN, 280
DOVE_HPP_, 279
DOVE_TESTS, 286
default_coeff, 286
default_func, 285
default_jacobi, 286
EXPLICIT, 279
FEHLBERG, 280
FE, 280
IMPLICIT, 279
integrate_subtype, 279
integrate_type, 279
JACOBI, 281
LGS, 281
linesearch_type, 280
NO_LS, 281
precond_Jac_BDF2, 284
precond_Jac_BE, 281
precond_Jac_CN, 283
precond_LowerGS_BDF2, 285
precond_LowerGS_BE, 282
precond_LowerGS_CN, 283
precond_SymmetricGS_BDF2, 285
precond_SymmetricGS_BE, 282
precond_SymmetricGS_CN, 284
precond_Tridiag_BDF2, 284
precond_Tridiag_BE, 281
precond_Tridiag_CN, 283
precond_UpperGS_BDF2, 285
precond_UpperGS_BE, 282
precond_UpperGS_CN, 283
precond_type, 281
RK4, 280
RKF, 280
residual_BDF2, 284
residual_BE, 281
residual_CN, 282
SGS, 281
TRIDIAG, 281
timestep_type, 280
UGS, 281
DoveFileOutput
Dove, 71
DoveOutput
Dove, 71
Dp
scopsowl.h, 354
Dp_ij
egret.h, 290
dq_dc
SCOPSOWL_PARAM_DATA, 206
dq_dco
SCOPSOWL_PARAM_DATA, 206
dq_dp
magpie.h, 332
dt
Dove, 70
FINCH_DATA, 77
MassBalance, 122
SHARK_DATA, 215
TRAJECTORY_DATA, 238
dt_max
SHARK_DATA, 215
dt_min
SHARK_DATA, 215
dt_old
Dove, 70
FINCH_DATA, 77
dtmax
Dove, 70
dtmin
Dove, 70
duplicate_variable
error.h, 293
dX
TRAJECTORY_DATA, 238
dxj
NUM_JAC_DATA, 174
dY
TRAJECTORY_DATA, 238

dynamic_viscosity
PURE_GAS, 188
dz
FINCH_DATA, 77

e
shark.h, 373
e0
GMRESRP_DATA, 97
e0_bar
GMRESRP_DATA, 97
e1
ARNOLDI_DATA, 23
e_norm
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
e_norm_old
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
ECO_EXECUTABLE
ui.h, 410
ECO_VERSION
ui.h, 410
EEL_TESTS
eel.h, 287
EGRET_TESTS
egret.h, 291
eMax
mSPD_DATA, 154
magpie.h, 333
EXECUTE
ui.h, 411
EXIT
ui.h, 411
EXPLICIT
dove.h, 279
edit
Matrix, 131
editAllOxidationStates
Molecule, 143
editAtomicWeight
Atom, 26
editCharge
Molecule, 143
editElectrons
Atom, 26
editEnergy
Molecule, 144
editEnthalpy
Molecule, 143
editEntropy
Molecule, 143
editHS
Molecule, 143
editNeutrons
Atom, 26
editOneOxidationState
Molecule, 143
editOxidationState
Atom, 26
editPair
ValueTypePair, 264
editProtons
Atom, 26
editRadii
Atom, 27
editValence
Atom, 26
editValue
ValueTypePair, 264
editValue4Key
KeyValueMap, 111
eduGuess
gsta_opt.h, 306
eel
ui.h, 411
eel.h, 286
EEL_TESTS, 287
egret
ui.h, 411
egret.h, 287
CE3, 289
calculate_properties, 291
Cstd, 289
D_ii, 290
D_ij, 290
Dp_ij, 290
EGRET_TESTS, 291
FilmMTCoeff, 290
initialize_data, 290
Mu, 290
Nu, 290
PE3, 289
PSI, 290
Po, 289
Pstd, 289
RE3, 289
ReNum, 290
Rstd, 289
ScNum, 290
set_variables, 290
ek
QR_DATA, 189
electric_potential
MultiligandAdsorption, 163
MultiligandChemisorption, 173
Electrons
Atom, 27
electrons
Atom, 28
empirical_kf
skua.h, 396
empty_matrix
error.h, 293
end
Document, 51
Header, 106

KeyValueMap, 110, 111
YamlWrapper, 270
end_temp
SHARK_DATA, 215
end_time
DOGFISH_DATA, 55
MONKFISH_DATA, 149
Energy
Molecule, 145
energy
Reaction, 194
Enthalpy
Molecule, 144
enthalpy
Reaction, 193
Entropy
Molecule, 144
entropy
Reaction, 193
eps
NUM_JAC_DATA, 174
PJFNK_DATA, 185
Equilibrium
Reaction, 193
error
error.h, 294
error.h, 291
anchor_alias_dne, 293
arg_matrix_same, 293
dim_mis_match, 293
duplicate_variable, 293
empty_matrix, 293
error, 294
error_type, 292
file_dne, 292
generic_error, 292
indexing_error, 292
initial_error, 293
invalid_atom, 293
invalid_boolean, 293
invalid_components, 292
invalid_console_input, 293
invalid_electron, 293
invalid_fraction, 293
invalid_gas_sum, 293
invalid_molefraction, 293
invalid_neutron, 293
invalid_norm, 293
invalid_proton, 293
invalid_size, 293
invalid_solid_sum, 293
invalid_species, 293
invalid_type, 293
invalid_valence, 293
key_not_found, 293
mError, 292
magpie_reverse_error, 292
matrix_too_small, 293
matvec_mis_match, 293
missing_information, 293
negative_mass, 293
negative_time, 293
no_diffusion, 293
non_real_edge, 293
non_square_matrix, 293
not_a_token, 293
nullptr_error, 293
nullptr_func, 293
opt_no_support, 293
ortho_check_fail, 293
out_of_bounds, 293
read_error, 293
rxn_rate_error, 293
scenario_fail, 293
simulation_fail, 292
singular_matrix, 293
string_parse_error, 293
tensor_out_of_bounds, 293
unregistered_name, 293
unstable_matrix, 293
vector_out_of_bounds, 293
zero_vector, 293
error_type
error.h, 292
eta
mSPD_DATA, 154
TRAJECTORY_DATA, 237
eval_Cex
MONKFISH_DATA, 151
Eval_ChargeResidual
MasterSpeciesList, 126
Eval_Coeff
Dove, 68
eval_Dex
MONKFISH_DATA, 150
eval_DI
DOGFISH_DATA, 56
Eval_Func
Dove, 68
eval_GPAST
magpie.h, 335
eval_GSTA
gsta_opt.h, 308
Eval_IC_Residual
MassBalance, 122
UnsteadyAdsorption, 249
UnsteadyReaction, 261
Eval_Jacobi
Dove, 68
Eval_ReactionRate
UnsteadyAdsorption, 249
UnsteadyReaction, 260
Eval_Residual
AdsorptionReaction, 17
MassBalance, 121
MultiligandAdsorption, 160

Reaction, 193
UnsteadyAdsorption, 248
UnsteadyReaction, 260, 261
eval_Ret
MONKFISH_DATA, 150
Eval_RxnResidual
ChemisorptionReaction, 44
MultiligandChemisorption, 170
eval_SCOPSOWL_Uptake
scopsowl_opt.h, 365
eval_SKUA_Uptake
skua_opt.h, 402
Eval_SiteBalanceResidual
ChemisorptionReaction, 45
MultiligandChemisorption, 170
eval_ads
MONKFISH_DATA, 150
SCOPSOWL_DATA, 199
eval_diff
SCOPSOWL_DATA, 199
SKUA_DATA, 222
eval_eps
MONKFISH_DATA, 150
eval_eta
magpie.h, 335
eval_kf
DOGFISH_DATA, 56
MONKFISH_DATA, 151
SCOPSOWL_DATA, 199
SKUA_DATA, 222
eval_po
magpie.h, 335
eval_po_PI
magpie.h, 334
eval_po_qo
magpie.h, 334
eval_qs
DOGFISH_DATA, 56
eval_R
DOGFISH_DATA, 56
eval_retard
SCOPSOWL_DATA, 199
eval_rho
MONKFISH_DATA, 150
eval_surfDiff
SCOPSOWL_DATA, 199
EvalActivity
SHARK_DATA, 218
evalprecon
FINCH_DATA, 85
evalres
FINCH_DATA, 85
evaluation
SCOPSOWL_OPT_DATA, 201
SKUA_OPT_DATA, 225
exec
ui.h, 412
exec_loop
ui.h, 416
executeYamlRead
yaml_cpp_class, 267
exit
ui.h, 411
Explicit_Eval
UnsteadyAdsorption, 249
UnsteadyReaction, 261
ExplicitFlux
FINCH_DATA, 79
exterior_concentration
MONKFISH_PARAM, 152
exterior_transfer_coeff
MONKFISH_PARAM, 152

F
PJFNK_DATA, 185
f_bias
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
f_bias_old
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
fC_E
FINCH_DATA, 81
fC_I
FINCH_DATA, 81
FEHLBERG
dove.h, 280
FINCH_DATA, 72
beta, 80
CC_E, 81
CC_I, 80
CL_E, 80
CL_I, 80
CR_E, 81
CR_I, 81
callroutine, 84
CheckMass, 79
CN, 79
d, 77
DIC, 78
Dirichlet, 79
discretize, 84
Dn, 83
Dnp1, 83
Do, 78
dt, 77
dt_old, 77
dz, 77
evalprecon, 85
evalres, 85
ExplicitFlux, 79
fC_E, 81
fC_I, 81
fL_E, 81
fL_I, 81
fR_E, 81
fR_I, 81

Fn, 83
Fnp1, 83
gE, 84
gI, 84
Iterative, 80
kIC, 78
kfn, 79
kfnp1, 79
kn, 83
knp1, 83
ko, 79
L, 77
lambda_E, 79
lambda_I, 79
LN, 79
max_iter, 80
ME, 82
MI, 82
NE, 82
NI, 81
nl_method, 80
NormTrack, 80
OE, 81
OI, 81
param_data, 85
picard_dat, 85
pjfnk_dat, 85
pres, 84
RIC, 78
res, 84
resettime, 85
Rn, 83
Rnp1, 83
Ro, 79
s, 77
setbcs, 84
setic, 84
setparams, 84
setpostprocess, 85
setpreprocess, 84
settime, 84
Sn, 83
Snp1, 83
solve, 84
SteadyState, 80
T, 77
t, 77
t_old, 77
tol_abs, 80
tol_rel, 80
total_iter, 80
u_star, 82
uAvg, 78
uAvg_old, 78
uIC, 78
uT_old, 78
ubest, 82
un, 82
unm1, 82
unp1, 82
uo, 78
Update, 79
uT, 78
uz_l_E, 82
uz_l_I, 82
uz_lm1_E, 82
uz_lm1_I, 82
uz_lp1_E, 82
uz_lp1_I, 82
vIC, 78
vn, 83
vnp1, 83
vo, 78
FINCH_Picard
finch.h, 297
FINCH_TESTS
finch.h, 301
fL_E
FINCH_DATA, 81
fL_I
FINCH_DATA, 81
FLORY_HUGGINS
shark.h, 375
FOM
lark.h, 313
fR_E
FINCH_DATA, 81
fR_I
FINCH_DATA, 81
Faraday
shark.h, 373
FE
dove.h, 280
fiber_diameter
DOGFISH_DATA, 56
fiber_length
DOGFISH_DATA, 56
fiber_specific_area
DOGFISH_DATA, 56
File_Output
SHARK_DATA, 217
file_dne
error.h, 292
file_name
yaml_cpp_class, 267
Files
UI_DATA, 240
film_transfer
SCOPSOWL_PARAM_DATA, 206
SKUA_PARAM, 228
film_transfer_coeff
DOGFISH_PARAM, 57
MONKFISH_PARAM, 153
FilmMTCoeff
egret.h, 290
finch

ui.h, 411
finch.h, 294
Cartesian, 297
check_Mass, 297
Cylindrical, 297
default_bcs, 300
default_execution, 299
default_ic, 299
default_params, 300
default_postprocess, 301
default_precon, 300
default_preprocess, 299
default_res, 300
default_reset, 301
default_solve, 299
default_timestep, 299
FINCH_Picard, 297
FINCH_TESTS, 301
finch_coord_type, 297
finch_solve_type, 297
l_direct, 297
LARK_PJFNK, 297
LARK_Picard, 297
lark_picard_step, 298
max, 297
min, 297
minmod, 297
minmod_discretization, 300
nl_picard, 298
ospre_discretization, 300
print2file_dim_header, 299
print2file_newline, 299
print2file_result_new, 299
print2file_result_old, 299
print2file_tab, 299
print2file_time_header, 299
setup_FINCH_DATA, 298
Spherical, 297
uAverage, 297
uTotal, 297
vanAlbada_discretization, 300
finch_coord_type
finch.h, 297
finch_dat
DOGFISH_DATA, 56
MONKFISH_DATA, 151
SCOPSOWL_DATA, 199
SKUA_DATA, 223
finch_solve_type
finch.h, 297
findAllTypes
KeyValueMap, 112
findType
KeyValueMap, 111
ValueTypePair, 264
Fk
BACKTRACK_DATA, 30
flock.h, 301
FloryHuggins
shark.h, 376
FloryHuggins_chemi
shark.h, 377
FloryHuggins_multichemi
shark.h, 377
FloryHuggins_multiligand
shark.h, 376
FloryHuggins_unsteady
shark.h, 376
flow_rate
MassBalance, 122
SHARK_DATA, 216
Fn
FINCH_DATA, 83
Fnp1
FINCH_DATA, 83
Fobj
GSTA_OPT_DATA, 101
fom
lark.h, 316
formation_energy
Molecule, 146
formation_enthalpy
Molecule, 145
formation_entropy
Molecule, 145
Formula
Molecule, 146
forward_rate
UnsteadyReaction, 261
forward_ref_rate
UnsteadyReaction, 262
fun_call
BACKTRACK_DATA, 30
PJFNK_DATA, 184
funeval
PJFNK_DATA, 187
Fv
PJFNK_DATA, 185
Fx
NUM_JAC_DATA, 174
Fxp
NUM_JAC_DATA, 174

GAS
mola.h, 346
GCR_DATA, 85
alpha, 87
bestres, 88
bestx, 88
beta, 87
breakdown, 87
c, 88
c_temp, 88
iter_inner, 87
iter_outer, 87
maxit, 87
Output, 88

r, 88
relres, 88
relres_base, 88
res, 88
restart, 87
tol_abs, 87
tol_rel, 87
total_iter, 87
transpose_dat, 89
u, 88
u_temp, 88
x, 88
GCR_Output
GMRESR_DATA, 93
GCR
lark.h, 313
GMRES_Output
GMRESR_DATA, 93
GMRESLP_DATA, 89
arnoldi_dat, 91
bestres, 90
bestx, 91
iter, 90
maxit, 90
Output, 91
r, 91
relres, 90
relres_base, 90
res, 90
restart, 90
steps, 90
tol_abs, 90
tol_rel, 90
x, 91
GMRESLP
lark.h, 313
GMRESR_DATA, 91
arg, 93
GCR_Output, 93
GMRES_Output, 93
gcr_abs_tol, 93
gcr_dat, 93
gcr_maxit, 92
gcr_rel_tol, 93
gcr_restart, 92
gmres_dat, 93
gmres_maxit, 92
gmres_restart, 92
gmres_tol, 93
iter_inner, 93
iter_outer, 93
matvec, 94
matvec_data, 94
N, 93
term_precon, 94
terminal_precon, 94
total_iter, 93
GMRESRP_DATA, 94
bestres, 96
bestx, 96
e0, 97
e0_bar, 97
H, 97
H_bar, 97
iter_inner, 96
iter_outer, 96
iter_total, 96
maxit, 95
Output, 96
r, 97
relres, 96
relres_base, 96
res, 96
restart, 95
sum, 97
tol_abs, 96
tol_rel, 96
v, 97
Vk, 97
w, 97
x, 96
y, 97
Zk, 97
GMRESRP
lark.h, 313
GMRESR
lark.h, 313
GPAST_DATA, 98
gama_inf, 99
He, 98
PIo, 99
po, 99
poi, 99
present, 99
q, 98
qo, 99
x, 98
y, 98
GSTA_DATA, 99
dHo, 100
dSo, 100
m, 100
qmax, 100
GSTA_OPT_DATA, 100
all_pars, 102
best_par, 102
Fobj, 101
iso, 101
Kno, 102
n_par, 101
norms, 102
opt_qmax, 102
P, 102
q, 102
qmax, 101
total_eval, 101

gama
mSPD_DATA, 154
gama_inf
GPAST_DATA, 99
gas_dat
SCOPSOWL_DATA, 199
SKUA_DATA, 223
gas_temperature
MIXED_GAS, 138
SCOPSOWL_DATA, 198
gas_velocity
SCOPSOWL_DATA, 197
SKUA_DATA, 222
gcr
lark.h, 320
gcr_abs_tol
GMRESR_DATA, 93
gcr_dat
GMRESR_DATA, 93
PJFNK_DATA, 186
gcr_maxit
GMRESR_DATA, 92
gcr_rel_tol
GMRESR_DATA, 93
gcr_restart
GMRESR_DATA, 92
gE
FINCH_DATA, 84
generic_error
error.h, 292
Get_ActivationEnergy
UnsteadyReaction, 260
Get_Affinity
UnsteadyReaction, 260
Get_Area
MassBalance, 121
Get_Delta
MassBalance, 120
Get_Energy
Reaction, 193
UnsteadyReaction, 259
Get_Enthalpy
Reaction, 193
UnsteadyReaction, 259
Get_Entropy
Reaction, 193
UnsteadyReaction, 259
Get_Equilibrium
Reaction, 193
UnsteadyReaction, 259
Get_FlowRate
MassBalance, 121
Get_Forward
UnsteadyReaction, 259
Get_ForwardRef
UnsteadyReaction, 260
Get_InitialConcentration
MassBalance, 121
Get_InitialValue
UnsteadyReaction, 259
Get_InletConcentration
MassBalance, 121
Get_MaximumValue
UnsteadyReaction, 259
Get_Name
MassBalance, 121
Get_Reverse
UnsteadyReaction, 260
Get_ReverseRef
UnsteadyReaction, 260
Get_Species_Index
UnsteadyReaction, 259
Get_Stoichiometric
Reaction, 192
UnsteadyReaction, 259
Get_TimeStep
MassBalance, 121
UnsteadyReaction, 260
Get_TotalConcentration
MassBalance, 120
Get_Type
MassBalance, 120
Get_Volume
MassBalance, 121
get_index
MasterSpeciesList, 126
get_species
MasterSpeciesList, 126
getActivity
AdsorptionReaction, 18
ChemisorptionReaction, 45
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 250
getActivityEnum
AdsorptionReaction, 19
ChemisorptionReaction, 46
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 251
getAdsorbIndex
AdsorptionReaction, 19
ChemisorptionReaction, 46
UnsteadyAdsorption, 251
getAdsorbentName
AdsorptionReaction, 19
ChemisorptionReaction, 46
MultiligandAdsorption, 162
MultiligandChemisorption, 172
UnsteadyAdsorption, 251
getAdsorptionObject
MultiligandAdsorption, 161
getAlias
Document, 52
Header, 107
SubHeader, 232

getAnchoredDoc
YamlWrapper, 271
getAnchoredHeader
Document, 53
getAnchoredSub
Header, 108
getAqueousIndex
AdsorptionReaction, 19
UnsteadyAdsorption, 251
getAreaFactor
AdsorptionReaction, 18
ChemisorptionReaction, 45
UnsteadyAdsorption, 250
getBool
KeyValueMap, 112
ValueTypePair, 265
getBulkDensity
AdsorptionReaction, 18
ChemisorptionReaction, 46
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 250
getChargeDensity
AdsorptionReaction, 18
ChemisorptionReaction, 46
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 250
getChemisorptionObject
MultiligandChemisorption, 170
getCurrentTime
Dove, 67
getCurrentU
Dove, 67
getDataMap
Document, 51
Header, 106
getDelta
ChemisorptionReaction, 45
getDocFromHeadAlias
YamlWrapper, 271
getDocFromSubAlias
YamlWrapper, 271
getDocMap
YamlWrapper, 270
getDocument
YamlWrapper, 270
getDouble
KeyValueMap, 112
ValueTypePair, 265
getElectricPotential
MultiligandAdsorption, 162
MultiligandChemisorption, 171
getEndTime
Dove, 67
getHeadFromSubAlias
Document, 53
getHeadMap
Document, 51
getHeader
Document, 51
getInt
KeyValueMap, 112
ValueTypePair, 265
getIonicStrength
AdsorptionReaction, 18
ChemisorptionReaction, 46
MultiligandAdsorption, 162
MultiligandChemisorption, 171
UnsteadyAdsorption, 251
getJacobiMap
Dove, 68
getLigandIndex
ChemisorptionReaction, 46
getLigandName
MultiligandAdsorption, 162
MultiligandChemisorption, 171
getMap
KeyValueMap, 110
SubHeader, 231
getMaxTimeStep
Dove, 68
getMinTimeStep
Dove, 68
getMolarFactor
AdsorptionReaction, 17
UnsteadyAdsorption, 250
getName
Document, 52
Header, 107
SubHeader, 232
getNewU
Dove, 67
getNonlinearRelativeRes
Dove, 68
getNonlinearResidual
Dove, 68
getNumFunc
Dove, 67
getNumberLigands
MultiligandAdsorption, 161
MultiligandChemisorption, 171
getNumberRxns
AdsorptionReaction, 18
ChemisorptionReaction, 46
UnsteadyAdsorption, 251
getOldActivity
UnsteadyAdsorption, 250
getOldTime
Dove, 67
getOlderTime
Dove, 67
getOldU
Dove, 67
getPair
KeyValueMap, 112

ValueTypePair, 265
getReaction
AdsorptionReaction, 17
ChemisorptionReaction, 45
UnsteadyAdsorption, 249
getSpecificArea
AdsorptionReaction, 18
ChemisorptionReaction, 45
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 250
getSpecificMolality
AdsorptionReaction, 18
ChemisorptionReaction, 46
UnsteadyAdsorption, 250
getState
Document, 52
Header, 107
SubHeader, 232
getString
KeyValueMap, 112
ValueTypePair, 265
getSubHeader
Header, 106
getSubMap
Header, 106
getSurfaceCharge
AdsorptionReaction, 18
UnsteadyAdsorption, 250
getTimeStep
Dove, 67
getTimeStepOld
Dove, 67
getTotalMass
AdsorptionReaction, 18
ChemisorptionReaction, 46
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 250
getTotalVolume
AdsorptionReaction, 18
ChemisorptionReaction, 46
MultiligandAdsorption, 161
MultiligandChemisorption, 171
UnsteadyAdsorption, 250
getType
KeyValueMap, 112
ValueTypePair, 265
getUserData
Dove, 67
getValue
KeyValueMap, 112
ValueTypePair, 265
getVolumeFactor
AdsorptionReaction, 18
ChemisorptionReaction, 45
UnsteadyAdsorption, 250
getYamlWrapper
yaml_cpp_class, 267
gI
FINCH_DATA, 84
gmres_dat
GMRESR_DATA, 93
gmres_in
KMS_DATA, 115
gmres_maxit
GMRESR_DATA, 92
gmres_out
KMS_DATA, 115
gmres_restart
GMRESR_DATA, 92
gmres_tol
GMRESR_DATA, 93
gmresLeftPreconditioned
lark.h, 314
gmresRightPreconditioned
lark.h, 317
gmreslp_dat
PJFNK_DATA, 186
gmresr
lark.h, 321
gmresr_dat
PJFNK_DATA, 186
gmresrPreconditioner
lark.h, 321
gmresrp_dat
PJFNK_DATA, 186
gpast_dat
MAGPIE_DATA, 116
grad_mSPD
magpie.h, 333
Grav_R
Trajectory.h, 407
Grav_T
Trajectory.h, 407
gsta_dat
MAGPIE_DATA, 116
gsta_opt
ui.h, 411
gsta_opt.h, 302
avgPar, 305
avgValue, 305
eduGuess, 306
eval_GSTA, 308
gsta_optimize, 308
gstaFunc, 306
gstaObjFunc, 308
isSmooth, 306
minIndex, 305
minValue, 304
Na, 304
orderMag, 304
orthoLinReg, 306
Po, 304
R, 304
rSq, 305

roundIt, 304
twoFifths, 304
weightedAvg, 305
gsta_optimize
gsta_opt.h, 308
gstaFunc
gsta_opt.h, 306
gstaObjFunc
gsta_opt.h, 308

H
GMRESRP_DATA, 97
TRAJECTORY_DATA, 238
H0
TRAJECTORY_DATA, 237
H_bar
GMRESRP_DATA, 97
HELP
ui.h, 411
Hamaker
TRAJECTORY_DATA, 237
hasConverged
Dove, 68
HaveEnergy
Molecule, 144
HaveEquil
Reaction, 194
haveEquilibrium
Reaction, 192
UnsteadyReaction, 258
HaveForRef
UnsteadyReaction, 262
HaveForward
UnsteadyReaction, 262
haveForward
UnsteadyReaction, 259
haveForwardRef
UnsteadyReaction, 258
HaveHS
Molecule, 144
Reaction, 194
haveHS
Molecule, 146
haveMinMax
MONKFISH_DATA, 149
haveRate
UnsteadyReaction, 258
HaveRevRef
UnsteadyReaction, 262
HaveReverse
UnsteadyReaction, 262
haveReverse
UnsteadyReaction, 259
haveReverseRef
UnsteadyReaction, 259
HaveG
Reaction, 194
haveG
Molecule, 146
He
GPAST_DATA, 98
magpie.h, 331
Head_Map
Document, 53
Header, 102
∼Header, 105
addPair, 106
addSubKey, 107
begin, 106
changeKey, 106
clear, 106
copyAnchor2Alias, 107
DisplayContents, 107
end, 106
getAlias, 107
getAnchoredSub, 108
getDataMap, 106
getName, 107
getState, 107
getSubHeader, 106
getSubMap, 106
Header, 105
isAlias, 107
isAnchor, 108
operator(), 105
operator=, 105
operator[], 105
resetKeys, 106
setAlias, 107
setName, 107
setNameAliasPair, 107
setState, 107
size, 107
Sub_Map, 108
header_state
yaml_wrapper.h, 420
help
ui.h, 412
Heterogeneous
SCOPSOWL_DATA, 197
Hkp1
ARNOLDI_DATA, 23
hp1
ARNOLDI_DATA, 22

I
SYSTEM_DATA, 234
IDEAL_ADS
shark.h, 375
IDEAL
shark.h, 375
IMPLICIT
dove.h, 279
INT
yaml_wrapper.h, 420
Ideal
SYSTEM_DATA, 235
ideal_solution

shark.h, 379
li
OPTRANS_DATA, 175
ln_P_Velocity
Trajectory.h, 407
ln_PVel_Rad
Trajectory.h, 407
ln_PVel_Theta
Trajectory.h, 407
IncludeSurfCharge
AdsorptionReaction, 21
MultiligandAdsorption, 163
MultiligandChemisorption, 173
includeSurfaceCharge
AdsorptionReaction, 19
ChemisorptionReaction, 46
MultiligandAdsorption, 162
MultiligandChemisorption, 171
UnsteadyAdsorption, 251
indexing_error
error.h, 292
initial_error
error.h, 293
initial_guess_SCOPSOWL
scopsowl_opt.h, 365
initial_guess_SKUA
skua_opt.h, 402
initial_sorption
DOGFISH_PARAM, 58
MONKFISH_PARAM, 152
initial_value
UnsteadyReaction, 261
InitialConcentration
MassBalance, 123
initialGuess_mSPD
magpie.h, 334
Initialize_Object
AdsorptionReaction, 12
ChemisorptionReaction, 41
MassBalance, 119
MultiligandAdsorption, 157
MultiligandChemisorption, 167
Reaction, 191
UnsteadyAdsorption, 244
UnsteadyReaction, 255
initialize_data
egret.h, 290
InletConcentration
MassBalance, 123
inner_iter
KMS_DATA, 114
inner_product
Matrix, 131
inner_reltol
KMS_DATA, 114
input
ui.h, 413
input_file
yaml_cpp_class, 267
input_files
UI_DATA, 240
int_sub
Dove, 71
int_type
Dove, 70
IntegralAvg
Matrix, 134
IntegralTotal
Matrix, 134
integrate_subtype
dove.h, 279
integrate_type
dove.h, 279
interparticle_diffusion
MONKFISH_PARAM, 152
intraparticle_diffusion
DOGFISH_PARAM, 57
MONKFISH_PARAM, 153
invalid_atom
error.h, 293
invalid_boolean
error.h, 293
invalid_components
error.h, 292
invalid_console_input
error.h, 293
invalid_electron
error.h, 293
invalid_fraction
error.h, 293
invalid_gas_sum
error.h, 293
invalid_input
ui.h, 414
invalid_molefraction
error.h, 293
invalid_neutron
error.h, 293
invalid_norm
error.h, 293
invalid_proton
error.h, 293
invalid_size
error.h, 293
invalid_solid_sum
error.h, 293
invalid_species
error.h, 293
invalid_type
error.h, 293
invalid_valence
error.h, 293
inverse
Matrix, 132
ionic_strength
AdsorptionReaction, 21

MultiligandAdsorption, 163
MultiligandChemisorption, 173
SHARK_DATA, 216
isAlias
Document, 53
Header, 107
SubHeader, 232
isAnchor
Document, 53
Header, 108
SubHeader, 232
isAreaBasis
AdsorptionReaction, 19
UnsteadyAdsorption, 251
isEven
yaml_wrapper.h, 421
isRegistered
Molecule, 144
isSmooth
gsta_opt.h, 306
isSteadyState
MassBalance, 121
isZeroInitialSolids
MassBalance, 121
iso
GSTA_OPT_DATA, 101
iter
ARNOLDI_DATA, 22
BiCGSTAB_DATA, 32
CGS_DATA, 36
GMRESLP_DATA, 90
PCG_DATA, 176
PICARD_DATA, 180
iter_inner
GCR_DATA, 87
GMRESR_DATA, 93
GMRESRP_DATA, 96
iter_outer
GCR_DATA, 87
GMRESR_DATA, 93
GMRESRP_DATA, 96
iter_total
GMRESRP_DATA, 96
Iterative
FINCH_DATA, 80

J
SYSTEM_DATA, 234
JACOBI
dove.h, 281
jacvec
lark.h, 324

K
SYSTEM_DATA, 234
k
ARNOLDI_DATA, 22
TRAJECTORY_DATA, 237
kIC
FINCH_DATA, 78
KMS_DATA, 113
gmres_in, 115
gmres_out, 115
inner_iter, 114
inner_reltol, 114
level, 114
matvec, 115
matvec_data, 115
max_level, 114
maxit, 114
outer_abstol, 114
outer_iter, 114
outer_reltol, 114
Output_inner, 115
Output_outer, 115
restart, 114
term_precon, 115
terminal_precon, 115
total_iter, 114
KMS
lark.h, 313
kB
magpie.h, 331
shark.h, 373
Key_Value
KeyValueMap, 112
key_not_found
error.h, 293
KeyValueMap, 108
~KeyValueMap, 110
addKey, 111
addPair, 111
assertType, 111
begin, 111
clear, 111
DisplayMap, 112
editValue4Key, 111
end, 110, 111
findAllTypes, 112
findType, 111
getBool, 112
getDouble, 112
getInt, 112
getMap, 110
getPair, 112
getString, 112
getType, 112
getValue, 112
Key_Value, 112
KeyValueMap, 110
operator=, 110
operator[], 110
size, 112
kfn
FINCH_DATA, 79
kfnp1
FINCH_DATA, 79

kinematic_viscosity
MIXED_GAS, 138
kms_dat
PJFNK_DATA, 186
kmsPreconditioner
lark.h, 322
kn
FINCH_DATA, 83
Kno
GSTA_OPT_DATA, 102
knp1
FINCH_DATA, 83
ko
FINCH_DATA, 79
krylov_method
lark.h, 313
krylovMultiSpace
lark.h, 322

L
FINCH_DATA, 77
TRAJECTORY_DATA, 237
L_Output
PJFNK_DATA, 185
l_direct
finch.h, 297
l_iter
PJFNK_DATA, 183
l_maxit
PJFNK_DATA, 184
l_restart
PJFNK_DATA, 184
L_wire
TRAJECTORY_DATA, 237
LARK_PJFNK
finch.h, 297
LARK_Picard
finch.h, 297
LARK_TESTS
lark.h, 326
LGS
dove.h, 281
LIQUID
mola.h, 345
LOCATION
Trajectory.h, 408
ladshawSolve
Matrix, 133
lambda_E
FINCH_DATA, 79
lambda_I
FINCH_DATA, 79
lambdaMin
BACKTRACK_DATA, 30
lark
ui.h, 411
lark.h, 309
arnoldi, 314
backtrackLineSearch, 324
BiCGSTAB, 313
bicgstab, 318
CGS, 313
cgs, 319
FOM, 313
fom, 316
GCR, 313
GMRESLP, 313
GMRESRP, 313
GMRESR, 313
gcr, 320
gmresLeftPreconditioned, 314
gmresRightPreconditioned, 317
gmresr, 321
gmresrPreconditioner, 321
jacvec, 324
KMS, 313
kmsPreconditioner, 322
krylov_method, 313
krylovMultiSpace, 322
LARK_TESTS, 326
MIN_TOL, 313
NumericalJacobian, 326
operatorTranspose, 319
PCG, 313
pcg, 317
picard, 323
pjfnk, 325
QRsolve, 323
QR, 313
update_arnoldi_solution, 313
lark_picard_step
finch.h, 298
LengthFactor
shark.h, 373
level
KMS_DATA, 114
MONKFISH_DATA, 149
SCOPSOWL_DATA, 197
ligand_index
ChemisorptionReaction, 47
ligand_obj
MultiligandAdsorption, 163
MultiligandChemisorption, 173
lin_precon
SHARK_DATA, 218
lin_tol_abs
PJFNK_DATA, 184
lin_tol_rel
PJFNK_DATA, 184
LineSearch
PJFNK_DATA, 185
Linear
Dove, 71
linear_solver
PJFNK_DATA, 184
linearsolve_choice
shark.h, 380

linesearch_choice
shark.h, 380
linesearch_type
dove.h, 280
linmax
Dove, 71
List
AdsorptionReaction, 19
MassBalance, 122
MultiligandAdsorption, 162
MultiligandChemisorption, 172
Reaction, 193
list_size
MasterSpeciesList, 126
LN
FINCH_DATA, 79
lnKo
magpie.h, 331
lnact_mSPD
magpie.h, 333
LocalMin
SHARK_DATA, 217
lowerHessenberg2Triangular
Matrix, 135
lowerHessenbergSolve
Matrix, 135
lowerTriangularSolve
Matrix, 135

M
TRAJECTORY_DATA, 238
m
GSTA_DATA, 100
M_PI
macaw.h, 328
mola.h, 345
m_rand
TRAJECTORY_DATA, 238
MACAW_TESTS
macaw.h, 328
MAGPIE_DATA, 115
gpast_dat, 116
gsta_dat, 116
mspd_dat, 116
sys_dat, 116
MAGPIE_SCENARIOS
magpie.h, 336
MAGPIE
magpie.h, 336
mError
error.h, 292
MIN_TOL
lark.h, 313
MIXED_GAS, 136
binary_diffusion, 138
char_length, 138
CheckMolefractions, 137
gas_temperature, 138
kinematic_viscosity, 138
molefraction, 138
N, 137
Reynolds, 138
species_dat, 138
total_density, 138
total_dyn_vis, 138
total_molecular_weight, 138
total_pressure, 137
total_specific_heat, 138
velocity, 138
MOLA_TESTS
mola.h, 346
MONKFISH_DATA, 146
avg_fiber_density, 150
DirichletBC, 149
dog_dat, 151
domain_diameter, 150
end_time, 149
eval_Cex, 151
eval_Dex, 150
eval_Ret, 150
eval_ads, 150
eval_eps, 150
eval_kf, 151
eval_rho, 150
finch_dat, 151
haveMinMax, 149
level, 149
max_fiber_density, 150
max_porosity, 150
min_fiber_density, 150
min_porosity, 150
MultiScale, 149
NonLinear, 149
NumComp, 149
Output, 150
param_dat, 151
Print2Console, 148
Print2File, 148
single_fiber_density, 149
t_counter, 149
t_print, 149
time, 148
time_old, 148
total_sorption, 149
total_sorption_old, 149
total_steps, 148
user_data, 151
MONKFISH_PARAM, 151
avg_sorption, 153
avg_sorption_old, 153
exterior_concentration, 152
exterior_transfer_coeff, 152
film_transfer_coeff, 153
initial_sorption, 152
interparticle_diffusion, 152
intraparticle_diffusion, 153
sorbed_molefraction, 152

sorption_bc, 153
species, 153
MONKFISH_TESTS
monkfish.h, 350
mSPD_DATA, 153
eMax, 154
eta, 154
gama, 154
s, 154
v, 154
macaw
ui.h, 411
macaw.h, 326
M_PI, 328
MACAW_TESTS, 328
Magnetic_R
Trajectory.h, 407
Magnetic_T
Trajectory.h, 407
magpie
ui.h, 411
magpie.h, 328
A, 331
DBL_EPSILON, 330
dq_dp, 332
eMax, 333
eval_GPAST, 335
eval_eta, 335
eval_po, 335
eval_po_PI, 334
eval_po_qo, 334
grad_mSPD, 333
He, 331
initialGuess_mSPD, 334
kB, 331
lnKo, 331
lnact_mSPD, 333
MAGPIE_SCENARIOS, 336
MAGPIE, 336
Na, 331
PI, 332
Po, 331
q_p, 332
qo, 331
Qst, 332
qT, 333
R, 331
shapeFactor, 331
V, 331
Z, 330
magpie_dat
SCOPSOWL_DATA, 199
SKUA_DATA, 222
magpie_reverse_error
error.h, 292
MassBalance, 116
∼MassBalance, 119
Delta, 122
Display_Info, 119
dt, 122
Eval_IC_Residual, 122
Eval_Residual, 121
flow_rate, 122
Get_Area, 121
Get_Delta, 120
Get_FlowRate, 121
Get_InitialConcentration, 121
Get_InletConcentration, 121
Get_Name, 121
Get_TimeStep, 121
Get_TotalConcentration, 120
Get_Type, 120
Get_Volume, 121
InitialConcentration, 123
Initialize_Object, 119
InletConcentration, 123
isSteadyState, 121
isZeroInitialSolids, 121
List, 122
MassBalance, 119
Name, 123
Set_Area, 120
Set_Delta, 119
Set_FlowRate, 120
Set_InitialConcentration, 120
Set_InletConcentration, 120
Set_Name, 120
Set_SteadyState, 120
Set_TimeStep, 120
Set_TotalConcentration, 119
Set_Type, 119
Set_Volume, 119
Set_ZeroInitialSolids, 120
SteadyState, 123
Sum_Delta, 120
TotalConcentration, 122
Type, 122
volume, 122
xsec_area, 122
ZeroInitialSolids, 123
MassBalanceList
SHARK_DATA, 211
MasterList
SHARK_DATA, 211
MasterSpeciesList, 123
∼MasterSpeciesList, 124
alkalinity, 126
charge, 126
DisplayAll, 125
DisplayConcentrations, 125
DisplayInfo, 125
Eval_ChargeResidual, 126
get_index, 126
get_species, 126
list_size, 126
MasterSpeciesList, 124, 125

operator=, 125
residual_alkalinity, 127
set_alkalinity, 125
set_list_size, 125
set_species, 125
size, 127
species, 127
speciesName, 126
Matrix
~Matrix, 130
adjoint, 132
cofactor, 131
columnExtend, 136
columnExtract, 135
columnProjection, 134
columnReplace, 135
columnShrink, 136
columnVectorFill, 134
columns, 131
ConstantICFill, 133
Data, 136
determinate, 131
diagonalSolve, 134
dirichletBCFill, 134
Display, 132
edit, 131
inner_product, 131
IntegralAvg, 134
IntegralTotal, 134
inverse, 132
ladshawSolve, 133
lowerHessenberg2Triangular, 135
lowerHessenbergSolve, 135
lowerTriangularSolve, 135
Matrix, 130
naturalLaplacian3D, 133
norm, 131
num_cols, 136
num_rows, 136
operator*, 132
operator(), 130
operator+, 131, 132
operator-, 131, 132
operator/, 132
operator=, 130
outer_product, 132
qrSolve, 135
rowExtend, 136
rowExtract, 135
rowReplace, 136
rowShrink, 136
rows, 131
set_size, 131
SolnTransform, 133
sphericalAvg, 133
sphericalBCFill, 133
sum, 131
transpose, 132
transpose_multiply, 132
tridiagonalFill, 133
tridiagonalSolve, 132
tridiagonalVectorFill, 134
upperHessenberg2Triangular, 135
upperHessenbergSolve, 135
upperTriangularSolve, 135
zeros, 131
Matrix< T >, 127
matrix_too_small
error.h, 293
matvec
GMRESR_DATA, 94
KMS_DATA, 115
matvec_data
GMRESR_DATA, 94
KMS_DATA, 115
matvec_mis_match
error.h, 293
max
finch.h, 297
UI_DATA, 240
max_bias
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
max_fiber_density
MONKFISH_DATA, 150
max_guess_iter
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
max_iter
FINCH_DATA, 80
max_level
KMS_DATA, 114
max_norm
SYSTEM_DATA, 235
max_porosity
MONKFISH_DATA, 150
max_value
UnsteadyReaction, 261
maxit
BiCGSTAB_DATA, 32
CGS_DATA, 36
GCR_DATA, 87
GMRESLP_DATA, 90
GMRESRP_DATA, 95
KMS_DATA, 114
PCG_DATA, 176
PICARD_DATA, 180
ME
FINCH_DATA, 82
MI
FINCH_DATA, 82
min
finch.h, 297
min_bias
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226

min_fiber_density
MONKFISH_DATA, 150
min_porosity
MONKFISH_DATA, 150
minIndex
gsta_opt.h, 305
minValue
gsta_opt.h, 304
minmod
finch.h, 297
minmod_discretization
finch.h, 300
missing_information
error.h, 293
MissingArg
UI_DATA, 240
modifyDeltas
AdsorptionReaction, 12
MultiligandAdsorption, 158
UnsteadyAdsorption, 244
modifyMBEdeltas
ChemisorptionReaction, 41
MultiligandChemisorption, 167
mola
ui.h, 411
mola.h, 338
ADSORBED, 346
AQUEOUS, 346
GAS, 346
LIQUID, 345
M_PI, 345
MOLA_TESTS, 346
OTHER, 346
PLASMA, 346
SOLID, 345
SphereArea, 345
SphereVolume, 345
valid_phase, 345
molar_area
Molecule, 145
molar_factor
AdsorptionReaction, 20
molar_volume
Molecule, 145
molar_weight
Molecule, 145
MolarArea
Molecule, 144
MolarVolume
Molecule, 144
MolarWeight
Molecule, 144
molecular_diffusion
PURE_GAS, 188
molecular_weight
PURE_GAS, 188
MolecularFormula
Molecule, 145
Molecule, 139
∼Molecule, 141
atoms, 146
calculateAvgOxiState, 143
calculateMolarArea, 143
calculateMolarVolume, 143
calculateMolarWeight, 143
Charge, 144
charge, 145
DisplayInfo, 145
editAllOxidationStates, 143
editCharge, 143
editEnergy, 144
editEnthalpy, 143
editEntropy, 143
editHS, 143
editOneOxidationState, 143
Energy, 145
Enthalpy, 144
Entropy, 144
formation_energy, 146
formation_enthalpy, 145
formation_entropy, 145
Formula, 146
HaveEnergy, 144
HaveHS, 144
haveHS, 146
haveG, 146
isRegistered, 144
molar_area, 145
molar_volume, 145
molar_weight, 145
MolarArea, 144
MolarVolume, 144
MolarWeight, 144
MolecularFormula, 145
Molecule, 141
MoleculeName, 145
MoleculePhase, 145
MoleculePhaseID, 145
Name, 146
Phase, 146
PhaseID, 146
Register, 142
registered, 146
removeAllAtoms, 144
removeOneAtom, 144
setFormula, 142
setMolarArea, 143
setMolarVolume, 143
setMolarWeigth, 143
MoleculeName
Molecule, 145
MoleculePhase
Molecule, 145
MoleculePhaseID
Molecule, 145
molefraction

MIXED_GAS, 138
molefractionCheck
skua.h, 396
monkfish
ui.h, 411
monkfish.h, 346
default_density, 348
default_exterior_concentration, 349
default_film_transfer, 349
default_interparticle_diffusion, 348
default_monk_adsorption, 348
default_monk_equilibrium, 348
default_monkfish_retardation, 349
default_porosity, 347
MONKFISH_TESTS, 350
setup_MONKFISH_DATA, 349
mp
TRAJECTORY_DATA, 238
Ms
TRAJECTORY_DATA, 237
mspd_dat
MAGPIE_DATA, 116
Mu
egret.h, 290
mu_0
TRAJECTORY_DATA, 237
MultiAdsList
SHARK_DATA, 212
MultiChemList
SHARK_DATA, 212
MultiScale
MONKFISH_DATA, 149
MultiligandAdsorption, 154
~MultiligandAdsorption, 157
act_fun, 163
activities, 163
activity_data, 163
adsorbent_name, 162
calculateAreaFactors, 159
calculateElecticPotential, 160
calculateEquilibria, 159
calculateEquilibriumCorrection, 160
callSurfaceActivity, 160
charge_density, 163
checkAqueousIndices, 158
electric_potential, 163
Eval_Residual, 160
getActivity, 161
getActivityEnum, 161
getAdsorbentName, 162
getAdsorptionObject, 161
getBulkDensity, 161
getChargeDensity, 161
getElectricPotential, 162
getIonicStrength, 162
getLigandName, 162
getNumberLigands, 161
getSpecificArea, 161
getTotalMass, 161
getTotalVolume, 161
IncludeSurfCharge, 163
includeSurfaceCharge, 162
Initialize_Object, 157
ionic_strength, 163
ligand_obj, 163
List, 162
modifyDeltas, 158
MultiligandAdsorption, 157
num_ligands, 162
setActivityEnum, 158
setActivityModelInfo, 158
setAdsorbIndices, 158
setAdsorbentName, 159
setAqueousIndexAuto, 158
setAreaFactor, 159
setChargeDensity, 160
setElectricPotential, 159
setIonicStrength, 160
setLigandName, 159
setMolarFactor, 158
setSpecificArea, 159
setSpecificMolality, 159
setSurfaceCharge, 159
setSurfaceChargeBool, 159
setTotalMass, 159
setTotalVolume, 159
setVolumeFactor, 159
specific_area, 163
surface_activity, 162
total_mass, 163
total_volume, 163
MultiligandChemisorption, 164
~MultiligandChemisorption, 167
act_fun, 172
activities, 172
activity_data, 172
adsorbent_name, 172
calculateAreaFactors, 169
calculateElecticPotential, 169
calculateEquilibria, 169
calculateEquilibriumCorrection, 170
callSurfaceActivity, 169
charge_density, 173
Display_Info, 167
electric_potential, 173
Eval_RxnResidual, 170
Eval_SiteBalanceResidual, 170
getActivity, 171
getActivityEnum, 171
getAdsorbentName, 172
getBulkDensity, 171
getChargeDensity, 171
getChemisorptionObject, 170
getElectricPotential, 171
getIonicStrength, 171
getLigandName, 171

getNumberLigands, 171
getSpecificArea, 171
getTotalMass, 171
getTotalVolume, 171
IncludeSurfCharge, 173
includeSurfaceCharge, 171
Initialize_Object, 167
ionic_strength, 173
ligand_obj, 173
List, 172
modifyMBEdeltas, 167
MultiligandChemisorption, 167
num_ligands, 172
setActivityEnum, 168
setActivityModelInfo, 168
setAdsorbIndices, 167
setAdsorbentName, 168
setAreaFactor, 168
setChargeDensity, 169
setDeltas, 168
setElectricPotential, 169
setIonicStrength, 169
setLigandIndices, 167
setLigandName, 168
setSpecificArea, 168
setSpecificMolality, 168
setSurfaceChargeBool, 169
setTotalMass, 168
setTotalVolume, 169
setVolumeFactor, 168
specific_area, 172
surface_activity, 172
total_mass, 173
total_volume, 173

N
GMRESR_DATA, 93
MIXED_GAS, 137
SYSTEM_DATA, 234
n_par
GSTA_OPT_DATA, 101
n_rand
TRAJECTORY_DATA, 238
NL_Output
PJFNK_DATA, 185
NO_LS
dove.h, 281
NONE
yaml_wrapper.h, 421
NUM_JAC_DATA, 173
dxj, 174
eps, 174
Fx, 174
Fxp, 174
Na
gsta_opt.h, 304
magpie.h, 331
shark.h, 373
Name
Atom, 29
MassBalance, 123
Molecule, 146
name
SubHeader, 232
naturalLaplacian3D
Matrix, 133
NaturalState
Atom, 29
NE
FINCH_DATA, 82
negative_mass
error.h, 293
negative_time
error.h, 293
Neutrons
Atom, 27
neutrons
Atom, 28
newton_dat
Dove, 72
Newton_data
SHARK_DATA, 219
NI
FINCH_DATA, 81
nl_bestres
PJFNK_DATA, 185
nl_iter
PJFNK_DATA, 183
nl_maxit
PJFNK_DATA, 184
nl_method
FINCH_DATA, 80
nl_picard
finch.h, 298
nl_relres
PJFNK_DATA, 184
nl_res
PJFNK_DATA, 184
nl_res_base
PJFNK_DATA, 184
nl_tol_abs
PJFNK_DATA, 184
nl_tol_rel
PJFNK_DATA, 184
no_diffusion
error.h, 293
non_real_edge
error.h, 293
non_square_matrix
error.h, 293
NonLinear
DOGFISH_DATA, 55
MONKFISH_DATA, 149
SCOPSOWL_DATA, 198
SKUA_DATA, 222
Norm
SHARK_DATA, 216

norm
Matrix, 131
normFkp1
BACKTRACK_DATA, 30
NormTrack
FINCH_DATA, 80
norms
GSTA_OPT_DATA, 102
not_a_token
error.h, 293
Nu
egret.h, 290
nullptr_error
error.h, 293
nullptr_func
error.h, 293
num_cols
Matrix, 136
num_curves
SCOPSOWL_OPT_DATA, 201
SKUA_OPT_DATA, 225
num_func
Dove, 71
num_ligands
MultiligandAdsorption, 162
MultiligandChemisorption, 172
num_mbe
SHARK_DATA, 212
num_multi_ssao
SHARK_DATA, 213
num_multi_ssar
SHARK_DATA, 213
num_multi_sschem
SHARK_DATA, 213
num_multichem_rxns
SHARK_DATA, 213
num_other
SHARK_DATA, 214
num_params
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
num_rows
Matrix, 136
num_rxns
AdsorptionReaction, 21
num_ssao
SHARK_DATA, 213
num_ssar
SHARK_DATA, 213
num_sschem
SHARK_DATA, 213
num_sschem_rxns
SHARK_DATA, 213
num_ssr
SHARK_DATA, 212
num_usao
SHARK_DATA, 213
num_usar
SHARK_DATA, 213
num_usr
SHARK_DATA, 213
NumComp
DOGFISH_DATA, 55
MONKFISH_DATA, 149
Number_Generator
Trajectory.h, 408
number_elements
PeriodicTable, 179
number_files
ui.h, 413
NumericalJacobian
lark.h, 326
numvar
SHARK_DATA, 212

OPTRANS_DATA, 174
Ai, 175
Ii, 175
OTHER
mola.h, 346
OE
FINCH_DATA, 81
OI
FINCH_DATA, 81
omega
BiCGSTAB_DATA, 32
omega_old
BiCGSTAB_DATA, 33
operator∗
Matrix, 132
operator()
Document, 50
Header, 105
Matrix, 130
YamlWrapper, 270
operator+
Matrix, 131, 132
operator-
Matrix, 131, 132
operator/
Matrix, 132
operator=
Document, 50
Header, 105
KeyValueMap, 110
MasterSpeciesList, 125
Matrix, 130
SubHeader, 231
ValueTypePair, 264
YamlWrapper, 270
operatorTranspose
lark.h, 319
operator[]
Document, 50
Header, 105
KeyValueMap, 110
SubHeader, 231

opt_no_support
error.h, 293
opt_qmax
GSTA_OPT_DATA, 102
Optimize
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
option
UI_DATA, 240
orderMag
gsta_opt.h, 304
ortho_check_fail
error.h, 293
orthoLinReg
gsta_opt.h, 306
ospre_discretization
finch.h, 300
other_data
SHARK_DATA, 219
OtherList
SHARK_DATA, 212
out_of_bounds
error.h, 293
outer_abstol
KMS_DATA, 114
outer_iter
KMS_DATA, 114
outer_product
Matrix, 132
outer_reltol
KMS_DATA, 114
Output
ARNOLDI_DATA, 23
BiCGSTAB_DATA, 33
CGS_DATA, 37
Dove, 71
GCR_DATA, 88
GMRESLP_DATA, 91
GMRESRP_DATA, 96
MONKFISH_DATA, 150
PCG_DATA, 177
PICARD_DATA, 181
SYSTEM_DATA, 235
Output_inner
KMS_DATA, 115
Output_outer
KMS_DATA, 115
OutputFile
DOGFISH_DATA, 56
SCOPSOWL_DATA, 199
SHARK_DATA, 219
SKUA_DATA, 222
owl_dat
SCOPSOWL_OPT_DATA, 204
oxidation_state
Atom, 28
OxidationState
Atom, 27
P
GSTA_OPT_DATA, 102
p
BiCGSTAB_DATA, 34
CGS_DATA, 37
PCG_DATA, 177
P_Velocity
Trajectory.h, 407
PCG_DATA, 175
alpha, 176
Ap, 177
bestres, 177
bestx, 177
beta, 176
iter, 176
maxit, 176
Output, 177
p, 177
r, 177
r_old, 177
relres, 176
relres_base, 177
res, 176
tol_abs, 176
tol_rel, 176
x, 177
z, 177
z_old, 177
PCG
lark.h, 313
PE3
egret.h, 289
PFR
shark.h, 374
pH_index
SHARK_DATA, 214
pH_step
SHARK_DATA, 215
PICARD_DATA, 179
bestres, 181
bestx, 181
iter, 180
maxit, 180
Output, 181
r, 181
relres, 181
relres_base, 181
res, 181
tol_abs, 180
tol_rel, 180
x0, 181
PITZER
shark.h, 375
Plo
GPAST_DATA, 99
PJFNK_DATA, 181
backtrack_dat, 186
bestx, 185

bicgstab_dat, 186
Bounce, 185
cgs_dat, 186
Converged, 185
eps, 185
F, 185
fun_call, 184
funeval, 187
Fv, 185
gcr_dat, 186
gmreslp_dat, 186
gmresr_dat, 186
gmresrp_dat, 186
kms_dat, 186
L_Output, 185
l_iter, 183
l_maxit, 184
l_restart, 184
lin_tol_abs, 184
lin_tol_rel, 184
LineSearch, 185
linear_solver, 184
NL_Output, 185
nl_bestres, 185
nl_iter, 183
nl_maxit, 184
nl_relres, 184
nl_res, 184
nl_res_base, 184
nl_tol_abs, 184
nl_tol_rel, 184
pcg_dat, 186
precon, 187
precon_data, 186
qr_dat, 186
res_data, 186
v, 185
x, 185
PLASMA
mola.h, 346
pOH_index
SHARK_DATA, 214
POLAR
Trajectory.h, 407
POL
TRAJECTORY_DATA, 238
PSI
egret.h, 290
PURE_GAS, 187
density, 188
dynamic_viscosity, 188
molecular_diffusion, 188
molecular_weight, 188
Schmidt, 188
specific_heat, 188
Sutherland_Const, 188
Sutherland_Temp, 188
Sutherland_Viscosity, 188
PVel_Rad
Trajectory.h, 407
PVel_Theta
Trajectory.h, 407
Par
SYSTEM_DATA, 235
param_dat
DOGFISH_DATA, 57
MONKFISH_DATA, 151
SCOPSOWL_DATA, 200
SKUA_DATA, 223
param_data
FINCH_DATA, 85
param_guess
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
param_guess_old
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
ParamFile
SCOPSOWL_OPT_DATA, 204
SKUA_OPT_DATA, 227
Path
UI_DATA, 240
path
UI_DATA, 240
ui.h, 412
pcg
lark.h, 317
pcg_dat
PJFNK_DATA, 186
pellet_density
SCOPSOWL_DATA, 198
pellet_radius
SCOPSOWL_DATA, 198
SKUA_DATA, 222
PeriodicTable, 178
∼PeriodicTable, 178
DisplayTable, 179
number_elements, 179
PeriodicTable, 178, 179
Table, 179
pH
SHARK_DATA, 215
Phase
Molecule, 146
PhaseID
Molecule, 146
PI
magpie.h, 332
SYSTEM_DATA, 234
pi
SYSTEM_DATA, 234
picard
lark.h, 323
picard_dat
FINCH_DATA, 85
pjfnk

lark.h, 325
pjfnk_dat
FINCH_DATA, 85
Po
egret.h, 289
gsta_opt.h, 304
magpie.h, 331
po
GPAST_DATA, 99
poi
GPAST_DATA, 99
pore_diffusion
SCOPSOWL_PARAM_DATA, 206
porosity
TRAJECTORY_DATA, 237
precon
Dove, 72
PJFNK_DATA, 187
precon_data
PJFNK_DATA, 186
SHARK_DATA, 219
precond_Jac_BDF2
dove.h, 284
precond_Jac_BE
dove.h, 281
precond_Jac_CN
dove.h, 283
precond_LowerGS_BDF2
dove.h, 285
precond_LowerGS_BE
dove.h, 282
precond_LowerGS_CN
dove.h, 283
precond_SymmetricGS_BDF2
dove.h, 285
precond_SymmetricGS_BE
dove.h, 282
precond_SymmetricGS_CN
dove.h, 284
precond_Tridiag_BDF2
dove.h, 284
precond_Tridiag_BE
dove.h, 281
precond_Tridiag_CN
dove.h, 283
precond_UpperGS_BDF2
dove.h, 285
precond_UpperGS_BE
dove.h, 282
precond_UpperGS_CN
dove.h, 283
precond_type
dove.h, 281
Preconditioner
Dove, 71
preconditioner
Dove, 71
pres
FINCH_DATA, 84
present
GPAST_DATA, 99
previous_token
yaml_cpp_class, 268
Print2Console
DOGFISH_DATA, 55
MONKFISH_DATA, 148
SCOPSOWL_DATA, 197
SKUA_DATA, 222
Print2File
DOGFISH_DATA, 55
MONKFISH_DATA, 148
SCOPSOWL_DATA, 197
SKUA_DATA, 221
print2file_DOGFISH_header
dogfish.h, 274
print2file_DOGFISH_result_new
dogfish.h, 274
print2file_DOGFISH_result_old
dogfish.h, 274
print2file_SCOPSOWL_header
scopsowl.h, 355
print2file_SCOPSOWL_result_new
scopsowl.h, 355
print2file_SCOPSOWL_result_old
scopsowl.h, 355
print2file_SCOPSOWL_time_header
scopsowl.h, 355
print2file_SKUA_header
skua.h, 394
print2file_SKUA_results_new
skua.h, 394
print2file_SKUA_results_old
skua.h, 394
print2file_SKUA_time_header
skua.h, 394
print2file_dim_header
finch.h, 299
print2file_newline
finch.h, 299
print2file_result_new
finch.h, 299
print2file_result_old
finch.h, 299
print2file_shark_header
shark.h, 375
print2file_shark_info
shark.h, 375
print2file_shark_results_new
shark.h, 375
print2file_shark_results_old
shark.h, 375
print2file_species_header
dogfish.h, 274
scopsowl.h, 355
skua.h, 394
print2file_tab

finch.h, 299
print2file_time_header
finch.h, 299
print_header
Dove, 66
print_newresult
Dove, 66
print_result
Dove, 67
Protons
Atom, 27
protons
Atom, 28
Pstd
egret.h, 289
PT
SYSTEM_DATA, 234

q
GPAST_DATA, 98
GSTA_OPT_DATA, 102
q_bar
TRAJECTORY_DATA, 238
q_data
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 227
Q_in
TRAJECTORY_DATA, 237
q_p
magpie.h, 332
q_sim
SCOPSOWL_OPT_DATA, 204
SKUA_OPT_DATA, 227
qAvg
SCOPSOWL_PARAM_DATA, 205
qAvg_old
SCOPSOWL_PARAM_DATA, 205
qIntegralAvg
SCOPSOWL_PARAM_DATA, 206
qIntegralAvg_old
SCOPSOWL_PARAM_DATA, 206
QR_DATA, 188
ek, 189
Ro, 189
x, 189
QRsolve
lark.h, 323
qTn
SKUA_DATA, 221
qTnp1
SKUA_DATA, 221
qmax
GSTA_DATA, 100
GSTA_OPT_DATA, 101
qo
GPAST_DATA, 99
magpie.h, 331
SCOPSOWL_PARAM_DATA, 206
QR
lark.h, 313
qr_dat
PJFNK_DATA, 186
qrSolve
Matrix, 135
Qst
magpie.h, 332
SCOPSOWL_PARAM_DATA, 205
Qst_old
SCOPSOWL_PARAM_DATA, 206
QstAvg
SCOPSOWL_PARAM_DATA, 206
QstAvg_old
SCOPSOWL_PARAM_DATA, 206
Qstn
SKUA_PARAM, 228
Qstnp1
SKUA_PARAM, 228
Qsto
SCOPSOWL_PARAM_DATA, 206
qT
magpie.h, 333
SYSTEM_DATA, 234

R
gsta_opt.h, 304
magpie.h, 331
r
BiCGSTAB_DATA, 33
CGS_DATA, 37
GCR_DATA, 88
GMRESLP_DATA, 91
GMRESRP_DATA, 97
PCG_DATA, 177
PICARD_DATA, 181
r0
BiCGSTAB_DATA, 33
CGS_DATA, 37
r_old
PCG_DATA, 177
RADIAL_FORCE
Trajectory.h, 407
RE3
egret.h, 289
RIC
FINCH_DATA, 78
RK4
dove.h, 280
RKF
dove.h, 280
rSq
gsta_opt.h, 305
RUN_SANDBOX
sandbox.h, 351
ReNum
egret.h, 290
Reaction, 189
∼Reaction, 191
calculateEnergies, 192

calculateEquilibrium, 192
CanCalcHS, 194
CanCalcG, 194
checkSpeciesEnergies, 192
Display_Info, 191
energy, 194
enthalpy, 193
entropy, 193
Equilibrium, 193
Eval_Residual, 193
Get_Energy, 193
Get_Enthalpy, 193
Get_Entropy, 193
Get_Equilibrium, 193
Get_Stoichiometric, 192
HaveEquil, 194
haveEquilibrium, 192
HaveHS, 194
HaveG, 194
Initialize_Object, 191
List, 193
Reaction, 191
Set_Energy, 192
Set_Enthalpy, 192
Set_EnthalpyANDEntropy, 192
Set_Entropy, 192
Set_Equilibrium, 192
Set_Stoichiometric, 191
Stoichiometric, 193
ReactionList
SHARK_DATA, 211
reactor_choice
shark.h, 380
reactor_type
SHARK_DATA, 214
read_adsorbobjects
shark.h, 383
read_chemisorbobjects
shark.h, 384
read_equilrxn
shark.h, 383
read_error
error.h, 293
read_massbalance
shark.h, 383
read_multichemi_scenario
shark.h, 382
read_multichemiobjects
shark.h, 384
read_multiligand_scenario
shark.h, 382
read_multiligandobjects
shark.h, 383
read_options
shark.h, 382
read_scenario
shark.h, 382
read_species
shark.h, 383
read_unsteadyadsorbobjects
shark.h, 383
read_unsteadyrxn
shark.h, 383
readInputFile
yaml_cpp_class, 267
Recover
SYSTEM_DATA, 235
ref_diffusion
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
ref_pressure
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
ref_temperature
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
Register
Atom, 26
Molecule, 142
registerCoeff
Dove, 66
registerFunction
Dove, 65
registerJacobi
Dove, 66
registered
Molecule, 146
rel_tol_norm
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
relative_permittivity
SHARK_DATA, 216
relres
BiCGSTAB_DATA, 33
CGS_DATA, 36
GCR_DATA, 88
GMRESLP_DATA, 90
GMRESRP_DATA, 96
PCG_DATA, 176
PICARD_DATA, 181
relres_base
BiCGSTAB_DATA, 33
CGS_DATA, 37
GCR_DATA, 88
GMRESLP_DATA, 90
GMRESRP_DATA, 96
PCG_DATA, 177
PICARD_DATA, 181
Removal_Efficiency
Trajectory.h, 408
removeAllAtoms
Molecule, 144
removeElectron
Atom, 27
removeNeutron
Atom, 27

removeOneAtom
Molecule, 144
removeProton
Atom, 27
res
BiCGSTAB_DATA, 33
CGS_DATA, 36
FINCH_DATA, 84
GCR_DATA, 88
GMRESLP_DATA, 90
GMRESRP_DATA, 96
PCG_DATA, 176
PICARD_DATA, 181
res_data
PJFNK_DATA, 186
reset_all
Dove, 69
resetKeys
Document, 51
Header, 106
YamlWrapper, 271
resettime
FINCH_DATA, 85
Residual
SHARK_DATA, 218
residual
Dove, 72
residual_BDF2
dove.h, 284
residual_BE
dove.h, 281
residual_CN
dove.h, 282
residual_alkalinity
MasterSpeciesList, 127
residual_data
SHARK_DATA, 219
restart
GCR_DATA, 87
GMRESLP_DATA, 90
GMRESRP_DATA, 95
KMS_DATA, 114
revalidateAllKeys
Document, 51
YamlWrapper, 271
reverse_rate
UnsteadyReaction, 262
reverse_ref_rate
UnsteadyReaction, 262
Reynolds
MIXED_GAS, 138
rho
BACKTRACK_DATA, 30
BiCGSTAB_DATA, 32
CGS_DATA, 36
rho_f
TRAJECTORY_DATA, 237
rho_old
BiCGSTAB_DATA, 32
rho_p
TRAJECTORY_DATA, 237
Rn
FINCH_DATA, 83
Rnp1
FINCH_DATA, 83
Ro
FINCH_DATA, 79
QR_DATA, 189
Rough
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
roundIt
gsta_opt.h, 304
rowExtend
Matrix, 136
rowExtract
Matrix, 135
rowReplace
Matrix, 136
rowShrink
Matrix, 136
rows
Matrix, 131
Rs
TRAJECTORY_DATA, 237
Rstd
egret.h, 289
shark.h, 373
Run_Trajectory
Trajectory.h, 408
run_exec
ui.h, 416
run_executable
ui.h, 416
run_test
ui.h, 416
rxn_rate_error
error.h, 293

s
BiCGSTAB_DATA, 34
FINCH_DATA, 77
mSPD_DATA, 154
s_rand
TRAJECTORY_DATA, 238
SCOPSOWL_DATA, 194
binder_fraction, 198
binder_poresize, 198
binder_porosity, 198
char_macro, 198
char_micro, 198
coord_macro, 196
coord_micro, 197
crystal_radius, 198
DirichletBC, 198
eval_ads, 199
eval_diff, 199

eval_kf, 199
eval_retard, 199
eval_surfDiff, 199
finch_dat, 199
gas_dat, 199
gas_temperature, 198
gas_velocity, 197
Heterogeneous, 197
level, 197
magpie_dat, 199
NonLinear, 198
OutputFile, 199
param_dat, 200
pellet_density, 198
pellet_radius, 198
Print2Console, 197
Print2File, 197
sim_time, 197
skua_dat, 200
SurfDiff, 197
t, 197
t_counter, 197
t_old, 197
t_print, 197
tempy, 199
total_pressure, 198
total_steps, 196
user_data, 199
y, 199
SCOPSOWL_Executioner
scopsowl.h, 358
SCOPSOWL_HPP_
scopsowl.h, 354
SCOPSOWL_OPT_DATA, 200
abs_tol_bias, 203
adsorb_index, 202
CompareFile, 204
current_equil, 202
current_points, 202
current_press, 202
current_temp, 202
diffusion_type, 202
e_norm, 203
e_norm_old, 203
evaluation, 201
f_bias, 203
f_bias_old, 203
max_bias, 203
max_guess_iter, 202
min_bias, 203
num_curves, 201
num_params, 202
Optimize, 202
owl_dat, 204
param_guess, 203
param_guess_old, 203
ParamFile, 204
q_data, 203
q_sim, 204
rel_tol_norm, 203
Rough, 202
simulation_equil, 202
t, 204
total_eval, 202
y_base, 203
SCOPSOWL_OPT_set_y
scopsowl_opt.h, 365
SCOPSOWL_OPTIMIZE
scopsowl_opt.h, 365
SCOPSOWL_PARAM_DATA, 204
activation_energy, 207
Adsorbable, 207
affinity, 207
dq_dc, 206
dq_dco, 206
film_transfer, 206
pore_diffusion, 206
qAvg, 205
qAvg_old, 205
qIntegralAvg, 206
qIntegralAvg_old, 206
qo, 206
Qst, 205
Qst_old, 206
QstAvg, 206
QstAvg_old, 206
Qsto, 206
ref_diffusion, 207
ref_pressure, 207
ref_temperature, 207
speciesName, 207
xIC, 206
SCOPSOWL_SCENARIOS
scopsowl.h, 361
SCOPSOWL_TESTS
scopsowl.h, 363
SCOPSOWL_postprocesses
scopsowl.h, 359
SCOPSOWL_preprocesses
scopsowl.h, 359
SCOPSOWL_reset
scopsowl.h, 359
SCOPSOWL
scopsowl.h, 361
SGS
dove.h, 281
SHARK_DATA, 207
act_fun, 214
activity_data, 219
activity_new, 218
activity_old, 218
AdsorptionList, 212
ChemisorptionList, 212
Conc_new, 218
Conc_old, 217
Console_Output, 217

const_pH, 217
Contains_pOH, 217
Contains_pH, 217
Converged, 217
dielectric_const, 216
dt, 215
dt_max, 215
dt_min, 215
end_temp, 215
EvalActivity, 218
File_Output, 217
flow_rate, 216
ionic_strength, 216
lin_precon, 218
LocalMin, 217
MassBalanceList, 211
MasterList, 211
MultiAdsList, 212
MultiChemList, 212
Newton_data, 219
Norm, 216
num_mbe, 212
num_multi_ssao, 213
num_multi_ssar, 213
num_multi_sschem, 213
num_multichem_rxns, 213
num_other, 214
num_ssao, 213
num_ssar, 213
num_sschem, 213
num_sschem_rxns, 213
num_ssr, 212
num_usao, 213
num_usar, 213
num_usr, 213
numvar, 212
other_data, 219
OtherList, 212
OutputFile, 219
pH_index, 214
pH_step, 215
pOH_index, 214
pH, 215
precon_data, 219
ReactionList, 211
reactor_type, 214
relative_permittivity, 216
Residual, 218
residual_data, 219
shark.h, 374
simulationtime, 215
SpeciationCurve, 217
ss_ads_names, 213
ss_chem_names, 214
ssmulti_names, 214
ssmultichem_names, 214
start_temp, 215
steadystate, 216
t_count, 215
t_out, 215
temp_step, 216
temperature, 216
TemperatureCurve, 217
time, 215
time_old, 215
TimeAdaptivity, 216
timesteps, 214
totalcalls, 214
totalsteps, 214
UnsteadyAdsList, 212
UnsteadyList, 211
us_ads_names, 214
volume, 216
X_new, 217
X_old, 217
xsec_area, 216
yaml_object, 219
ZeroInitialSolids, 216
SHARK_SCENARIO
shark.h, 387
SHARK_TESTS_OLD
shark.h, 391
SHARK_TESTS
shark.h, 391
SHARK
shark.h, 387
SIT
shark.h, 375
SKUA_DATA, 219
char_measure, 222
coord, 221
DirichletBC, 222
eval_diff, 222
eval_kf, 222
finch_dat, 223
gas_dat, 223
gas_velocity, 222
magpie_dat, 222
NonLinear, 222
OutputFile, 222
param_dat, 223
pellet_radius, 222
Print2Console, 222
Print2File, 221
qTn, 221
qTnp1, 221
sim_time, 221
t, 221
t_counter, 221
t_old, 221
t_print, 221
total_steps, 221
user_data, 222
y, 222
SKUA_Executioner
skua.h, 397

SKUA_HPP_
skua.h, 394
SKUA_OPT_DATA, 223
abs_tol_bias, 226
adsorb_index, 225
CompareFile, 227
current_equil, 225
current_points, 225
current_press, 225
current_temp, 225
diffusion_type, 225
e_norm, 226
e_norm_old, 226
evaluation, 225
f_bias, 226
f_bias_old, 226
max_bias, 226
max_guess_iter, 225
min_bias, 226
num_curves, 225
num_params, 225
Optimize, 225
param_guess, 226
param_guess_old, 226
ParamFile, 227
q_data, 227
q_sim, 227
rel_tol_norm, 226
Rough, 225
simulation_equil, 226
skua_dat, 227
t, 227
total_eval, 225
y_base, 226
SKUA_OPT_set_y
skua_opt.h, 402
SKUA_OPTIMIZE
skua_opt.h, 402
SKUA_PARAM, 227
activation_energy, 228
Adsorbable, 228
affinity, 228
film_transfer, 228
Qstn, 228
Qstnp1, 228
ref_diffusion, 228
ref_pressure, 228
ref_temperature, 228
speciesName, 228
xIC, 228
xn, 228
xnp1, 228
y_eff, 228
SKUA_SCENARIOS
skua.h, 398
SKUA_TESTS
skua.h, 400
SKUA_postprocesses
skua.h, 397
SKUA_preprocesses
skua.h, 397
SKUA_reset
skua.h, 398
SKUA
skua.h, 398
SOLID
mola.h, 345
STRING
yaml_wrapper.h, 420
SYSTEM_DATA, 233
As, 234
avg_norm, 234
Carrier, 235
I, 234
Ideal, 235
J, 234
K, 234
max_norm, 235
N, 234
Output, 235
Par, 235
PI, 234
pi, 234
PT, 234
qT, 234
Recover, 235
Sys, 235
T, 234
total_eval, 234
sandbox
ui.h, 411
sandbox.h, 350
RUN_SANDBOX, 351
ScNum
egret.h, 290
scenario_fail
error.h, 293
Schmidt
PURE_GAS, 188
school.h, 351
scops_opt
ui.h, 411
scopsowl
ui.h, 411
scopsowl.h, 352
avgDp, 354
const_filmMassTransfer, 357
const_pore_diffusion, 357
default_adsorption, 355
default_effective_diffusion, 356
default_filmMassTransfer, 357
default_pore_diffusion, 355
default_retardation, 355
default_surf_diffusion, 356
Dk, 354
Dp, 354

print2file_SCOPSOWL_header, 355
print2file_SCOPSOWL_result_new, 355
print2file_SCOPSOWL_result_old, 355
print2file_SCOPSOWL_time_header, 355
print2file_species_header, 355
SCOPSOWL_Executioner, 358
SCOPSOWL_HPP_, 354
SCOPSOWL_SCENARIOS, 361
SCOPSOWL_TESTS, 363
SCOPSOWL_postprocesses, 359
SCOPSOWL_preprocesses, 359
SCOPSOWL_reset, 359
SCOPSOWL, 361
set_SCOPSOWL_ICs, 358
set_SCOPSOWL_params, 359
set_SCOPSOWL_timestep, 358
setup_SCOPSOWL_DATA, 357
zero_surf_diffusion, 356
scopsowl_opt.h, 363
eval_SCOPSOWL_Uptake, 365
initial_guess_SCOPSOWL, 365
SCOPSOWL_OPT_set_y, 365
SCOPSOWL_OPTIMIZE, 365
Set_ActivationEnergy
UnsteadyReaction, 257
Set_Affinity
UnsteadyReaction, 257
Set_Area
MassBalance, 120
set_DOGFISH_ICs
dogfish.h, 275
set_DOGFISH_params
dogfish.h, 276
set_DOGFISH_timestep
dogfish.h, 276
Set_Delta
MassBalance, 119
Set_Energy
Reaction, 192
UnsteadyReaction, 256
Set_Enthalpy
Reaction, 192
UnsteadyReaction, 256
Set_EnthalpyANDEntropy
Reaction, 192
UnsteadyReaction, 256
Set_Entropy
Reaction, 192
UnsteadyReaction, 256
Set_Equilibrium
Reaction, 192
UnsteadyReaction, 256
Set_FlowRate
MassBalance, 120
Set_Forward
UnsteadyReaction, 257
Set_ForwardRef
UnsteadyReaction, 257
Set_InitialConcentration
MassBalance, 120
Set_InitialValue
UnsteadyReaction, 256
Set_InletConcentration
MassBalance, 120
set_LineSearchMethod
Dove, 65
set_LinearAbsTol
Dove, 64
set_LinearMethod
Dove, 65
set_LinearOutput
Dove, 65
set_LinearRelTol
Dove, 65
set_LinearStatus
Dove, 65
set_MaxLinearIterations
Dove, 65
set_MaxNonLinearIterations
Dove, 65
Set_MaximumValue
UnsteadyReaction, 256
Set_Name
MassBalance, 120
set_NonlinearAbsTol
Dove, 64
set_NonlinearOutput
Dove, 65
set_NonlinearRelTol
Dove, 64
set_Preconditioning
Dove, 65
set_RecursionLevel
Dove, 65
set_RestartLimit
Dove, 65
Set_Reverse
UnsteadyReaction, 257
Set_ReverseRef
UnsteadyReaction, 257
set_SCOPSOWL_ICs
scopsowl.h, 358
set_SCOPSOWL_params
scopsowl.h, 359
set_SCOPSOWL_timestep
scopsowl.h, 358
set_SKUA_ICs
skua.h, 397
set_SKUA_params
skua.h, 397
set_SKUA_timestep
skua.h, 397
Set_Species_Index
UnsteadyReaction, 255
Set_SteadyState
MassBalance, 120

Set_Stoichiometric
Reaction, 191
UnsteadyReaction, 256
Set_TimeStep
MassBalance, 120
UnsteadyReaction, 258
Set_TotalConcentration
MassBalance, 119
Set_Type
MassBalance, 119
Set_Volume
MassBalance, 119
Set_ZeroInitialSolids
MassBalance, 120
set_alkalinity
MasterSpeciesList, 125
set_defaultCoeffs
Dove, 64
set_defaultJacobis
Dove, 64
set_endtime
Dove, 63
set_fileoutput
Dove, 64
set_initialcondition
Dove, 64
set_integrationtype
Dove, 63
set_list_size
MasterSpeciesList, 125
set_numfunc
Dove, 63
set_output
Dove, 64
set_outputfile
Dove, 64
set_preconditioner
Dove, 64
set_size
Matrix, 131
set_species
MasterSpeciesList, 125
set_timestep
Dove, 63
set_timestepmax
Dove, 63
set_timestepmin
Dove, 63
set_timestepper
Dove, 63
set_tolerance
Dove, 64
set_userdata
Dove, 64
set_variables
egret.h, 290
setActivities
AdsorptionReaction, 14
ChemisorptionReaction, 43
setActivityEnum
AdsorptionReaction, 13
ChemisorptionReaction, 42
MultiligandAdsorption, 158
MultiligandChemisorption, 168
UnsteadyAdsorption, 245
setActivityModelInfo
AdsorptionReaction, 13
ChemisorptionReaction, 42
MultiligandAdsorption, 158
MultiligandChemisorption, 168
UnsteadyAdsorption, 245
setAdsorbIndices
AdsorptionReaction, 13
ChemisorptionReaction, 41
MultiligandAdsorption, 158
MultiligandChemisorption, 167
UnsteadyAdsorption, 244
setAdsorbentName
AdsorptionReaction, 14
ChemisorptionReaction, 43
MultiligandAdsorption, 159
MultiligandChemisorption, 168
UnsteadyAdsorption, 246
setAlias
Document, 52
Header, 107
SubHeader, 231, 232
setAqueousIndex
AdsorptionReaction, 13
UnsteadyAdsorption, 245
setAqueousIndexAuto
AdsorptionReaction, 13
MultiligandAdsorption, 158
UnsteadyAdsorption, 245
setAreaBasisBool
AdsorptionReaction, 14
UnsteadyAdsorption, 246
setAreaFactor
AdsorptionReaction, 13
ChemisorptionReaction, 42
MultiligandAdsorption, 159
MultiligandChemisorption, 168
UnsteadyAdsorption, 245
setBasis
AdsorptionReaction, 14
UnsteadyAdsorption, 246
setChargeDensity
AdsorptionReaction, 15
ChemisorptionReaction, 43
MultiligandAdsorption, 160
MultiligandChemisorption, 169
UnsteadyAdsorption, 246
setChargeDensityValue
AdsorptionReaction, 14
ChemisorptionReaction, 43
setDelta

ChemisorptionReaction, 42
setDeltas
ChemisorptionReaction, 42
MultiligandChemisorption, 168
setElectricPotential
MultiligandAdsorption, 159
MultiligandChemisorption, 169
setFormula
Molecule, 142
setInputFile
yaml_cpp_class, 267
setIonicStrength
AdsorptionReaction, 15
ChemisorptionReaction, 43
MultiligandAdsorption, 160
MultiligandChemisorption, 169
UnsteadyAdsorption, 246
setIonicStrengthValue
AdsorptionReaction, 14
ChemisorptionReaction, 43
setLigandIndex
ChemisorptionReaction, 42
setLigandIndices
MultiligandChemisorption, 167
setLigandName
MultiligandAdsorption, 159
MultiligandChemisorption, 168
setMolarArea
Molecule, 143
setMolarFactor
AdsorptionReaction, 13
MultiligandAdsorption, 158
UnsteadyAdsorption, 245
setMolarVolume
Molecule, 143
setMolarWeigth
Molecule, 143
setName
Document, 52
Header, 107
SubHeader, 231
setNameAliasPair
Document, 52
Header, 107
SubHeader, 232
setSpecificArea
AdsorptionReaction, 14
ChemisorptionReaction, 42
MultiligandAdsorption, 159
MultiligandChemisorption, 168
UnsteadyAdsorption, 245
setSpecificMolality
AdsorptionReaction, 14
ChemisorptionReaction, 42
MultiligandAdsorption, 159
MultiligandChemisorption, 168
UnsteadyAdsorption, 245
setState
Document, 52
Header, 107
SubHeader, 232
setSurfaceCharge
AdsorptionReaction, 14
MultiligandAdsorption, 159
UnsteadyAdsorption, 245
setSurfaceChargeBool
AdsorptionReaction, 14
ChemisorptionReaction, 43
MultiligandAdsorption, 159
MultiligandChemisorption, 169
UnsteadyAdsorption, 246
setTotalMass
AdsorptionReaction, 14
ChemisorptionReaction, 42
MultiligandAdsorption, 159
MultiligandChemisorption, 168
UnsteadyAdsorption, 246
setTotalVolume
AdsorptionReaction, 14
ChemisorptionReaction, 43
MultiligandAdsorption, 159
MultiligandChemisorption, 169
UnsteadyAdsorption, 246
setVolumeFactor
AdsorptionReaction, 13
ChemisorptionReaction, 42
MultiligandAdsorption, 159
MultiligandChemisorption, 168
UnsteadyAdsorption, 245
setbcs
FINCH_DATA, 84
setic
FINCH_DATA, 84
setparams
FINCH_DATA, 84
setpostprocess
FINCH_DATA, 85
setpreprocess
FINCH_DATA, 84
settime
FINCH_DATA, 84
setup_DOGFISH_DATA
dogfish.h, 275
setup_FINCH_DATA
finch.h, 298
setup_MONKFISH_DATA
monkfish.h, 349
setup_SCOPSOWL_DATA
scopsowl.h, 357
setup_SHARK_DATA
shark.h, 384
setup_SKUA_DATA
skua.h, 396
shapeFactor
magpie.h, 331
shark

ui.h, 411
shark.h, 368
AbsPerm, 374
act_choice, 380
AreaSTD, 373
BATCH, 374
CSTR, 374
calculate_ionic_strength, 375
Convert2Concentration, 382
Convert2LogConcentration, 382
CoordSTD, 373
DAVIES, 375
DEBYE_HUCKEL, 375
Davies_equation, 379
DebyeHuckel_equation, 379
e, 373
FLORY_HUGGINS, 375
Faraday, 373
FloryHuggins, 376
FloryHuggins_chemi, 377
FloryHuggins_multichemi, 377
FloryHuggins_multiligand, 376
FloryHuggins_unsteady, 376
IDEAL_ADS, 375
IDEAL, 375
ideal_solution, 379
kB, 373
LengthFactor, 373
linearsolve_choice, 380
linesearch_choice, 380
Na, 373
PFR, 374
PITZER, 375
print2file_shark_header, 375
print2file_shark_info, 375
print2file_shark_results_new, 375
print2file_shark_results_old, 375
reactor_choice, 380
read_adsorbobjects, 383
read_chemisorbobjects, 384
read_equilrxn, 383
read_massbalance, 383
read_multichemi_scenario, 382
read_multichemiobjects, 384
read_multiligand_scenario, 382
read_multiligandobjects, 383
read_options, 382
read_scenario, 382
read_species, 383
read_unsteadyadsorbobjects, 383
read_unsteadyrxn, 383
Rstd, 373
SHARK_DATA, 374
SHARK_SCENARIO, 387
SHARK_TESTS_OLD, 391
SHARK_TESTS, 391
SHARK, 387
SIT, 375
setup_SHARK_DATA, 384
shark_add_customResidual, 385
shark_energy_calculations, 385
shark_executioner, 386
shark_guess, 385
shark_initial_conditions, 385
shark_pH_finder, 385
shark_parameter_check, 385
shark_postprocesses, 386
shark_preprocesses, 386
shark_reset, 386
shark_residual, 387
shark_solver, 386
shark_temperature_calculations, 385
shark_timestep_adapt, 386
shark_timestep_const, 386
surf_act_choice, 380
UNIQUAC_ACT, 375
UNIQUAC_chemi, 378
UNIQUAC_multichemi, 378
UNIQUAC_multiligand, 378
UNIQUAC_unsteady, 377
UNIQUAC, 377
VacuumPermittivity, 374
valid_act, 374
valid_mb, 374
valid_surf_act, 375
VolumeSTD, 373
WaterRelPerm, 374
shark_add_customResidual
shark.h, 385
shark_energy_calculations
shark.h, 385
shark_executioner
shark.h, 386
shark_guess
shark.h, 385
shark_initial_conditions
shark.h, 385
shark_pH_finder
shark.h, 385
shark_parameter_check
shark.h, 385
shark_postprocesses
shark.h, 386
shark_preprocesses
shark.h, 386
shark_reset
shark.h, 386
shark_residual
shark.h, 387
shark_solver
shark.h, 386
shark_temperature_calculations
shark.h, 385
shark_timestep_adapt
shark.h, 386
shark_timestep_const

shark.h, 386
sigma
CGS_DATA, 36
sigma_m
TRAJECTORY_DATA, 238
sigma_n
TRAJECTORY_DATA, 238
sigma_v
TRAJECTORY_DATA, 238
sigma_vz
TRAJECTORY_DATA, 238
sigma_z
TRAJECTORY_DATA, 238
sim_time
SCOPSOWL_DATA, 197
SKUA_DATA, 221
simple_darken_Dc
skua.h, 395
simulation_equil
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 226
simulation_fail
error.h, 292
simulationtime
SHARK_DATA, 215
single_fiber_density
MONKFISH_DATA, 149
singular_matrix
error.h, 293
size
Document, 52
Header, 107
KeyValueMap, 112
MasterSpeciesList, 127
YamlWrapper, 271
skua
ui.h, 411
skua.h, 392
const_Dc, 395
const_kf, 396
D_c, 394
D_inf, 394
D_o, 394
default_Dc, 394
default_kf, 395
empirical_kf, 396
molefractionCheck, 396
print2file_SKUA_header, 394
print2file_SKUA_results_new, 394
print2file_SKUA_results_old, 394
print2file_SKUA_time_header, 394
print2file_species_header, 394
SKUA_Executioner, 397
SKUA_HPP_, 394
SKUA_SCENARIOS, 398
SKUA_TESTS, 400
SKUA_postprocesses, 397
SKUA_preprocesses, 397
SKUA_reset, 398
SKUA, 398
set_SKUA_ICs, 397
set_SKUA_params, 397
set_SKUA_timestep, 397
setup_SKUA_DATA, 396
simple_darken_Dc, 395
theoretical_darken_Dc, 395
skua_dat
SCOPSOWL_DATA, 200
SKUA_OPT_DATA, 227
skua_opt
ui.h, 411
skua_opt.h, 400
eval_SKUA_Uptake, 402
initial_guess_SKUA, 402
SKUA_OPT_set_y, 402
SKUA_OPTIMIZE, 402
Sn
FINCH_DATA, 83
Snp1
FINCH_DATA, 83
SolnTransform
Matrix, 133
solve
FINCH_DATA, 84
solve_FE
Dove, 69
solve_RK4
Dove, 69
solve_RKF
Dove, 69
solve_all
Dove, 69
solve_timestep
Dove, 68
sorbed_molefraction
DOGFISH_PARAM, 58
MONKFISH_PARAM, 152
sorption_bc
MONKFISH_PARAM, 153
SpeciationCurve
SHARK_DATA, 217
species
DOGFISH_PARAM, 58
MONKFISH_PARAM, 153
MasterSpeciesList, 127
species_dat
MIXED_GAS, 138
species_index
UnsteadyReaction, 262
speciesName
MasterSpeciesList, 126
SCOPSOWL_PARAM_DATA, 207
SKUA_PARAM, 228
specific_area
AdsorptionReaction, 20
MultiligandAdsorption, 163

MultiligandChemisorption, 172
specific_heat
PURE_GAS, 188
specific_molality
AdsorptionReaction, 20
SphereArea
mola.h, 345
SphereVolume
mola.h, 345
Spherical
finch.h, 297
sphericalAvg
Matrix, 133
sphericalBCFill
Matrix, 133
ss_ads_names
SHARK_DATA, 213
ss_chem_names
SHARK_DATA, 214
ssmulti_names
SHARK_DATA, 214
ssmultichem_names
SHARK_DATA, 214
start_temp
SHARK_DATA, 215
state
SubHeader, 233
SteadyState
FINCH_DATA, 80
MassBalance, 123
steadystate
SHARK_DATA, 216
steps
GMRESLP_DATA, 90
Stoichiometric
Reaction, 193
string_parse_error
error.h, 293
Sub_Map
Header, 108
SubHeader, 229
∼SubHeader, 230
addPair, 231
alias, 232
clear, 231
Data_Map, 232
DisplayContents, 232
getAlias, 232
getMap, 231
getName, 232
getState, 232
isAlias, 232
isAnchor, 232
name, 232
operator=, 231
operator[], 231
setAlias, 231, 232
setName, 231
setNameAliasPair, 232
setState, 232
state, 233
SubHeader, 230, 231
sum
ARNOLDI_DATA, 23
GMRESRP_DATA, 97
Matrix, 131
Sum_Delta
MassBalance, 120
surf_act_choice
shark.h, 380
SurfDiff
SCOPSOWL_DATA, 197
surface_activity
AdsorptionReaction, 19
MultiligandAdsorption, 162
MultiligandChemisorption, 172
surface_charge
AdsorptionReaction, 20
surface_concentration
DOGFISH_PARAM, 58
Sutherland_Const
PURE_GAS, 188
Sutherland_Temp
PURE_GAS, 188
Sutherland_Viscosity
PURE_GAS, 188
Symbol
Atom, 29
Sys
SYSTEM_DATA, 235
sys_dat
MAGPIE_DATA, 116

T
FINCH_DATA, 77
SYSTEM_DATA, 234
t
BiCGSTAB_DATA, 34
FINCH_DATA, 77
SCOPSOWL_DATA, 197
SCOPSOWL_OPT_DATA, 204
SKUA_DATA, 221
SKUA_OPT_DATA, 227
t_count
SHARK_DATA, 215
t_counter
DOGFISH_DATA, 55
MONKFISH_DATA, 149
SCOPSOWL_DATA, 197
SKUA_DATA, 221
t_old
FINCH_DATA, 77
SCOPSOWL_DATA, 197
SKUA_DATA, 221
t_out
SHARK_DATA, 215
t_print

DOGFISH_DATA, 55
MONKFISH_DATA, 149
SCOPSOWL_DATA, 197
SKUA_DATA, 221
t_rand
TRAJECTORY_DATA, 238
TANGENTIAL_FORCE
Trajectory.h, 407
TEST
ui.h, 411
TRAJECTORY_DATA, 235
a, 237
A_separator, 237
A_wire, 237
b, 237
B0, 237
beta, 238
Cap, 238
chi_p, 237
dt, 238
dX, 238
dY, 238
eta, 237
H, 238
H0, 237
Hamaker, 237
k, 237
L, 237
L_wire, 237
M, 238
m_rand, 238
mp, 238
Ms, 237
mu_0, 237
n_rand, 238
POL, 238
porosity, 237
q_bar, 238
Q_in, 237
rho_f, 237
rho_p, 237
Rs, 237
s_rand, 238
sigma_m, 238
sigma_n, 238
sigma_v, 238
sigma_vz, 238
sigma_z, 238
t_rand, 238
Temp, 237
V0, 237
V_separator, 237
V_wire, 237
Vr, 238
Vt, 238
X, 238
Y, 238
Y_initial, 238
TRIDIAG
dove.h, 281
Table
PeriodicTable, 179
Temp
TRAJECTORY_DATA, 237
temp_step
SHARK_DATA, 216
temperature
SHARK_DATA, 216
temperature_affinity
UnsteadyReaction, 262
TemperatureCurve
SHARK_DATA, 217
tempy
SCOPSOWL_DATA, 199
tensor_out_of_bounds
error.h, 293
term_precon
GMRESR_DATA, 94
KMS_DATA, 115
terminal_precon
GMRESR_DATA, 94
KMS_DATA, 115
test
ui.h, 412
test_loop
ui.h, 415
theoretical_darken_Dc
skua.h, 395
time
DOGFISH_DATA, 55
Dove, 70
MONKFISH_DATA, 148
SHARK_DATA, 215
time_end
Dove, 70
time_old
DOGFISH_DATA, 55
Dove, 70
MONKFISH_DATA, 148
SHARK_DATA, 215
time_older
Dove, 70
time_step
UnsteadyReaction, 262
TimeAdaptivity
SHARK_DATA, 216
timestep_type
dove.h, 280
timestepper
Dove, 71
timesteps
SHARK_DATA, 214
token_parser
yaml_cpp_class, 268
tol_abs
BiCGSTAB_DATA, 33

CGS_DATA, 36
FINCH_DATA, 80
GCR_DATA, 87
GMRESLP_DATA, 90
GMRESRP_DATA, 96
PCG_DATA, 176
PICARD_DATA, 180
tol_rel
BiCGSTAB_DATA, 33
CGS_DATA, 36
FINCH_DATA, 80
GCR_DATA, 87
GMRESLP_DATA, 90
GMRESRP_DATA, 96
PCG_DATA, 176
PICARD_DATA, 180
tolerance
Dove, 70
total_density
MIXED_GAS, 138
total_dyn_vis
MIXED_GAS, 138
total_eval
GSTA_OPT_DATA, 101
SCOPSOWL_OPT_DATA, 202
SKUA_OPT_DATA, 225
SYSTEM_DATA, 234
total_iter
FINCH_DATA, 80
GCR_DATA, 87
GMRESR_DATA, 93
KMS_DATA, 114
total_mass
AdsorptionReaction, 20
MultiligandAdsorption, 163
MultiligandChemisorption, 173
total_molecular_weight
MIXED_GAS, 138
total_pressure
MIXED_GAS, 137
SCOPSOWL_DATA, 198
total_sorption
DOGFISH_DATA, 56
MONKFISH_DATA, 149
total_sorption_old
DOGFISH_DATA, 56
MONKFISH_DATA, 149
total_specific_heat
MIXED_GAS, 138
total_steps
DOGFISH_DATA, 55
MONKFISH_DATA, 148
SCOPSOWL_DATA, 196
SKUA_DATA, 221
total_volume
AdsorptionReaction, 20
MultiligandAdsorption, 163
MultiligandChemisorption, 173
TotalConcentration
MassBalance, 122
totalcalls
SHARK_DATA, 214
totalsteps
SHARK_DATA, 214
trajectory
ui.h, 411
Trajectory.h, 405
Brown_RAD, 407
Brown_THETA, 407
CARTESIAN, 407
Capture_Force, 407
DISPLACEMENT, 408
Grav_R, 407
Grav_T, 407
In_P_Velocity, 407
In_PVel_Rad, 407
In_PVel_Theta, 407
LOCATION, 408
Magnetic_R, 407
Magnetic_T, 407
Number_Generator, 408
P_Velocity, 407
POLAR, 407
PVel_Rad, 407
PVel_Theta, 407
RADIAL_FORCE, 407
Removal_Efficiency, 408
Run_Trajectory, 408
TANGENTIAL_FORCE, 407
Trajectory_SetupConstants, 408
V_RAD, 407
V_THETA, 407
Van_R, 407
Trajectory_SetupConstants
Trajectory.h, 408
transpose
Matrix, 132
transpose_dat
GCR_DATA, 89
transpose_multiply
Matrix, 132
tridiagonalFill
Matrix, 133
tridiagonalSolve
Matrix, 132
tridiagonalVectorFill
Matrix, 134
twoFifths
gsta_opt.h, 304
Type
MassBalance, 122
type
ValueTypePair, 265

u
CGS_DATA, 37
GCR_DATA, 88

u_star
FINCH_DATA, 82
u_temp
GCR_DATA, 88
uAverage
finch.h, 297
uAvg
FINCH_DATA, 78
uAvg_old
FINCH_DATA, 78
UGS
dove.h, 281
UI_DATA, 239
argc, 240
argv, 240
BasicUI, 240
count, 240
Files, 240
input_files, 240
max, 240
MissingArg, 240
option, 240
Path, 240
path, 240
user_input, 239
value_type, 239
UI_HPP_
ui.h, 410
uIC
FINCH_DATA, 78
UNIQUAC_ACT
shark.h, 375
UNIQUAC_chemi
shark.h, 378
UNIQUAC_multichemi
shark.h, 378
UNIQUAC_multiligand
shark.h, 378
UNIQUAC_unsteady
shark.h, 377
UNIQUAC
shark.h, 377
UNKNOWN
yaml_wrapper.h, 420
uT_old
FINCH_DATA, 78
uTotal
finch.h, 297
ubest
FINCH_DATA, 82
ui.h, 408
aui_help, 411
bui_help, 411
CONTINUE, 411
display_help, 414
display_version, 414
dogfish, 411
dove, 411
ECO_EXECUTABLE, 410
ECO_VERSION, 410
EXECUTE, 411
EXIT, 411
eel, 411
egret, 411
exec, 412
exec_loop, 416
exit, 411
finch, 411
gsta_opt, 411
HELP, 411
help, 412
input, 413
invalid_input, 414
lark, 411
macaw, 411
magpie, 411
mola, 411
monkfish, 411
number_files, 413
path, 412
run_exec, 416
run_executable, 416
run_test, 416
sandbox, 411
scops_opt, 411
scopsowl, 411
shark, 411
skua, 411
skua_opt, 411
TEST, 411
test, 412
test_loop, 415
trajectory, 411
UI_HPP_, 410
valid_addon_options, 414
valid_exec_string, 413
valid_input_execute, 415
valid_input_main, 415
valid_input_tests, 415
valid_options, 411
valid_test_string, 413
version, 412
un
Dove, 70
FINCH_DATA, 82
unm1
Dove, 70
FINCH_DATA, 82
unp1
Dove, 70
FINCH_DATA, 82
unregistered_name
error.h, 293
unstable_matrix
error.h, 293
UnsteadyAdsList

SHARK_DATA, 212
UnsteadyAdsorption, 241
~UnsteadyAdsorption, 244
activities_old, 251
ads_rxn, 251
calculateActiveFraction, 247
calculateAqueousChargeExchange, 247
calculateAreaFactors, 246
calculateEquilibria, 246
calculateEquilibriumCorrection, 248
calculatePsi, 247
calculateRates, 246
calculateSurfaceChargeDensity, 247
callSurfaceActivity, 247
checkAqueousIndices, 244
Display_Info, 244
Eval_IC_Residual, 249
Eval_ReactionRate, 249
Eval_Residual, 248
Explicit_Eval, 249
getActivity, 250
getActivityEnum, 251
getAdsorbIndex, 251
getAdsorbentName, 251
getAqueousIndex, 251
getAreaFactor, 250
getBulkDensity, 250
getChargeDensity, 250
getIonicStrength, 251
getMolarFactor, 250
getNumberRxns, 251
getOldActivity, 250
getReaction, 249
getSpecificArea, 250
getSpecificMolality, 250
getSurfaceCharge, 250
getTotalMass, 250
getTotalVolume, 250
getVolumeFactor, 250
includeSurfaceCharge, 251
Initialize_Object, 244
isAreaBasis, 251
modifyDeltas, 244
setActivityEnum, 245
setActivityModelInfo, 245
setAdsorbIndices, 244
setAdsorbentName, 246
setAqueousIndex, 245
setAqueousIndexAuto, 245
setAreaBasisBool, 246
setAreaFactor, 245
setBasis, 246
setChargeDensity, 246
setIonicStrength, 246
setMolarFactor, 245
setSpecificArea, 245
setSpecificMolality, 245
setSurfaceCharge, 245
setSurfaceChargeBool, 246
setTotalMass, 246
setTotalVolume, 246
setVolumeFactor, 245
UnsteadyAdsorption, 244
updateActivities, 246
UnsteadyList
SHARK_DATA, 211
UnsteadyReaction, 252
~UnsteadyReaction, 255
activation_energy, 262
calculateEnergies, 258
calculateEquilibrium, 258
calculateRate, 258
checkSpeciesEnergies, 258
Display_Info, 255
Eval_IC_Residual, 261
Eval_ReactionRate, 260
Eval_Residual, 260, 261
Explicit_Eval, 261
forward_rate, 261
forward_ref_rate, 262
Get_ActivationEnergy, 260
Get_Affinity, 260
Get_Energy, 259
Get_Enthalpy, 259
Get_Entropy, 259
Get_Equilibrium, 259
Get_Forward, 259
Get_ForwardRef, 260
Get_InitialValue, 259
Get_MaximumValue, 259
Get_Reverse, 260
Get_ReverseRef, 260
Get_Species_Index, 259
Get_Stoichiometric, 259
Get_TimeStep, 260
haveEquilibrium, 258
HaveForRef, 262
HaveForward, 262
haveForward, 259
haveForwardRef, 258
haveRate, 258
HaveRevRef, 262
HaveReverse, 262
haveReverse, 259
haveReverseRef, 259
initial_value, 261
Initialize_Object, 255
max_value, 261
reverse_rate, 262
reverse_ref_rate, 262
Set_ActivationEnergy, 257
Set_Affinity, 257
Set_Energy, 256
Set_Enthalpy, 256
Set_EnthalpyANDEntropy, 256
Set_Entropy, 256

Set_Equilibrium, 256
Set_Forward, 257
Set_ForwardRef, 257
Set_InitialValue, 256
Set_MaximumValue, 256
Set_Reverse, 257
Set_ReverseRef, 257
Set_Species_Index, 255
Set_Stoichiometric, 256
Set_TimeStep, 258
species_index, 262
temperature_affinity, 262
time_step, 262
UnsteadyReaction, 255
uo
FINCH_DATA, 78
Update
FINCH_DATA, 79
update_arnoldi_solution
lark.h, 313
update_states
Dove, 69
update_timestep
Dove, 69
updateActivities
UnsteadyAdsorption, 246
upperHessenberg2Triangular
Matrix, 135
upperHessenbergSolve
Matrix, 135
upperTriangularSolve
Matrix, 135
us_ads_names
SHARK_DATA, 214
user_coeff
Dove, 72
user_data
DOGFISH_DATA, 56
Dove, 72
MONKFISH_DATA, 151
SCOPSOWL_DATA, 199
SKUA_DATA, 222
user_func
Dove, 71
user_input
UI_DATA, 239
user_jacobi
Dove, 72
uT
FINCH_DATA, 78
uz_l_E
FINCH_DATA, 82
uz_l_I
FINCH_DATA, 82
uz_lm1_E
FINCH_DATA, 82
uz_lm1_I
FINCH_DATA, 82
uz_lp1_E
FINCH_DATA, 82
uz_lp1_I
FINCH_DATA, 82

V
magpie.h, 331
v
ARNOLDI_DATA, 23
BiCGSTAB_DATA, 34
CGS_DATA, 37
GMRESRP_DATA, 97
mSPD_DATA, 154
PJFNK_DATA, 185
V0
TRAJECTORY_DATA, 237
V_RAD
Trajectory.h, 407
V_THETA
Trajectory.h, 407
V_separator
TRAJECTORY_DATA, 237
V_wire
TRAJECTORY_DATA, 237
vIC
FINCH_DATA, 78
VacuumPermittivity
shark.h, 374
valence_e
Atom, 28
valid_act
shark.h, 374
valid_addon_options
ui.h, 414
valid_exec_string
ui.h, 413
valid_input_execute
ui.h, 415
valid_input_main
ui.h, 415
valid_input_tests
ui.h, 415
valid_mb
shark.h, 374
valid_options
ui.h, 411
valid_phase
mola.h, 345
valid_surf_act
shark.h, 375
valid_test_string
ui.h, 413
validate_linearsteps
Dove, 69
validate_precond
Dove, 68
Value_Type
ValueTypePair, 265
value_type

UI_DATA, 239
ValueTypePair, 263
~ValueTypePair, 264
assertType, 264
DisplayPair, 265
editPair, 264
editValue, 264
findType, 264
getBool, 265
getDouble, 265
getInt, 265
getPair, 265
getString, 265
getType, 265
getValue, 265
operator=, 264
type, 265
Value_Type, 265
ValueTypePair, 264
Van_R
Trajectory.h, 407
vanAlbada_discretization
finch.h, 300
vector_out_of_bounds
error.h, 293
velocity
MIXED_GAS, 138
version
ui.h, 412
Vk
ARNOLDI_DATA, 23
GMRESRP_DATA, 97
vn
FINCH_DATA, 83
vnp1
FINCH_DATA, 83
vo
FINCH_DATA, 78
volume
MassBalance, 122
SHARK_DATA, 216
volume_factors
AdsorptionReaction, 20
VolumeSTD
shark.h, 373
Vr
TRAJECTORY_DATA, 238
Vt
TRAJECTORY_DATA, 238

w
ARNOLDI_DATA, 23
CGS_DATA, 37
GMRESRP_DATA, 97
WaterRelPerm
shark.h, 374
weightedAvg
gsta_opt.h, 305

X
TRAJECTORY_DATA, 238
x
BiCGSTAB_DATA, 33
CGS_DATA, 37
GCR_DATA, 88
GMRESLP_DATA, 91
GMRESRP_DATA, 96
GPAST_DATA, 98
PCG_DATA, 177
PJFNK_DATA, 185
QR_DATA, 189
x0
PICARD_DATA, 181
X_new
SHARK_DATA, 217
X_old
SHARK_DATA, 217
xIC
SCOPSOWL_PARAM_DATA, 206
SKUA_PARAM, 228
xk
BACKTRACK_DATA, 30
xn
SKUA_PARAM, 228
xnp1
SKUA_PARAM, 228
xsec_area
MassBalance, 122
SHARK_DATA, 216

Y
TRAJECTORY_DATA, 238
y
BiCGSTAB_DATA, 34
GMRESRP_DATA, 97
GPAST_DATA, 98
SCOPSOWL_DATA, 199
SKUA_DATA, 222
y_base
SCOPSOWL_OPT_DATA, 203
SKUA_OPT_DATA, 226
y_eff
SKUA_PARAM, 228
Y_initial
TRAJECTORY_DATA, 238
YAML_CPP_TEST
yaml_wrapper.h, 421
YAML_WRAPPER_TESTS
yaml_wrapper.h, 421
yaml_cpp_class, 266
~yaml_cpp_class, 267
cleanup, 267
current_token, 268
DisplayContents, 267
executeYamlRead, 267
file_name, 267
getYamlWrapper, 267
input_file, 267

previous_token, 268
readInputFile, 267
setInputFile, 267
token_parser, 268
yaml_cpp_class, 267
yaml_wrapper, 267
yaml_object
SHARK_DATA, 219
yaml_wrapper
yaml_cpp_class, 267
yaml_wrapper.h, 416
ALIAS, 421
ANCHOR, 421
allLower, 421
BOOLEAN, 420
DOUBLE, 420
data_type, 420
header_state, 420
INT, 420
isEven, 421
NONE, 421
STRING, 420
UNKNOWN, 420
YAML_CPP_TEST, 421
YAML_WRAPPER_TESTS, 421
YamlWrapper, 268
∼YamlWrapper, 270
addDocKey, 271
begin, 270, 271
changeKey, 271
clear, 271
copyAnchor2Alias, 271
DisplayContents, 271
Doc_Map, 272
end, 270
getAnchoredDoc, 271
getDocFromHeadAlias, 271
getDocFromSubAlias, 271
getDocMap, 270
getDocument, 270
operator(), 270
operator=, 270
resetKeys, 271
revalidateAllKeys, 271
size, 271
YamlWrapper, 270
yk
ARNOLDI_DATA, 23

Z
magpie.h, 330
z
BiCGSTAB_DATA, 34
CGS_DATA, 38
PCG_DATA, 177
z_old
PCG_DATA, 177
zero_surf_diffusion
scopsowl.h, 356
zero_vector
error.h, 293
ZeroInitialSolids
MassBalance, 123
SHARK_DATA, 216
zeros
Matrix, 131
Zk
GMRESRP_DATA, 97