Abstract

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

This report serves as a developers manual for the DAKOTA software and describes the DAKOTA class hierarchies and their interrelationships. It derives directly from annotation of the actual source code and provides detailed class documentation, including all member functions and attributes.
# Contents

1 DAKOTA Developers Manual ................................. 11
   1.1 Introduction ......................................... 11
   1.2 Overview of DAKOTA ................................. 11
   1.3 Services ............................................. 15
   1.4 Additional Resources ................................. 16

2 DAKOTA Hierarchical Index ............................... 17
   2.1 DAKOTA Class Hierarchy ............................. 17

3 DAKOTA Compound Index ................................. 21
   3.1 DAKOTA Compound List ............................. 21

4 DAKOTA File Index ........................................ 25
   4.1 DAKOTA File List .................................... 25

5 DAKOTA Page Index ........................................ 27
   5.1 DAKOTA Related Pages ............................... 27

6 DAKOTA Class Documentation ............................. 29
   6.1 AllMergedVarConstraints Class Reference .......... 29
   6.2 AllMergedVariables Class Reference ................ 32
   6.3 AllVarConstraints Class Reference ................. 36
   6.4 AllVariables Class Reference ....................... 39
   6.5 AnalysisCode Class Reference ....................... 43
   6.6 ANNSurf Class Reference ............................ 46
   6.7 ApplicationInterface Class Reference .............. 48
   6.8 ApproximationInterface Class Reference .......... 59
   6.9 BaseConstructor Struct Reference ................. 62
   6.10 BranchBndStrategy Class Reference ............... 63
   6.11 COLINApplication Class Template Reference .... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Class / Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.12</td>
<td>COLINOptimizer Class Template Reference</td>
</tr>
<tr>
<td>6.13</td>
<td>ColinPoint Class Reference</td>
</tr>
<tr>
<td>6.14</td>
<td>CommandLineHandler Class Reference</td>
</tr>
<tr>
<td>6.15</td>
<td>CommandShell Class Reference</td>
</tr>
<tr>
<td>6.16</td>
<td>ConcurrentStrategy Class Reference</td>
</tr>
<tr>
<td>6.17</td>
<td>CONMINOptimizer Class Reference</td>
</tr>
<tr>
<td>6.18</td>
<td>CtelRegexp Class Reference</td>
</tr>
<tr>
<td>6.19</td>
<td>DACEIterator Class Reference</td>
</tr>
<tr>
<td>6.20</td>
<td>DakotaApproximation Class Reference</td>
</tr>
<tr>
<td>6.21</td>
<td>DakotaArray Class Template Reference</td>
</tr>
<tr>
<td>6.22</td>
<td>DakotaBaseVector Class Template Reference</td>
</tr>
<tr>
<td>6.23</td>
<td>DakotaBiStream Class Reference</td>
</tr>
<tr>
<td>6.24</td>
<td>DakotaBoStream Class Reference</td>
</tr>
<tr>
<td>6.25</td>
<td>DakotaGraphics Class Reference</td>
</tr>
<tr>
<td>6.26</td>
<td>DakotaInterface Class Reference</td>
</tr>
<tr>
<td>6.27</td>
<td>DakotaIterator Class Reference</td>
</tr>
<tr>
<td>6.28</td>
<td>DakotaLeastSq Class Reference</td>
</tr>
<tr>
<td>6.29</td>
<td>DakotaList Class Template Reference</td>
</tr>
<tr>
<td>6.30</td>
<td>DakotaMatrix Class Template Reference</td>
</tr>
<tr>
<td>6.31</td>
<td>DakotaModel Class Reference</td>
</tr>
<tr>
<td>6.32</td>
<td>DakotaNonD Class Reference</td>
</tr>
<tr>
<td>6.33</td>
<td>DakotaOptimizer Class Reference</td>
</tr>
<tr>
<td>6.34</td>
<td>DakotaOptLeastSq Class Reference</td>
</tr>
<tr>
<td>6.35</td>
<td>DakotaResponse Class Reference</td>
</tr>
<tr>
<td>6.36</td>
<td>DakotaResponseRep Class Reference</td>
</tr>
<tr>
<td>6.37</td>
<td>DakotaStrategy Class Reference</td>
</tr>
<tr>
<td>6.38</td>
<td>DakotaString Class Reference</td>
</tr>
<tr>
<td>6.39</td>
<td>DakotaVarConstraints Class Reference</td>
</tr>
<tr>
<td>6.40</td>
<td>DakotaVariables Class Reference</td>
</tr>
<tr>
<td>6.41</td>
<td>DakotaVector Class Template Reference</td>
</tr>
<tr>
<td>6.42</td>
<td>DataInterface Class Reference</td>
</tr>
<tr>
<td>6.43</td>
<td>DataMethod Class Reference</td>
</tr>
<tr>
<td>6.44</td>
<td>DataResponses Class Reference</td>
</tr>
<tr>
<td>6.45</td>
<td>DataStrategy Class Reference</td>
</tr>
<tr>
<td>6.46</td>
<td>DataVariables Class Reference</td>
</tr>
<tr>
<td>6.47</td>
<td>DirectFnApplicInterface Class Reference</td>
</tr>
<tr>
<td>Section</td>
<td>Class/Struct Reference</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>6.48</td>
<td>DOTOptimizer Class Reference</td>
</tr>
<tr>
<td>6.49</td>
<td>ErrorTable Struct Reference</td>
</tr>
<tr>
<td>6.50</td>
<td>ForkAnalysisCode Class Reference</td>
</tr>
<tr>
<td>6.51</td>
<td>ForkApplicInterface Class Reference</td>
</tr>
<tr>
<td>6.52</td>
<td>FunctionCompare Class Template Reference</td>
</tr>
<tr>
<td>6.53</td>
<td>FundamentalVarConstraints Class Reference</td>
</tr>
<tr>
<td>6.54</td>
<td>FundamentalVariables Class Reference</td>
</tr>
<tr>
<td>6.55</td>
<td>GetLongOpt Class Reference</td>
</tr>
<tr>
<td>6.56</td>
<td>HermiteSurf Class Reference</td>
</tr>
<tr>
<td>6.57</td>
<td>HierLayeredModel Class Reference</td>
</tr>
<tr>
<td>6.58</td>
<td>KrigApprox Class Reference</td>
</tr>
<tr>
<td>6.59</td>
<td>KrigingSurf Class Reference</td>
</tr>
<tr>
<td>6.60</td>
<td>LayeredModel Class Reference</td>
</tr>
<tr>
<td>6.61</td>
<td>MARSSurf Class Reference</td>
</tr>
<tr>
<td>6.62</td>
<td>MergedVarConstraints Class Reference</td>
</tr>
<tr>
<td>6.63</td>
<td>MergedVariables Class Reference</td>
</tr>
<tr>
<td>6.64</td>
<td>MultilevelOptStrategy Class Reference</td>
</tr>
<tr>
<td>6.65</td>
<td>NestedModel Class Reference</td>
</tr>
<tr>
<td>6.66</td>
<td>NLSSOLLeastSq Class Reference</td>
</tr>
<tr>
<td>6.67</td>
<td>NoDBBaseConstructor Struct Reference</td>
</tr>
<tr>
<td>6.68</td>
<td>NonAdvMeanValue Class Reference</td>
</tr>
<tr>
<td>6.69</td>
<td>NonDLHSSampling Class Reference</td>
</tr>
<tr>
<td>6.70</td>
<td>NonDOptStrategy Class Reference</td>
</tr>
<tr>
<td>6.71</td>
<td>NonDPCESampling Class Reference</td>
</tr>
<tr>
<td>6.72</td>
<td>NonDSampling Class Reference</td>
</tr>
<tr>
<td>6.73</td>
<td>NPSOLOptimizer Class Reference</td>
</tr>
<tr>
<td>6.74</td>
<td>ParallelLibrary Class Reference</td>
</tr>
<tr>
<td>6.75</td>
<td>ParamResponsePair Class Reference</td>
</tr>
<tr>
<td>6.76</td>
<td>ParamStudy Class Reference</td>
</tr>
<tr>
<td>6.77</td>
<td>ProblemDescDB Class Reference</td>
</tr>
<tr>
<td>6.78</td>
<td>RespSurf Class Reference</td>
</tr>
<tr>
<td>6.79</td>
<td>rSQPOptimizer Class Reference</td>
</tr>
<tr>
<td>6.80</td>
<td>SGOPTApplication Class Reference</td>
</tr>
<tr>
<td>6.81</td>
<td>SGOPTOptimizer Class Reference</td>
</tr>
<tr>
<td>6.82</td>
<td>SingleMethodStrategy Class Reference</td>
</tr>
<tr>
<td>6.83</td>
<td>SingleModel Class Reference</td>
</tr>
</tbody>
</table>
11.2 Rebuild IDR ................................................................. 390
11.3 Update keywordtable.C in $DAKOTA/src .......................... 390
11.4 Update ProblemDescDB.C in $DAKOTA/src ...................... 390
11.5 Update Corresponding Data Classes ............................... 393
11.6 Use get_<data_type>() Functions ................................. 394
11.7 Update the Documentation .......................................... 394
Chapter 1

DAKOTA Developers Manual

Author:
Michael S. Eldred, Anthony A. Giunta, Bart G. van Bloemen Waanders, Steven F. Wojtkiewicz, Jr., William E. Hart, Mario P. Alleva

1.1 Introduction

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible, extensible interface between analysis codes and iteration methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods, uncertainty quantification with sampling, analytic reliability, and stochastic finite element methods, parameter estimation with nonlinear least squares methods, and sensitivity/main effects analysis with design of experiments and parameter study capabilities. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible problem-solving environment as well as a platform for rapid prototyping of new solution approaches.

The Developers Manual focuses on documentation of the class structures used by the DAKOTA system. It derives directly from annotation of the actual source code. For information on input command syntax, refer to the Reference Manual, and for a tour of DAKOTA features and capabilities, refer to the Users Manual.

1.2 Overview of DAKOTA

In the DAKOTA system, the strategy creates and manages iterators and models. In the simplest case, the strategy creates a single iterator and a single model and executes the iterator on the model to perform a single study. In a more advanced case, a hybrid optimization strategy might manage a global optimizer operating on a low-fidelity model in coordination with a local optimizer operating on a high-fidelity model. And on the high end, a surrogate-based optimization under uncertainty strategy would employ an uncertainty quantification iterator nested within an optimization iterator and would employ truth models layered
within surrogate models. Thus, iterators and models provide both stand-alone capabilities as well as building blocks for more sophisticated studies.

A model contains a set of variables, an interface, and a set of responses, and the iterator operates on the model to map the variables into responses using the interface. Each of these components is a flexible abstraction with a variety of specializations for supporting different types of iterative studies. In a DAKOTA input file, the user specifies these components through strategy, method, variables, interface, and responses keyword specifications.

The use of class hierarchies provides a clear direction for extensibility in DAKOTA components. In each of the various class hierarchies, adding a new capability typically involves deriving a new class and providing a small number of virtual function redefinitions. These redefinitions define the coding portions specific to the new derived class, with the common portions already defined at the base class. Thus, with a small amount of new code, the existing facilities can be extended, reused, and leveraged for new purposes.

The software components are presented in the following sections using a top-down order.

1.2.1 Strategies

Class hierarchy: DakotaStrategy.

Strategies provide a control layer for creation and management of iterators and models. Specific strategies include:

- **SingleMethodStrategy**: the simplest strategy. A single iterator is run on a single model to perform a single study.
- **MultilevelOptStrategy**: hybrid optimization using a succession of iterators employing a succession of models of varying fidelity. The best results obtained are passed from one iterator to the next.
- **SurrBasedOptStrategy**: surrogate-based optimization. Employs a single iterator with a LayeredModel (either data fit or hierarchical). A sequence of approximate optimizations is performed, each of which involves build, optimize, and verify steps.
- **NonDOptStrategy**: optimization under uncertainty (OUU). Employs a single optimization iterator with a NestedModel. This NestedModel contains a sub-iterator and sub-model for performing uncertainty quantifications. In OUU approaches involving surrogates, NestedModels and LayeredModels can be chained together in a variety of ways using recursion in sub-models.
- **BranchBndStrategy**: mixed integer nonlinear programming using the PICO library for parallel branch and bound. Employs a single iterator with a single model, but runs multiple instances of the iterator concurrently for different variable bounds within the model.
- **ConcurrentStrategy**: two similar algorithms are available: (1) multi-start iteration from several different starting points, and (2) pareto set optimization for several different multiobjective weightings. Employs a single iterator with a single model, but runs multiple instances of the iterator concurrently for different settings within the model.

1.2.2 Iterators

Class hierarchy: DakotaIterator.

The iterator hierarchy contains a variety of iterative algorithms for optimization, uncertainty quantification, nonlinear least squares, design of experiments, and parameter studies.
1.2 Overview of DAKOTA

- Optimization: DakotaOptimizer provides a base class for DOTOptimizer, CONMINOptimizer, NPSOLOptimizer, rSQPOptimizer, SNLLOptimizer, SGOPTOptimizer, and COLINOptimizer.

- Uncertainty quantification: DakotaNonD inherits from DakotaIterator and provides a base class for NonDAdvMeanValue and NonDSampling. NonDSampling is then further specialized with the NonNLOHSSampling and NonDPCESampling derived classes.

- Parameter estimation: DakotaLeastSq provides a base class for SNLLLeastSq, a Gauss-Newton least squares solver, and NLSSOLLeastSq, an SQP-based least squares solver.

- Design of experiments: DACEIterator inherits directly from DakotaIterator. NonDLHSSampling from the uncertainty quantification branch also supports a design of experiments mode.

- Parameter studies: ParamStudy inherits directly from DakotaIterator.

1.2.3 Models

Class hierarchy: DakotaModel.

The model classes are responsible for mapping variables into responses when an iterator makes a function evaluation request. There are several types of models, some supporting sub-iterators and sub-models for enabling layered and nested relationships. When sub-models are used, they may be of arbitrary type so that a variety of recursions are supported.

- **SingleModel**: variables are mapped into responses using a single DakotaInterface object. No sub-iterators or sub-models are used.

- **LayeredModel**: variables are mapped into responses using an approximation. The approximation is built and/or corrected using data from a sub-model (the truth model) and the data may be obtained using a sub-iterator (a design of experiments iterator). LayeredModel has two derived classes: SurrLayeredModel for data fit surrogates and HierLayeredModel for hierarchical models of varying fidelity. The relationship of the sub-iterators and sub-models is considered to be "layered" since they are not used as part of every response evaluation on the top level model, but rather used periodically in surrogate update and verification steps.

- **NestedModel**: variables are mapped into responses using a combination of an optional DakotaInterface and a sub-iterator/sub-model pair. The relationship of the sub-iterators and sub-models is considered to be "nested" since they are used to perform a complete iterative study as part of every response evaluation on the top level model.

1.2.4 Variables

Class hierarchy: DakotaVariables.

The DakotaVariables class hierarchy manages design, uncertain, and state variable types for continuous and discrete domain types. This hierarchy is specialized according to various views of the data.

- **FundamentalVariables**: both variable and domain type distinctions are retained, i.e. separate arrays for design, uncertain, and state variables types and for continuous and discrete domains.

- **AllVariables**: variable types are combined and domain type distinction is retained, i.e. design, uncertain, and state variable types combined into a single continuous variables array and a single discrete variables array.
• **MergedVariables**: variable type distinction is retained and domain types are combined, i.e. continuous and discrete variables merged into continuous arrays (integrality is relaxed) for design, uncertain, and state variable types.

• **AllMergedVariables**: both variable and domain types are combined, i.e. design, uncertain, and state variable types combined (all) and continuous and discrete domain types combined (merged). The result is a single array of continuous variables.

The variables view that is chosen depends on the type of iterative study. For design optimization and uncertainty quantification, for example, variable and domain type distinctions are important and a **FundamentalVariables** view is used. For parameter studies and design of experiments, however, the variable type distinctions can be ignored and an **AllVariables** view is used. Finally, the branch and bound strategy relies on relaxation of integrality so that continuous optimizers may be used for mixed integer problems. In this case, a **MergedVariables** view is used.

The **DakotaVarConstraints** hierarchy contains the same specializations for managing linear and bound constraints on the variables (see **FundamentalVarConstraints**, **AllVarConstraints**, **MergedVarConstraints**, **AllMergedVarConstraints**).

### 1.2.5 Interfaces

Class hierarchy: **DakotaInterface**.

Interfaces provide access to simulation codes or, conversely, approximations based on simulation code data. In the simulation case, an **ApplicationInterface** is used. **ApplicationInterface** is specialized according to the simulation invocation mechanism, for which the following nonintrusive approaches

- **SysCallApplicInterface**: the simulation is invoked using a system call (the C function `system()`). Asynchronous invocation utilizes a background system call. Utilizes the **SysCallAnalysisCode** class to define syntax for input filter, analysis code, output filter, or combined spawning, which in turn utilize the **CommandShell** overloaded operator definitions.

- **ForkApplicInterface**: the simulation is invoked using a fork (the `fork/exec/wait` family of functions). Asynchronous invocation utilizes a nonblocking fork. Utilizes the **ForkAnalysisCode** class for lower level fork operations.

- **GridApplicInterface**: the simulation is invoked using distributed resource facilities. This capability is experimental and still under development. The design is evolving into the use of Condor and/or Globus tools.

and the following semi-intrusive approach

- **DirectFnApplicInterface**: the simulation is linked into the DAKOTA executable and is invoked using a procedure call. Asynchronous invocation utilizes a nonblocking thread (capability not yet available).

are supported. Scheduling of jobs for asynchronous local, message passing, and hybrid parallelism approaches is performed in the **ApplicationInterface** class, with job initiation and job capture specifics implemented in the derived classes.

In the data fit approximation case, global, multipoint, or local approximations to simulation code response data can be built and used as surrogates for the actual, expensive simulation. The interface class providing this capability is
1.3 Services

- **ApproximationInterface**: builds an approximation using data from a truth model and then employs the approximation for mapping variables to responses. This class contains an array of DakotaApproximation objects, one per response function, which allows mixing of approximation types (using the DakotaApproximation derived classes: ANNSurf, KrigingSurf, MARSSurf, RespSurf, HermiteSurf, and TaylorSurf).

Note: in the data fit approximation case, SurrLayeredModel provides the bulk of the surrogate management logic. It contains an ApproximationInterface object which provides the approximate parameter to response mappings. In the hierarchical approximation case, an ApproximationInterface object is not used since HierLayeredModel contains low and high fidelity application interfaces.

1.2.6 Responses

Class: DakotaResponse.

The DakotaResponse class provides an abstract data representation of response functions and their first and second derivatives (gradient vectors and Hessian matrices). These response functions can be interpreted as an objective function and constraints (optimization data set), residual functions and constraints (least squares data set), or generic response functions (uncertainty quantification data set). This class is not currently part of a class hierarchy, since the abstraction has been sufficiently general and has not required specialization.

1.3 Services

A variety of services are provided in DAKOTA for parallel computing, failure capturing, restart, graphics, etc. An overview of the classes and member functions involved in performing these services is included below.

- Multilevel parallel computing: DAKOTA supports up to 4 nested levels of parallelism: a strategy can manage concurrent iterators, each of which manages concurrent function evaluations, each of which manages concurrent analyses executing on multiple processors. Partitioning of these levels with MPI communicators is managed in ParallelLibrary and scheduling routines for the levels are part of ConcurrentStrategy, ApplicationInterface, and ForkApplicInterface.
- Parsing: DAKOTA employs the Input Deck Reader (IDR) parser to retrieve information from user input files. Parsing options are processed in CommandLineHandler and parsing occurs in ProblemDescDB::manageInputs() called from main.C. IDR populates data within the ProblemDescDB support class, which maintains a DataStrategy specification and lists of DataMethod, DataVariables, DataInterface, and DataResponses specifications. Procedures for modifying the parsing subsystem are described in Instructions for Modifying DAKOTA's Input Specification.
- Failure capturing: Simulation failures can be trapped and managed using exception handling in ApplicationInterface and its derived classes.
- Restart: DAKOTA maintains a record of all function evaluations both in memory (for capturing any duplication) and on the file system (for restarting runs). Restart options are processed in CommandLineHandler, restart file management occurs in ParallelLibrary::manageOutputs_restart() called from main.C, and restart file insertions occur in ApplicationInterface. The dakota-restartutil executable, built from restartutil.C, provides a variety of services for interrogating, converting, repairing, concatenating, and post-processing restart files.
• Memory management: DAKOTA employs the techniques of reference counting and representation sharing through the use of letter-envelope and handle-body idioms (Coplien, ”Advanced C++”). The former idiom provides for memory efficiency and enhanced polymorphism in the following class hierarchies: DakotaStrategy, DakotaIterator, DakotaModel, DakotaVariables, DakotaVarConstraints, DakotaInterface, and DakotaApproximation. The latter idiom provides for memory efficiency in data-intensive classes which do not involve a class hierarchy. Currently, only the DakotaResponse class uses this idiom.

• Graphics: DAKOTA provides 2D iteration history graphics using Motif widgets and 3D surface plotting graphics from the PLPLOT package. Graphics data can also be catalogued in a tabular data file for post-processing with 3rd party tools such as Matlab, Tecplot, etc. All of these capabilities are encapsulated within the DakotaGraphics class.

1.4 Additional Resources

Additional development resources include:

• Recommended Practices for DAKOTA Development
• Instructions for Modifying DAKOTA's Input Specification
• The execution of function evaluations is a core component of DAKOTA involving several class hierarchies. An overview of the classes and member functions involved in performing these evaluations is provided in Performing Function Evaluations.
• In addition to its normal usage as a stand-alone application, DAKOTA may be interfaced as an algorithm library as described in Interfacing with DAKOTA as a Library.
Chapter 2

DAKOTA Hierarchical Index

2.1 DAKOTA Class Hierarchy

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

AnalysisCode ................................................................. 43
ForkAnalysisCode ........................................................... 221
SysCallAnalysisCode ...................................................... 366
BaseConstructor .............................................................. 62
COLINApplication< DomainT, ResponseT > .............................. 65
ColinPoint ................................................................. 70
CommandShell ............................................................. 72
CtelRegexp ................................................................. 83
DakotaApproximation ...................................................... 89
          ANNSurf .......................................................... 46
          HermiteSurf ...................................................... 240
          KrigingSurf ...................................................... 254
          MARSSurf ........................................................ 261
          RespSurf .......................................................... 325
          TaylorSurf ........................................................ 370
DakotaArray< T > ........................................................... 94
DakotaBaseVector< T > ...................................................... 98
DakotaVector< T > .......................................................... 184
DakotaBaseVector< DakotaBaseVector< T > > ..................... 98
DakotaMatrix< T > ........................................................ 130
DakotaBiStream ........................................................ 102
DakotaBoStream ........................................................ 105
DakotaGraphics .......................................................... 108
DakotaInterface .......................................................... 111
          ApplicationInterface ............................................. 48
          DirectFnApplicInterface ....................................... 212
          ForkApplicInterface ............................................ 223
          SysCallApplicInterface ......................................... 368
          ApproximationInterface ........................................ 59
DakotaIterator ........................................................... 117
DACEIterator ........................................................... 85
<table>
<thead>
<tr>
<th>Class/Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DakotaNonD</td>
<td>143</td>
</tr>
<tr>
<td>NonDAdvMeanValue</td>
<td>282</td>
</tr>
<tr>
<td>NonDSampling</td>
<td>295</td>
</tr>
<tr>
<td>NonDLHSSampling</td>
<td>289</td>
</tr>
<tr>
<td>NonDPCESampling</td>
<td>293</td>
</tr>
<tr>
<td>DakotaOptLeastSq</td>
<td>150</td>
</tr>
<tr>
<td>DakotaLeastSq</td>
<td>124</td>
</tr>
<tr>
<td>NLSSOLLeastSq</td>
<td>279</td>
</tr>
<tr>
<td>SNLLLeastSq</td>
<td>342</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
<td>147</td>
</tr>
<tr>
<td>COLINOptimizer&lt; OptimizerT &gt;</td>
<td>68</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>76</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>216</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>298</td>
</tr>
<tr>
<td>rSQPOptimizer</td>
<td>327</td>
</tr>
<tr>
<td>SGOPTOptimizer</td>
<td>332</td>
</tr>
<tr>
<td>SNLLOptimizer</td>
<td>345</td>
</tr>
<tr>
<td>LayeredModel</td>
<td>132</td>
</tr>
<tr>
<td>HierLayeredModel</td>
<td>256</td>
</tr>
<tr>
<td>SurrLayeredModel</td>
<td>242</td>
</tr>
<tr>
<td>NestedModel</td>
<td>273</td>
</tr>
<tr>
<td>SingleModel</td>
<td>337</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>154</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>158</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>163</td>
</tr>
<tr>
<td>BranchBndStrategy</td>
<td>63</td>
</tr>
<tr>
<td>ConcurrentStrategy</td>
<td>74</td>
</tr>
<tr>
<td>MultilevelOptStrategy</td>
<td>270</td>
</tr>
<tr>
<td>NonDOptStrategy</td>
<td>291</td>
</tr>
<tr>
<td>SingleMethodStrategy</td>
<td>335</td>
</tr>
<tr>
<td>SurrBasedOptStrategy</td>
<td>354</td>
</tr>
<tr>
<td>DakotaString</td>
<td>169</td>
</tr>
<tr>
<td>DakotaVarConstraints</td>
<td>172</td>
</tr>
<tr>
<td>AllmergedVarConstraints</td>
<td>29</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>FundamentalVarConstraints</td>
<td>227</td>
</tr>
<tr>
<td>MergedVarConstraints</td>
<td>263</td>
</tr>
<tr>
<td>DakotaVariables</td>
<td>178</td>
</tr>
<tr>
<td>AllmergedVariables</td>
<td>32</td>
</tr>
<tr>
<td>AllVariables</td>
<td>39</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>231</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>266</td>
</tr>
<tr>
<td>DataInterface</td>
<td>188</td>
</tr>
<tr>
<td>DataMethod</td>
<td>193</td>
</tr>
<tr>
<td>DataResponses</td>
<td>200</td>
</tr>
<tr>
<td>DataStrategy</td>
<td>203</td>
</tr>
<tr>
<td>DataVariables</td>
<td>207</td>
</tr>
<tr>
<td>ErrorTable</td>
<td>220</td>
</tr>
<tr>
<td>FunctionCompare&lt; T &gt;</td>
<td>226</td>
</tr>
<tr>
<td>Class Name</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>GetLongOpt</td>
<td>236</td>
</tr>
<tr>
<td>CommandLineHandler</td>
<td>71</td>
</tr>
<tr>
<td>KrigApprox</td>
<td>246</td>
</tr>
<tr>
<td>NoDBBaseConstructor</td>
<td>281</td>
</tr>
<tr>
<td>ParallelLibrary</td>
<td>301</td>
</tr>
<tr>
<td>ParamResponsePair</td>
<td>313</td>
</tr>
<tr>
<td>ProblemDescDB</td>
<td>320</td>
</tr>
<tr>
<td>SGOPTApplication</td>
<td>329</td>
</tr>
<tr>
<td>SNLLBase</td>
<td>339</td>
</tr>
<tr>
<td>SNLLLeastSq</td>
<td>342</td>
</tr>
<tr>
<td>SNLLOptimizer</td>
<td>345</td>
</tr>
<tr>
<td>SOLBase</td>
<td>350</td>
</tr>
<tr>
<td>NLSSOLOleastSq</td>
<td>279</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>298</td>
</tr>
<tr>
<td>SortCompare&lt;T&gt;</td>
<td>353</td>
</tr>
<tr>
<td>SurrogateDataPoint</td>
<td>364</td>
</tr>
<tr>
<td>VariablesUtil</td>
<td>371</td>
</tr>
<tr>
<td>AllMergedVarConstraints</td>
<td>29</td>
</tr>
<tr>
<td>AllMergedVariables</td>
<td>32</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>AllVariables</td>
<td>39</td>
</tr>
<tr>
<td>FundamentalVarConstraints</td>
<td>227</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>231</td>
</tr>
<tr>
<td>MergedVarConstraints</td>
<td>263</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>266</td>
</tr>
</tbody>
</table>
Chapter 3

DAKOTA Compound Index

3.1 DAKOTA Compound List

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

- **AllMergedVarConstraints**: (Derived class within the *DakotaVarConstraints* hierarchy which combines the all and merged data views) .................................................. 29
- **AllMergedVariables**: (Derived class within the *DakotaVariables* hierarchy which combines the all and merged data views) .................................................. 32
- **AllVarConstraints**: (Derived class within the *DakotaVarConstraints* hierarchy which employs the all data view) .................................................. 36
- **AllVariables**: (Derived class within the *DakotaVariables* hierarchy which employs the all data view) .................................................. 39
- **AnalysisCode**: (Base class providing common functionality for derived classes (*SysCallAnalysisCode* and *ForkAnalysisCode*) which spawn separate processes for managing simulations) .................................................. 43
- **ANNSurf**: (Derived approximation class for artificial neural networks) .................................................. 46
- **ApplicationInterface**: (Derived class within the interface class hierarchy for supporting interfaces to simulation codes) .................................................. 48
- **ApproximationInterface**: (Derived class within the interface class hierarchy for supporting approximations to simulation-based results) .................................................. 59
- **BaseConstructor**: (Dummy struct for overloading letter-envelope constructors) .................................................. 62
- **BranchBndStrategy**: (Strategy for mixed integer nonlinear programming using the PICO parallel branch and bound engine) .................................................. 63
- **COLINApplication<T, ResponseT>**: .................................................. 65
- **CONMINOptimizer<OptimizerT>**: (Wrapper class for optimizers defined using COLIN) .................................................. 68
- **ColinPoint**: ................................................................................. 70
- **CommandLineHandler**: (Utility class for managing command line inputs to DAKOTA) .................................................. 71
- **CommandShell**: (Utility class which defines convenience operators for spawning processes with system calls) .................................................. 72
- **ConcurrentStrategy**: (Strategy for multi-start iteration or pareto set optimization) .................................................. 74
- **CONMINOptimizer**: (Wrapper class for the CONMIN optimization library) .................................................. 76
- **CtelRegexp**: ................................................................................. 83
- **DACEIterator**: (Wrapper class for the DDACE design of experiments library) .................................................. 85
- **DakotaApproximation**: (Base class for the approximation class hierarchy) .................................................. 89
- **DakotaArray<T>**: (Template class for the Dakota bookkeeping array) .................................................. 94
- **DakotaBaseVector<T>**: (Base class for the *DakotaMatrix* and *DakotaVector* classes) .................................................. 98
- **DakotaBiStream**: (The binary input stream class. Overloads the >> operator for all data types) .................................................. 102
- **DakotaBoStream**: (The binary output stream class. Overloads the << operator for all data types) .................................................. 105
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DakotaGraphics</td>
<td>Single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.</td>
<td>108</td>
</tr>
<tr>
<td>DakotaInterface</td>
<td>Base class for the interface class hierarchy</td>
<td>111</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>Base class for the iterator class hierarchy</td>
<td>117</td>
</tr>
<tr>
<td>DakotaLeastSq</td>
<td>Base class for the nonlinear least squares branch of the iterator hierarchy</td>
<td>124</td>
</tr>
<tr>
<td>DakotaList&lt;T&gt;</td>
<td>Template class for the Dakota bookkeeping list</td>
<td>126</td>
</tr>
<tr>
<td>DakotaMatrix&lt;T&gt;</td>
<td>Template class for the Dakota numerical matrix</td>
<td>130</td>
</tr>
<tr>
<td>DakotaModel</td>
<td>Base class for the model class hierarchy</td>
<td>132</td>
</tr>
<tr>
<td>DakotaNonID</td>
<td>Base class for all nondeterministic iterators (the DAKOTA/UQ branch)</td>
<td>143</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
<td>Base class for the optimizer branch of the iterator hierarchy</td>
<td>147</td>
</tr>
<tr>
<td>DakotaOptLeastSq</td>
<td>Base class for the optimizer and least squares branches of the iterator hierarchy</td>
<td>150</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>Container class for response functions and their derivatives. DakotaResponse provides the handle class</td>
<td>154</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>Container class for response functions and their derivatives. DakotaResponseRep provides the body class</td>
<td>158</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>Base class for the strategy class hierarchy</td>
<td>163</td>
</tr>
<tr>
<td>DakotaString</td>
<td>DakotaString class, used as main string class for Dakota</td>
<td>169</td>
</tr>
<tr>
<td>DakotaVarConstraints</td>
<td>Base class for the variable constraints class hierarchy</td>
<td>172</td>
</tr>
<tr>
<td>DakotaVariables</td>
<td>Base class for the variables class hierarchy</td>
<td>178</td>
</tr>
<tr>
<td>DakotaVector&lt;T&gt;</td>
<td>Template class for the Dakota numerical vector</td>
<td>184</td>
</tr>
<tr>
<td>DataInterface</td>
<td>Container class for interface specification data</td>
<td>188</td>
</tr>
<tr>
<td>DataMethod</td>
<td>Container class for method specification data</td>
<td>193</td>
</tr>
<tr>
<td>DataResponses</td>
<td>Container class for responses specification data</td>
<td>200</td>
</tr>
<tr>
<td>DataStrategy</td>
<td>Container class for strategy specification data</td>
<td>203</td>
</tr>
<tr>
<td>DataVariables</td>
<td>Container class for variables specification data</td>
<td>207</td>
</tr>
<tr>
<td>DirectFnApplicInterface</td>
<td>Derived application interface class which spawns simulation codes and testers using direct procedure calls</td>
<td>212</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>Wrapper class for the DOT optimization library</td>
<td>216</td>
</tr>
<tr>
<td>ErrorTable</td>
<td>Data structure to hold errors</td>
<td>220</td>
</tr>
<tr>
<td>ForkAnalysisCode</td>
<td>Derived class in the AnalysisCode class hierarchy which spawns simulations using forks</td>
<td>221</td>
</tr>
<tr>
<td>ForkApplicInterface</td>
<td>Derived application interface class which spawns simulation codes using forks</td>
<td>223</td>
</tr>
<tr>
<td>FunctionCompare&lt;T&gt;</td>
<td>Derived class within the DakotaVarConstraints hierarchy which employs the default data view (no variable or domain type array merging)</td>
<td>226</td>
</tr>
<tr>
<td>FundamentalVarConstraints</td>
<td>Derived class within the DakotaVarConstraints hierarchy which employs the default data view (no variable or domain type array merging)</td>
<td>227</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>Derived class within the DakotaVariables hierarchy which employs the default data view (no variable or domain type array merging)</td>
<td>231</td>
</tr>
<tr>
<td>GetLongOpt</td>
<td>GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France))</td>
<td>236</td>
</tr>
<tr>
<td>HermiteSurf</td>
<td>Derived approximation class for Hermite polynomials (global approximation)</td>
<td>240</td>
</tr>
<tr>
<td>HierLayeredModel</td>
<td>Derived model class within the layered model branch for managing hierarchical surrogates (models of varying fidelity)</td>
<td>242</td>
</tr>
<tr>
<td>KrigApprox</td>
<td>Utility class for kriging interpolation</td>
<td>246</td>
</tr>
<tr>
<td>KrigingSurf</td>
<td>Derived approximation class for kriging interpolation</td>
<td>254</td>
</tr>
<tr>
<td>LayeredModel</td>
<td>Base class for the layered models (SurrLayeredModel and HierLayeredModel)</td>
<td>256</td>
</tr>
<tr>
<td>MARSurf</td>
<td>Derived approximation class for multivariate adaptive regression splines</td>
<td>261</td>
</tr>
<tr>
<td>MergedVarConstraints</td>
<td>Derived class within the DakotaVarConstraints hierarchy which employs the merged data view</td>
<td>263</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>Derived class within the DakotaVariables hierarchy which employs the merged data view</td>
<td>266</td>
</tr>
<tr>
<td>MultilevelOptStrategy</td>
<td>Strategy for hybrid optimization using multiple optimizers on multiple models of varying fidelity</td>
<td>270</td>
</tr>
</tbody>
</table>
NestedModel (Derived model class which performs a complete sub-iterator execution within
every evaluation of the model) .............................................. 273
NLSSOLLeastSq (Wrapper class for the NLSSOL nonlinear least squares library) .......... 279
NoDBBaseConstructor (Dummy struct for overloading constructors used in on-the-fly instantia-
tions) ........................................................................ 281
NonDA dvMeanValue (Class for the analytical reliability methods within DAKOTA/UQ) .... 282
NonDLHSSampling (Performs LHS and Monte Carlo sampling for uncertainty quantification) . . 289
NonDOptStrategy (Strategy for optimization under uncertainty (robust and reliability-based de-
sign)) ...................................................................... 291
NonDPCESampling (Stochastic finite element approach to uncertainty quantification using poly-
nomial chaos expansions) .............................................. 293
NonDSampling (Base class for common code between NonDLHSSampling and
NonDPCESampling) .......................................................... 295
NPSOLOptimizer (Wrapper class for the NPSOL optimization library) ......................... 298
ParallelLibrary (Class for managing partitioning of multiple levels of parallelism and message
passing within the levels) ...................................................... 301
ParamResponsePair (Container class for a variables object, a response object, and an evaluation id) 313
ParamStudy (Class for vector, list, centered, and multidimensional parameter studies) ........ 316
ProblemDescDB (The database containing information parsed from the DAKOTA input file) ...... 320
RespSurf (Derived approximation class for polynomial regression) ................................. 325
rSQPOptimizer .................................................................. 327
SGOPTApplication (Maps the evaluation functions used by SGOPT algorithms to the DAKOTA
evaluation functions) ....................................................... 329
SGOPTOptimizer (Wrapper class for the SGOPT optimization library) ......................... 332
SingleMethodStrategy (Simple fall-through strategy for running a single iterator on a single model) 335
SingleModel (Derived model class which utilizes a single interface to map variables into responses) 337
SNLLLBase (Base class for OPT++ optimization and least squares methods) .................. 339
SNLLLLeastSq (Wrapper class for the OPT++ optimization library) ............................... 342
SNLLOptimizer (Wrapper class for the OPT++ optimization library) ............................. 345
SOLBase (Base class for Stanford SOL software) ....................................................... 350
SortCompare<T> ................................................................ 353
SurrBasedOptStrategy (Strategy for provably-convergent surrogate-based optimization) ...... 354
SurrLayeredModel (Derived model class within the layered model branch for managing data fit
surrogates (global and local)) .................................................. 359
SurrogateDataPoint (Simple container class encapsulating basic parameter and response data for
defining a "truth" data point) .................................................. 364
SysCallAnalysisCode (Derived class in the AnalysisCode class hierarchy which spawns simula-
tions using system calls) ....................................................... 366
SysCallApplicationInterface (Derived application interface class which spawns simulation codes us-
ing system calls) ............................................................. 368
TaylorSurf (Derived approximation class for 1st order Taylor series (local approximation)) ........ 370
VariablesUtil (Utility class for the DakotaVariables and DakotaVarConstraints hierarchies which
provides convenience functions for variable vectors and label arrays for combining de-
sign, uncertain, and state variable types and merging continuous and discrete variable
domains) .................................................................. 371
Chapter 4

DAKOTA File Index

4.1 DAKOTA File List

Here is a list of all documented files with brief descriptions:

- **keywordtable.C** (File containing keywords for the strategy, method, variables, interface, and responses input specifications from `dakota.input.spec`) ........................................... 373
- **main.C** (File containing the main program for DAKOTA) ........................................... 374
- **restart_util.C** (File containing the DAKOTA restart utility main program) ............... 375
Chapter 5

DAKOTA Page Index

5.1 DAKOTA Related Pages

Here is a list of all related documentation pages:
- Interfacing with DAKOTA as a Library .................................................. 379
- Performing Function Evaluations ......................................................... 383
- Recommended Practices for DAKOTA Development ............................... 385
- Instructions for Modifying DAKOTA's Input Specification ....................... 389
Chapter 6

DAKOTA Class Documentation

6.1 AllMergedVarConstraints Class Reference

Derived class within the DakotaVarConstraints hierarchy which combines the all and merged data views. Inheritance diagram for AllMergedVarConstraints:

```
DakotaVarConstraints
  VariablesUtil
    AllMergedVarConstraints
```

Public Methods

- `AllMergedVarConstraints (const ProblemDescDB &problem_db)`
  constructor.

- `~AllMergedVarConstraints ()`
  destructor.

- `const DakotaRealVector & continuous_lower_bounds () const`
  return the active continuous variable lower bounds.

- `void continuous_lower_bounds (const DakotaRealVector &c_l_bnds)`
  set the active continuous variable lower bounds.

- `const DakotaRealVector & continuous_upper_bounds () const`
  return the active continuous variable upper bounds.

- `void continuous_upper_bounds (const DakotaRealVector &c_u_bnds)`
  set the active continuous variable upper bounds.`
- `const DakotaIntVector & discrete_lower_bounds()` return the active discrete variable lower bounds.
- `void discrete_lower_bounds(const DakotaIntVector & dlbnds)` set the active discrete variable lower bounds.
- `const DakotaIntVector & discrete_upper_bounds()` return the active discrete variable upper bounds.
- `void discrete_upper_bounds(const DakotaIntVector & dulbnds)` set the active discrete variable upper bounds.
- `const DakotaRealVector & inactive_continuous_lower_bounds()` return the inactive continuous lower bounds.
- `void inactive_continuous_lower_bounds(const DakotaRealVector & iclbnds)` set the inactive continuous lower bounds.
- `const DakotaRealVector & inactive_continuous_upper_bounds()` return the inactive continuous upper bounds.
- `void inactive_continuous_upper_bounds(const DakotaRealVector & icubnds)` set the inactive continuous upper bounds.
- `const DakotaIntVector & inactive_discrete_lower_bounds()` return the inactive discrete lower bounds.
- `void inactive_discrete_lower_bounds(const DakotaIntVector & idlbnds)` set the inactive discrete lower bounds.
- `const DakotaIntVector & inactive_discrete_upper_bounds()` return the inactive discrete upper bounds.
- `void inactive_discrete_upper_bounds(const DakotaIntVector & idubnds)` set the inactive discrete upper bounds.
- `DakotaRealVector all_continuous_lower_bounds()` return a single array with all continuous lower bounds.
- `DakotaRealVector all_continuous_upper_bounds()` return a single array with all continuous upper bounds.
- `DakotaIntVector all_discrete_lower_bounds()` return a single array with all discrete lower bounds.
- `DakotaIntVector all_discrete_upper_bounds()` return a single array with all discrete upper bounds.
- `void write(ostream &s)` const
write a variable constraints object to an ostream.

- void read (istream &s)
  read a variable constraints object from an istream.

Private Attributes

- DakotaRealVector allMergedLowerBnds
  a continuous lower bounds array combining design, uncertain, and state variable types and merging continuous and discrete domains. The order is continuous design, discrete design, uncertain, continuous state, and discrete state.

- DakotaRealVector allMergedUpperBnds
  a continuous upper bounds array combining design, uncertain, and state variable types and merging continuous and discrete domains. The order is continuous design, discrete design, uncertain, continuous state, and discrete state.

6.1.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which combines the all and merged data views.

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllMergedVarConstraints derived class combines design, uncertain, and state variable types (all) and continuous and discrete domain types (merged). The result is a single continuous lower bounds array (allMergedLowerBnds) and a single continuous upper bounds array (allMergedUpperBnds). No iterators/strategies currently use this approach; it is included for completeness and future capability.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 AllMergedVarConstraints::AllMergedVarConstraints (const ProblemDescDB & problem_db)

can be implemented.

constructor.

Extract fundamental variable bounds and combine them into allMergedLowerBnds and allMergedUpperBnds using utilities from VariablesUtil.

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

- AllMergedVarConstraints.H
- AllMergedVarConstraints.C
6.2 AllMergedVariables Class Reference

Derived class within the DakotaVariables hierarchy which combines the all and merged data views. 

Inheritance diagram for AllMergedVariables:

```
DakotaVariables          VariablesUtil
                      |                  |
                      |                  |
                      |                  |
                      v                  v
AllMergedVariables
```

Public Methods

- `AllMergedVariables ()`
  
  default constructor.

- `AllMergedVariables (const ProblemDescDB &problem_db)`
  
  standard constructor.

- `~AllMergedVariables ()`
  
  destructor.

- `size_t tv () const`
  
  Returns total number of vars.

- `size_t cv () const`
  
  Returns number of active continuous vars.

- `size_t dv () const`
  
  Returns number of active discrete vars.

- `const DakotaRealVector & continuous_variables () const`
  
  return the active continuous variables.

- `void continuous_variables (const DakotaRealVector &c_vars)`
  
  set the active continuous variables.

- `const DakotaIntVector & discrete_variables () const`
  
  return the active discrete variables.

- `void discrete_variables (const DakotaIntVector &d_vars)`
  
  set the active discrete variables.

- `const DakotaStringArray & continuous_variable_labels () const`
  
  return the active continuous variable labels.
void continuous_variable_labels (const DakotaStringArray &cv_labels)
  set the active continuous variable labels.

const DakotaStringArray & discrete_variable_labels () const
  return the active discrete variable labels.

void discrete_variable_labels (const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

const DakotaRealVector & inactive_continuous_variables () const
  return the inactive continuous variables.

void inactive_continuous_variables (const DakotaRealVector &i_cvars)
  set the inactive continuous variables.

const DakotaIntVector & inactive_discrete_variables () const
  return the inactive discrete variables.

void inactive_discrete_variables (const DakotaIntVector &i_dvars)
  set the inactive discrete variables.

size_t acv () const
  returns total number of continuous vars.

size_t adv () const
  returns total number of discrete vars.

DakotaRealVector all_continuous_variables () const
  returns a single array with all continuous variables.

DakotaIntVector all_discrete_variables () const
  returns a single array with all discrete variables.

DakotaStringArray all_continuous_variable_labels () const
  returns a single array with all continuous variable labels.

DakotaStringArray all_discrete_variable_labels () const
  returns a single array with all discrete variable labels.

void read (istream &s)
  read a variables object from an istream.

void write (ostream &s) const
  write a variables object to an ostream.

void read_annotated (istream &s)
  read a variables object in annotated format from an istream.

void write_annotated (ostream &s) const
write a variables object in annotated format to an ostream.

- void **read** (DakotaBiStream &s)  
  read a variables object from the binary restart stream.

- void **write** (DakotaBoStream &s) const  
  write a variables object to the binary restart stream.

- void **read** (UnPackBuffer &s)  
  read a variables object from a packed MPI buffer.

- void **write** (PackBuffer &s) const  
  write a variables object to a packed MPI buffer.

**Private Methods**

- void **copy** (const DakotaVariables *vars)  
  Used by **copy()** to copy the contents of a letter class.

**Private Attributes**

- DakotaRealVector **allMergedVars**  
  a continuous array combining design, uncertain, and state variable types and merging continuous and discrete domains. The order is continuous design, discrete design, uncertain, continuous state, and discrete state.

- DakotaStringArray **allMergedLabels**  
  an array containing labels for continuous design, discrete design, uncertain, continuous state, and discrete state variables.

**Friends**

- bool **operator==** (const AllMergedVariables &vars1, const AllMergedVariables &vars2)  
  equality operator.

**6.2.1 Detailed Description**

Derived class within the DakotaVariables hierarchy which combines the all and merged data views.

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllMergedVariables derived class combines design, uncertain, and state variable types (all) and continuous and discrete domain types (merged). The result is a single array of continuous variables (allMergedVars). No iterators/strategies currently use this approach; it is included for completeness and future capability.
6.2 AllMergedVariables Class Reference

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AllMergedVariables::AllMergedVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels and combine them into allMergedVars and allMergedLabels using utilities from VariablesUtil.

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

- AllMergedVariables.H
- AllMergedVariables.C
6.3 AllVarConstraints Class Reference

Derived class within the DakotaVarConstraints hierarchy which employs the all data view. Inheritance diagram for AllVarConstraints::

![Inheritance Diagram]

Public Methods

- **AllVarConstraints** (const ProblemDescDB &problem_db)
  
  *constructor.*

- **~AllVarConstraints ()**
  
  *destructor.*

- const DakotaRealVector & continuous_lower_bounds () const
  
  *return the active continuous variable lower bounds.*

- void continuous_lower_bounds (const DakotaRealVector &c_l_bounds)
  
  *set the active continuous variable lower bounds.*

- const DakotaRealVector & continuous_upper_bounds () const
  
  *return the active continuous variable upper bounds.*

- void continuous_upper_bounds (const DakotaRealVector &c_u_bounds)
  
  *set the active continuous variable upper bounds.*

- const DakotaIntVector & discrete_lower_bounds () const
  
  *return the active discrete variable lower bounds.*

- void discrete_lower_bounds (const DakotaIntVector &d_l_bounds)
  
  *set the active discrete variable lower bounds.*

- const DakotaIntVector & discrete_upper_bounds () const
  
  *return the active discrete variable upper bounds.*

- void discrete_upper_bounds (const DakotaIntVector &d_u_bounds)
  
  *set the active discrete variable upper bounds.*

- const DakotaRealVector & inactive_continuous_lower_bounds () const
  
  *return the inactive continuous lower bounds.*
6.3 AllVarConstraints Class Reference

- void inactive_continuous_lower_bounds (const DakotaRealVector &i_c_lbnds) 
  set the inactive continuous lower bounds.

- const DakotaRealVector & inactive_continuous_upper_bounds () const
  return the inactive continuous upper bounds.

- void inactive_continuous_upper_bounds (const DakotaRealVector &i_c_ubnds) 
  set the inactive continuous upper bounds.

- const DakotaIntVector & inactive_discrete_lower_bounds () const
  return the inactive discrete lower bounds.

- void inactive_discrete_lower_bounds (const DakotaIntVector &i_d_lbnds) 
  set the inactive discrete lower bounds.

- const DakotaIntVector & inactive_discrete_upper_bounds () const
  return the inactive discrete upper bounds.

- void inactive_discrete_upper_bounds (const DakotaIntVector &i_d_ubnds) 
  set the inactive discrete upper bounds.

- DakotaRealVector all_continuous_lower_bounds () const
  returns a single array with all continuous lower bounds.

- DakotaRealVector all_continuous_upper_bounds () const
  returns a single array with all continuous upper bounds.

- DakotaIntVector all_discrete_lower_bounds () const
  returns a single array with all discrete lower bounds.

- DakotaIntVector all_discrete_upper_bounds () const
  returns a single array with all discrete upper bounds.

- void write (ostream &s) const
  write a variable constraints object to an ostream.

- void read (istream &s)
  read a variable constraints object from an istream.

Private Attributes

- DakotaRealVector allContinuousLowerBnds
  a continuous lower bounds array combining continuous design, uncertain, and continuous state variable types (all view).

- DakotaRealVector allContinuousUpperBnds
  a continuous upper bounds array combining continuous design, uncertain, and continuous state variable types (all view).
- DakotaIntVector `allDiscreteLowerBnds`
  a discrete lower bounds array combining discrete design and discrete state variable types (all view).

- DakotaIntVector `allDiscreteUpperBnds`
  a discrete upper bounds array combining discrete design and discrete state variable types (all view).

- `size_t numCDV`
  number of continuous design variables.

- `size_t numDDV`
  number of discrete design variables.

- `size_t numUV`
  number of uncertain variables.

- `size_t numCSV`
  number of continuous state variables.

- `size_t numDSV`
  number of discrete state variables.

### 6.3.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which employs the all data view.

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllVarConstraints derived class combines design, uncertain, and state variable types but separates continuous and discrete domain types. The result is combined continuous bounds arrays (`allContinuousLowerBnds`, `allContinuousUpperBnds`) and combined discrete bounds arrays (`allDiscreteLowerBnds`, `allDiscreteUpperBnds`). Parameter and DACE studies currently use this approach (see DakotaVariables::get_variables(problem_db) for variables type selection; variables type is passed to the DakotaVarConstraints constructor in DakotaModel).

### 6.3.2 Constructor & Destructor Documentation

#### 6.3.2.1 AllVarConstraints::AllVarConstraints (const ProblemDescDB & problem_db)

constructor.

Extract fundamental lower and upper bounds and combine them into allContinuousLowerBnds, allContinuousUpperBnds, allDiscreteLowerBnds, and allDiscreteUpperBnds using utilities from VariablesUtil.

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

- AllVarConstraints.H
- AllVarConstraints.C
6.4 AllVariables Class Reference

Derived class within the DakotaVariables hierarchy which employs the all data view.

Inheritance diagram for AllVariables:

```
       DakotaVariables
          |            |
          |            |
          VariablesUtil
          |            |
          |            |
       AllVariables
```

Public Methods

- **AllVariables()**
  
  *default constructor.*

- **AllVariables(const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **~AllVariables()**
  
  *destructor.*

- **size_tv() const**
  
  *Returns total number of vars.*

- **size_cv() const**
  
  *Returns number of active continuous vars.*

- **size_dv() const**
  
  *Returns number of active discrete vars.*

- **const DakotaRealVector & continuous_variables() const**
  
  *return the active continuous variables.*

- **void continuous_variables(const DakotaRealVector &c_vars)**
  
  *set the active continuous variables.*

- **const DakotaIntVector & discrete_variables() const**
  
  *return the active discrete variables.*

- **void discrete_variables(const DakotaIntVector &d_vars)**
  
  *set the active discrete variables.*

- **const DakotaStringArray & continuous_variable_labels() const**
  
  *return the active continuous variable labels.*
void continuous_variable_labels (const DakotaStringArray &cv_labels)
  set the active continuous variable labels.

const DakotaStringArray & discrete_variable_labels () const
  return the active discrete variable labels.

void discrete_variable_labels (const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

const DakotaRealVector & inactive_continuous_variables () const
  return the inactive continuous variables.

void inactive_continuous_variables (const DakotaRealVector &ic_vars)
  set the inactive continuous variables.

const DakotaIntVector & inactive_discrete_variables () const
  return the inactive discrete variables.

void inactive_discrete_variables (const DakotaIntVector &id_vars)
  set the inactive discrete variables.

size_t acv () const
  returns total number of continuous vars.

size_t adv () const
  returns total number of discrete vars.

DakotaRealVector all_continuous_variables () const
  returns a single array with all continuous variables.

DakotaIntVector all_discrete_variables () const
  returns a single array with all discrete variables.

DakotaStringArray all_continuous_variable_labels () const
  returns a single array with all continuous variable labels.

DakotaStringArray all_discrete_variable_labels () const
  returns a single array with all discrete variable labels.

void read (istream &s)
  read a variables object from an istream.

void write (ostream &s) const
  write a variables object to an ostream.

void read_annotated (istream &s)
  read a variables object in annotated format from an istream.

void write_annotated (ostream &s) const
write a variables object in annotated format to an ostream.

- void read (DakotaBiStream &s)  
  read a variables object from the binary restart stream.

- void write (DakotaBoStream &s) const  
  write a variables object to the binary restart stream.

- void read (UnPackBuffer &s)  
  read a variables object from a packed MPI buffer.

- void write (PackBuffer &s) const  
  write a variables object to a packed MPI buffer.

Private Methods

- void copy_rep (const DakotaVariables *vars_rep)  
  Used by copy() to copy the contents of a letter class.

Private Attributes

- DakotaRealVector allContinuousVars  
  a continuous array combining continuous design, uncertain, and continuous state variable types (all).

- DakotaIntVector allDiscreteVars  
  a discrete array combining discrete design and discrete state variable types (all).

- DakotaStringArray allContinuousLabels  
  a label array combining continuous design, uncertain, and continuous state variable types (all).

- DakotaStringArray allDiscreteLabels  
  a label array combining discrete design and discrete state variable types (all).

- size_t numCDV  
  number of continuous design variables.

- size_t numDDV  
  number of discrete design variables.

- size_t numUV  
  number of uncertain variables.

- size_t numCSV  
  number of continuous state variables.

- size_t numDSV  
  number of discrete state variables.
Friends

- bool operator== (const AllVariables &vars1, const AllVariables &vars2)

  equality operator.

6.4.1 Detailed Description

Derived class within the DakotaVariables hierarchy which employs the all data view.

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllVariables derived class combines design, uncertain, and state variable types but separates continuous and discrete domain types. The result is a single array of continuous variables (allContinuousVars) and a single array of discrete variables (allDiscreteVars). Parameter and DACE studies currently use this approach (see DakotaVariables::get_variables(problem_db)).

6.4.2 Constructor & Destructor Documentation

6.4.2.1 AllVariables::AllVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels and combine them into allContinuousVars, allDiscreteVars, allContinuousLabels, and allDiscreteLabels using utilities from VariablesUtil.

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

- AllVariables.H
- AllVariables.C
6.5 AnalysisCode Class Reference

Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations.

Inheritance diagram for AnalysisCode::

```
AnalysisCode
   
ForkAnalysisCode
   
SysCallAnalysisCode
```

Public Methods

- void define_filenames (const int id)
  define modified filenames from user input by handling Unix temp file and tagging options.

- void write_parameters_file (const DakotaVariables &vars, const DakotaIntArray &asv, const int id)
  write the variables and active set vector objects to the parameters file in either standard or aprepro format.

- void read_results_file (DakotaResponse &response, const int id)
  read the response object from the results file.

- const DakotaStringList & program_names () const
  return programNames.

- const DakotaString & input_filter_name () const
  return iFilterName.

- const DakotaString & output_filter_name () const
  return oFilterName.

- const DakotaString & modified_parameters_filename () const
  return modifiedParamsFileName.

- const DakotaString & modified_results_filename () const
  return modifiedResFileName.

- const DakotaString & results_fname (const int id) const
  return the entry in resultsFNameList corresponding to id.

- void suppress_output_flag (const bool flag)
  set suppressOutputFlag.

- bool suppress_output_flag () const
  return suppressOutputFlag.
Protected Methods

- **AnalysisCode** (const ProblemDescDB &problem_db)
  
  *constructor.*

- virtual ~AnalysisCode ()
  
  *destructor.*

Protected Attributes

- **bool suppressOutputFlag**
  
  *flag set by master processor to suppress output from slave processors.*

- **bool verboseFlag**
  
  *flag for additional analysis code output if method verbosity is set.*

- **bool fileTagFlag**
  
  *flags tagging of parameter/results files.*

- **bool fileSaveFlag**
  
  *flags retention of parameter/results files.*

- **bool apreproFlag**
  
  *flags use of the APREPRO (the Sandia “A PRE PROcessor” utility) format for parameter files.*

- **DakotaString iFilterName**
  
  *the name of the input filter (input_file filter user specification).*

- **DakotaString oFilterName**
  
  *the name of the output filter (output_file filter user specification).*

- **DakotaStringList programNames**
  
  *the names of the analysis code programs (analysis_drivers user specification).*

- **size_t numPrograms**
  
  *the number of analysis code programs (length of programNames list).*

- **DakotaString parametersFileName**
  
  *the name of the parameters file from user specification.*

- **DakotaString modifiedParamsFileName**
  
  *the parameters file name actually used (modified with tagging or temp files).*

- **DakotaString resultsFileName**
  
  *the name of the results file from user specification.*

- **DakotaString modifiedResFileName**
  
  *the results file name actually used (modified with tagging or temp files).*
6.5 AnalysisCode Class Reference

- DakotaStringList parametersFNameList
  
  list of parameters file names used in spawning function evaluations.

- DakotaStringList resultsFNameList
  
  list of results file names used in spawning function evaluations.

- DakotaIntList fileNameKey
  
  stores function evaluation identifiers to allow key-based retrieval of file names from parametersFNameList and resultsFNameList.

Private Attributes

- ParallelLibrary & parallelLib
  
  reference to the ParallelLibrary object. Used in define_filenames().

6.5.1 Detailed Description

Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations.

The AnalysisCode class hierarchy provides simulation spawning services for ApplicationInterface derived classes and alleviates these classes of some of the specifics of simulation code management. The hierarchy does not employ the letter-envelope technique since the ApplicationInterface derived classes instantiate the appropriate derived AnalysisCode class directly.

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

- AnalysisCode.H
- AnalysisCode.C
6.6 ANNSurf Class Reference

Derived approximation class for artificial neural networks.

Inheritance diagram for ANNSurf:

```
DakotaApproximation
  ANNSurf
```

Public Methods

- **ANNSurf** (const ProblemDescDB &problemDb, const size_t &numAcv)
  
  constructor.

- **~ANNSurf** ()
  
  destructor.

Protected Methods

- int **required_samples** ()
  
  return the minimum number of samples required to build the derived class approximation type in numVars dimensions.

- void **find_coefficients** ()
  
  calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

- Real **get_value** (const DakotaRealVector &x)
  
  retrieve the approximate function value for a given parameter vector.

Private Attributes

- ANNAprox * **annObject**
  
  pointer to the ANNAprox object (see VendorPackages/ann for class declaration).

6.6.1 Detailed Description

Derived approximation class for artificial neural networks.

The ANNSurf class uses a layered-perceptron artificial neural network. Unlike most neural networks, it does not employ a back-propagation approach to training. Rather it uses a direct training approach
developed by Prof. David Zimmerman of the University of Houston and modified by Tom Paez and Chris O’Gorman of Sandia. It is more computationally efficient that back-propagation networks, but relative accuracy can be a concern.

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

- ANNSurf.H
- ANNSurf.C
6.7 ApplicationInterface Class Reference

Derived class within the interface class hierarchy for supporting interfaces to simulation codes.

Inheritance diagram for ApplicationInterface:

```
  DakotaInterface
     ^
     |  ApplicationInterface
     |  \\|-- DirectFnApplicInterface
     |  |-- ForkApplicInterface
     |  |-- SysCallApplicInterface
```

Protected Methods

- **ApplicationInterface** (const ProblemDescDB &problem_db, const size_t &num_fns)
  
  `constructor.`

- **~ApplicationInterface ()**
  
  `destructor.`

- void `init_communicators` (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  
  `allocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.`

- void `free_communicators ()`
  
  `deallocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.`

- void `init_serial ()`

- int `asynch_local_evaluation_concurrency ()` const
  
  `return asynchLocalEvalConcurrency.`

- DakotaString `interface_synchronization ()` const
  
  `return interfaceSynchronization.`

- void `map` (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, const bool asynch_flag=0)
  
  `Provides a "mapping" of variables to responses using a simulation. Protected due to DakotaInterface letter-envelope idiom.`

- void `manage_failure` (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int failed_eval_id)
  
  `manages a simulation failure using abort/retry/recover/continuation.`
• const DakotaResponseArray & synch ()
  executes a blocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns all jobs.

• const DakotaResponseList & synch_nowait ()
  executes a nonblocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns a partial list of completed jobs.

• void serve_evaluations ()
  run on evaluation servers to serve the iterator master.

• void stop_evaluation_servers ()
  used by the iterator master to terminate evaluation servers.

• virtual void derived_map (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int fn_eval_id)=0
  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

• virtual void derived_map_asynch (const ParamResponsePair &pair)=0
  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

• virtual void derived_synch (DakotaPRPList &prp_list)=0
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

• virtual void derived_synch_nowait (DakotaPRPList &prp_list)=0
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

• virtual void clear_bookkeeping ()
  clears any bookkeeping in derived classes.

• void self_schedule_analyses ()
  blocking self-schedule of all analyses within a function evaluation using message passing.

• void serve_analyses_synch ()
  serve the master analysis scheduler and manage one synchronous analysis job at a time.

• virtual int derived_synchronous_local_analysis (const int &analysis_id)=0
  Execute a particular analysis (identified by analysis_id) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve_analyses_synch().

Protected Attributes

• ParallelLibrary & parallelLib
reference to the ParallelLibrary object used to manage MPI partitions for the concurrent evaluations and concurrent analyses parallelism levels.

- `bool evalMessagePass`
  flags use of message passing at the level of evaluation scheduling.

- `bool analysisMessagePass`
  flags use of message passing at the level of analysis scheduling.

- `bool suppressOutput`
  flag for suppressing output on slave processors.

- `int asynchLocalAnalysisConcurrency`
  limits the number of concurrent analyses in asynchronous local scheduling and specifies hybrid concurrency when message passing.

- `bool asynchLocalAnalysisFlag`
  flag for asynchronous local parallelism of analyses.

- `int worldSize`
  size of MPI_COMM_WORLD.

- `int iteratorCommSize`
  size of iteratorComm.

- `int evalCommSize`
  size of evalComm.

- `int analysisCommSize`
  size of analysisComm.

- `int worldRank`
  processor rank within MPI_COMM_WORLD.

- `int iteratorCommRank`
  processor rank within iteratorComm.

- `int evalCommRank`
  processor rank within evalComm.

- `int analysisCommRank`
  processor rank within analysisComm.

- `int evalServerId`
  evaluation server identifier.

- `int analysisServerId`
  analysis server identifier.

- `bool evalDedMasterFlag`
flag for dedicated master partitioning at the level of evaluation scheduling.

- **bool multiProcAnalysisFlag**
  flag for multiprocessor analysis partitions.

- **DakotaStringList analysis Drivers**
  the set of analyses within each function evaluation (from the analysis_drivers interface specification).

- **int numAnalysisDrivers**
  length of analysisDrivers list.

- **int numAnalysisServers**
  number of analysis servers.

- **MPI_Comm evalComm**
  intracomm for fn eval; partition of iteratorComm.

- **MPI_Comm analysisComm**
  intracomm for analysis; partition of evalComm.

- **MPI_Comm evalAnalysisIntraComm**
  intracomm for all analysisCommRank==0 within evalComm.

- **int lenVarsMessage**
  length of a PackBuffer containing a DakotaVariables object; computed in DakotaModel::init_communicators().

- **int lenVarsASVMessage**
  length of a PackBuffer containing a DakotaVariables object and an active set vector object; computed in DakotaModel::init_communicators().

- **int lenResponseMessage**
  length of a PackBuffer containing a DakotaResponse object; computed in DakotaModel::init_communicators().

- **int lenPRPairMessage**
  length of a PackBuffer containing a ParamResponsePair object; computed in DakotaModel::init_communicators().

**Private Methods**

- **bool duplication detect (const DakotaVariables &vars, DakotaResponse &response, const bool asynch_flag)**
  checks data pairs and beforeSynchPRPList to see if the current evaluation request has already been performed or queued.

- **void self_schedule_evaluations ()**
  blocking self-schedule of all evaluations in beforeSynchPRPList using message passing; executes on iteratorComm master.
• void static_schedule_evaluations()  
  blocking static schedule of all evaluations in beforeSynchPRPList using message passing; executes on iteratorComm master.

• void asynchronous_local_evaluations(DakotaPRPList &prp_list)  
  perform all jobs in prp_list using asynchronous approaches on the local processor.

• void synchronous_local_evaluations(DakotaPRPList &prp_list)  
  perform all jobs in prp_list using synchronous approaches on the local processor.

• void asynchronous_local_evaluations_nowait(DakotaPRPList &prp_list)  
  launch new jobs in prp_list asynchronously (if capacity is available), perform nonblocking query of all running jobs, and process any completed jobs.

• void serve_evaluations_synch()  
  serve the evaluation message passing schedulers and perform one synchronous evaluation at a time.

• void serve_evaluations_asynch()  
  serve the evaluation message passing schedulers and manage multiple asynchronous evaluations.

• void serve_evaluations_peer()  
  serve the evaluation message passing schedulers and perform one synchronous evaluation at a time as part of the 1st peer.

• const ParamResponsePair & get_source_pair(const DakotaVariables &target_vars)  
  convenience function for the continuation approach in manage_failure() for finding the nearest successful "source" evaluation to the failed "target".

• void continuation(const DakotaVariables &target_vars, const DakotaIntArray &asv, DakotaResponse &response, const ParamResponsePair &source_pair, int failed_eval_id)  
  performs a 0th order continuation method to step from a successful "source" evaluation to the failed "target". Invoked by manage_failure() for failAction == "continuation".

Private Attributes

• int numEvalServers  
  number of evaluation servers.

• int procsPerAnalysis  
  processors per analysis servers.

• DakotaString evalScheduling  
  user specification of evaluation scheduling algorithm (self, static, or no spec). Used for manual overrides of the auto-configure logic in ParallelLibrary::resolve_inputs().

• DakotaString analysisScheduling  
  user specification of analysis scheduling algorithm (self, static, or no spec). Used for manual overrides of the auto-configure logic in ParallelLibrary::resolve_inputs().

• int asynchronousLocalEvalConcurrency
limits the number of concurrent evaluations in asynchronous local scheduling and specifies hybrid concurrency when message passing.

- **DakotaString interfaceSynchronization**
  
  interface synchronization specification: synchronous (default) or asynchronous.

- **bool headerFlag**
  
  used by synch_nowait to manage output frequency (since this function may be called many times prior to any completions).

- **bool asvControlFlag**
  
  used to manage a user request to deactivate the active set vector control. true = modify the ASV each evaluation as appropriate (default); false = ASV values are static so that the user need not check them on each evaluation.

- **bool evalCacheFlag**
  
  used to manage a user request to deactivate the function evaluation cache (i.e., queries and insertions using the data_pairs list).

- **bool restartFileFlag**
  
  used to manage a user request to deactivate the restart file (i.e., insertions into write_restart).

- **DakotaIntArray defaultASV**
  
  the static ASV values used when the user has selected asvControl = off.

- **DakotaString failAction**
  
  mitigation action for captured simulation failures: abort, retry, recover, or continuation.

- **int failRetryLimit**
  
  limit on the number of retries for the retry failAction.

- **DakotaRealVector failRecoveryFnVals**
  
  the dummy function values used for the recover failAction.

- **DakotaIntList historyDuplicateIds**
  
  used to bookkeep fnEvalId of asynchronous evaluations which duplicate data_pairs evaluations.

- **DakotaResponseList historyDuplicateResponses**
  
  used to bookkeep response of asynchronous evaluations which duplicate data_pairs evaluations.

- **DakotaIntList beforeSynchDuplicateIds**
  
  used to bookkeep fnEvalId of asynchronous evaluations which duplicate queued beforeSynchPRPList evaluations.

- **DakotaSizeList beforeSynchDuplicateIndices**
  
  used to bookkeep beforeSynchPRPList index of asynchronous evaluations which duplicate queued before-SynchPRPList evaluations.

- **DakotaResponseList beforeSynchDuplicateResponses**
  
  used to bookkeep response of asynchronous evaluations which duplicate queued beforeSynchPRPList evaluations.
- DakotaIntList runningList
  used by asynchronous_local_nowait to bookkeep which jobs are running.

- DakotaPRPList beforeSynchPRPList
  used to bookkeep vars/asv/response of nonduplicate asynchronous evaluations. This is the queue of jobs populated by asynchronous map() invocations which is later scheduled on a call to synch() or synch_nowait().

### 6.7.1 Detailed Description

Derived class within the interface class hierarchy for supporting interfaces to simulation codes.

ApplicationInterface provides an interface class for performing parameter to response mappings using simulation code(s). It provides common functionality for a number of derived classes and contains the majority of all of the scheduling algorithms in DAKOTA. The derived classes provide the specifics for managing code invocations using system calls, forks, direct procedure calls, or distributed resource facilities.

### 6.7.2 Member Function Documentation

#### 6.7.2.1 void ApplicationInterface::init_serial () [protected, virtual]

DataInterface.C defaults of 0 servers are needed to distinguish an explicit user request for 1 server (serialization of a parallelism level) from no user request (use parallel auto-config). This default causes problems when init_communicators() is not called for an interface object (e.g., static scheduling fails in DirectFnApplicInterface::derived_map() for NestedModel::optionalInterface). This is the reason for this function: to reset certain defaults for interface objects that are used serially.

Reimplemented from DakotaInterface.

#### 6.7.2.2 void ApplicationInterface::map (const DakotaVariables & vars, const DakotaIntArray & asv, DakotaResponse & response, const bool asynch_flag = 0) [protected, virtual]

Provides a “mapping” of variables to responses using a simulation. Protected due to DakotaInterface letter-envelope idiom.

The function evaluator for application interfaces. Called from derived_compute_response() and derived_async_compute_response() in derived DakotaModel classes. If asynch_flag is not set, perform a blocking evaluation (using derived_map()). If asynch_flag is set, add the job to the beforeSynchPRPList queue for execution by one of the scheduler routines in synch() or synch_nowait(). Duplicate function evaluations are detected with duplication_detect().

Reimplemented from DakotaInterface.

#### 6.7.2.3 const DakotaResponseArray & ApplicationInterface::synch () [protected, virtual]

executes a blocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns all jobs.
This function provides blocking synchronization for all cases of asynchronous evaluations, including the local asynchronous case (background system call, nonblocking fork, & multithreads), the message passing case, and the hybrid case. Called from derived synchronize() in derived DakotaModel classes.

Reimplemented from DakotaInterface.

### 6.7.2.4 const DakotaResponseList & ApplicationInterface::synch_nowait () [protected, virtual]

executes a nonblocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns a partial list of completed jobs.

This function will eventually provide nonblocking synchronization for all cases of asynchronous evaluations, however it currently supports only the local asynchronous case since nonblocking message passing schedulers have not yet been implemented. Called from derived synchronize_nowait() in derived DakotaModel classes.

Reimplemented from DakotaInterface.

### 6.7.2.5 void ApplicationInterface::serve_evaluations () [protected, virtual]

run on evaluation servers to serve the iterator master.

Invoked by the serve() function in derived DakotaModel classes. Passes control to serve_evaluations_asynch(), serve_evaluations_peer(), or serve_evaluations_synch() according to specified concurrency and self/static scheduler configuration.

Reimplemented from DakotaInterface.

### 6.7.2.6 void ApplicationInterface::stop_evaluation_servers () [protected, virtual]

used by the iterator master to terminate evaluation servers.

This code is executed on the iteratorComm rank 0 processor when iteration on a particular model is complete. It sends a termination signal (tag = 0 instead of a valid fn_eval_id) to each of the slave analysis servers. NOTE: This function is called from the Strategy layer even when in serial mode. Therefore, use both USE_MPI and iteratorCommSize to provide appropriate fall through behavior.

Reimplemented from DakotaInterface.

### 6.7.2.7 void ApplicationInterface::self_schedule_analyses () [protected]

blocking self-schedule of all analyses within a function evaluation using message passing.

This code is called from derived classes to provide the master portion of a master-slave algorithm for the dynamic self-scheduling of analyses among slave servers. It is patterned after self_schedule_evaluations(). It performs no analyses locally and matches either serve_analyses_synch() or serve_analyses_asynch() on the slave servers, depending on the value of asynchLocalAnalysisConcurrency. Self-scheduling approach assigns jobs in 2 passes. The 1st pass gives each server the same number of jobs (equal to asynchLocalAnalysisConcurrency). The 2nd pass assigns the remaining jobs to slave servers as previous jobs are completed. Single- and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within ParallelLibrary.
6.7.2.8 void ApplicationInterface::serve_analyses_synch () [protected]

serve the master analysis scheduler and manage one synchronous analysis job at a time.
This code is called from derived classes to run synchronous analyses on slave processors. The slaves
receive requests (blocking receive), do local derived_map_ac’s, and return codes. This is done continuously
until a termination signal is received from the master. It is patterned after serve_evaluations_synch().

6.7.2.9 bool ApplicationInterface::duplication_detect (const DakotaVariables & vars,
DakotaResponse & response, const bool asynch_flag) [private]

checks data_pairs and beforeSynchPRPList to see if the current evaluation request has already been per-
formed or queued.
Check incoming evaluation request for duplication with content of data_pairs and beforeSynchPRPList.
If duplication is detected, return true, else return false. Manage bookkeeping with historyDuplicate and
beforeSynchDuplicate lists. Called from map(). Note that the list searches can get very expensive if a
long list is searched on every new function evaluation (either from a large number of previous jobs, a large
number of pending jobs, or both). For this reason, a user request for deactivation of the evaluation cache
results in a complete bypass of duplication_detect(), even though a beforeSynchPRPList search would still
be meaningful. Since the intent of this request is to streamline operations, both list searches are bypassed.

6.7.2.10 void ApplicationInterface::self_schedule_evaluations () [private]

blocking self-schedule of all evaluations in beforeSynchPRPList using message passing; executes on
iteratorComm master.
This code is called from synch() to provide the master portion of a master-slave algorithm for the dynamic
self-scheduling of evaluations among slave servers. It performs no evaluations locally and matches either
serve_evaluations_synch() or serve_evaluations_asynch() on the slave servers, depending on the value of
asynchLocalEvalConcurrency. Self-scheduling approach assigns jobs in 2 passes. The 1st pass gives each
server the same number of jobs (equal to asynchLocalEvalConcurrency). The 2nd pass assigns the re-
main ing jobs to slave servers as previous jobs are completed. Single- and multilevel parallel use intra- and
inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within ParallelLibrary.

6.7.2.11 void ApplicationInterface::static_schedule_evaluations () [private]

blocking static schedule of all evaluations in beforeSynchPRPList using message passing; executes on
iteratorComm master.
This code runs on the iteratorCommRank 0 processor (the iterator) and is called from synch() in order to
assign a static schedule. It matches serve_evaluations_peer() for any other processors within the 1st evalu-
ation partition and serve_evaluations_synch()/serve_evaluations_asynch() for all other evaluation partitions
(depending on asynchLocalEvalConcurrency). It performs function evaluations locally for its portion of the
static schedule using either asynchronous_local_evaluations() or synchronous_local_evaluations(). Single-
level and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific
syntax is encapsulated within ParallelLibrary. The iteratorCommRank 0 processor assigns the static sched-
ule since it is the only processor with access to beforeSynchPRPList (it runs the iterator and calls syn-
chronize). The alternate design of each peer selecting its own jobs using the modulus operator would be
applicable if execution of this function (and therefore the job list) were distributed.
6.7.2.12 void ApplicationInterface::asynchronous_local_evaluations (DakotaPRPList & prp_list) [private]

perform all jobs in prp_list using asynchronous approaches on the local processor.
This function provides blocking synchronization for the local async case (background system call, non-blocking fork, or threads). It can be called from synch() for a complete local scheduling of all asynchronous jobs or from static_schedule_evaluations() to perform a local portion of the total job set. It uses the derived_map_asynch() to initiate asynchronous evaluations and derived_synch() to capture completed jobs, and mirrors the self_schedule_evaluations() message passing scheduler as much as possible (derived_synch() is modeled after MPI_Waitsome()).

6.7.2.13 void ApplicationInterface::synchronous_local_evaluations (DakotaPRPList & prp_list) [private]

perform all jobs in prp_list using synchronous approaches on the local processor.
This function provides blocking synchronization for the local synchronous case (foreground system call, blocking fork, or procedure call from derived_map()). It is called from static_schedule_evaluations() to perform a local portion of the total job set.

6.7.2.14 void ApplicationInterface::asynchronous_local_evaluations_nowait (DakotaPRPList & prp_list) [private]

launch new jobs in prp_list asynchronously (if capacity is available), perform nonblocking query of all running jobs, and process any completed jobs.
This function provides nonblocking synchronization for the local async case (background system call, nonblocking fork, or threads). It is called from synch_nowait() and passed the complete set of all asynchronous jobs (beforeSynchPRPList). It uses derived_map_asynch() to initiate asynchronous evaluations and derived_synch_nowait() to capture completed jobs in nonblocking mode. It mirrors a nonblocking message passing scheduler as much as possible (derived_synch_nowait() modeled after MPI_Testsome()). The results of this function are rawResponseList and completionList. Since rawResponseList is in no particular order, completionList must be used as a key. It is assumed that the incoming prp_list contains only active and new jobs - i.e., all completed jobs are cleared by synch_nowait().

6.7.2.15 void ApplicationInterface::serve_evaluations_synch () [private]

serve the evaluation message passing schedulers and perform one synchronous evaluation at a time.
This code is invoked by serve_evaluations() to perform one synchronous job at a time on each slave/peer server. The servers receive requests (blocking receive), do local synchronous maps, and return results. This is done continuously until a termination signal is received from the master (sent via stop_evaluation_servers()).

6.7.2.16 void ApplicationInterface::serve_evaluations_asynch () [private]

serve the evaluation message passing schedulers and manage multiple asynchronous evaluations.
This code is invoked by serve_evaluations() to perform multiple asynchronous jobs on each slave/peer server. The servers test for any incoming jobs, launch any new jobs, process any completed jobs, and return any results. Each of these components is nonblocking, although the server loop continues until a termination signal is received from the master (sent via stop_evaluation_servers()). In the master-slave case,
the master maintains the correct number of jobs on each slave. In the static scheduling case, each server is responsible for limiting concurrency (since the entire static schedule is sent to the peers at start up).

6.7.2.17  void ApplicationInterface::serve_evaluations_peer () [private]

serve the evaluation message passing schedulers and perform one synchronous evaluation at a time as part of the 1st peer.

This code is invoked by serve_evaluations() to perform a synchronous evaluation in coordination with the iteratorCommRank 0 processor (the iterator) for static schedules. The bcast() matches either the bcast() in synchronous_local_evaluations(), which is invoked by static_schedule_evaluations()), or the bcast() in map().

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

- ApplicationInterface.H
- ApplicationInterface.C
6.8 ApproximationInterface Class Reference

Derived class within the interface class hierarchy for supporting approximations to simulation-based results.

Inheritance diagram for ApproximationInterface::

```
DakotaInterface
  ApproximationInterface
```

Public Methods

- `ApproximationInterface` (ProblemDescDB &problem_db, const size_t &num_acv, const size_t &num_fns)
  
  constructor.

- `~ApproximationInterface` ()
  
  destructor.

Protected Methods

- void `map` (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, const bool async_flag=0)
  
  the function evaluator: provides an approximate "mapping" from the variables to the responses using functionSurfaces.

- int `minimum_samples` () const
  
  returns minSamples.

- void `build_global_approximation` (DakotaIterator &dace_iterator, const DakotaRealVector &lower_bnds, const DakotaRealVector &upper_bnds)
  
  builds a global approximation for use as a surrogate.

- void `build_local_approximation` (DakotaModel &actual_model)
  
  builds a local approximation for use as a surrogate.

- void `update_approximation` (const DakotaRealVector &x_star, const DakotaResponse &response_star)
  
  updates an existing global approximation with new data.

- const DakotaRealVectorArray & `approximation_coefficients` ()
  
  retrieve the approximation coefficients from each DakotaApproximation within an ApproximationInterface.
- `const DakotaResponseArray & synch()`  
  recovers data from a series of asynchronous evaluations (blocking).

- `const DakotaResponseList & synch_nowait()`  
  recovers data from a series of asynchronous evaluations (nonblocking).

### Private Attributes

- **DakotaString daceMethodPointer**  
  string pointer to the dace iterator specified by the user in the global approximation specification.

- **DakotaString actualInterfacePointer**  
  string pointer to the actual interface specified by the user in the local/multipoint approximation specifications.

- **DakotaArray<DakotaApproximation> functionSurfaces**  
  list of approximations, one per response function.

- **DakotaRealVectorArray functionSurfaceCoeffs**  
  array of approximation coefficient vectors, one vector per response function.

- **DakotaString sampleReuse**  
  user selection of type of sample reuse for approximation builds: all, region, file, or none (default).

- **DakotaString sampleReuseFile**  
  file name for sampleReuse == "file".

- **bool graphicsFlag**  
  controls 3D graphics of approximation surfaces.

- **int minSamples**  
  the minimum number of samples over all functionSurfaces.

- **DakotaResponseList beforeSynchResponseList**  
  bookkeeping list to catalogue responses generated in map for use in synch() and synch_nowait(). This supports pseudo-asynchronous operations (approximate responses all always computed synchronously, but asynchronous virtual functions are supported through bookkeeping).

### 6.8.1 Detailed Description

Derived class within the interface class hierarchy for supporting approximations to simulation-based results.

ApproximationInterface provides an interface class for building a set of global/local/multipoint approximations and performing approximate function evaluations using them. It contains a list of DakotaApproximation objects, one for each response function.
6.8.2 Member Data Documentation

6.8.2.1 DakotaString ApproximationInterface::daceMethodPointer [private]

string pointer to the dace iterator specified by the user in the global approximation specification. This pointer is not used for building objects since this is managed in SurrLayeredModels. Its use in ApproximationInterface is currently limited to flagging dace contributions to data sets in build_global_approximation().

6.8.2.2 DakotaString ApproximationInterface::actualInterfacePointer [private]

string pointer to the actual interface specified by the user in the local/multipoint approximation specifications. This pointer is not used for building objects since this is managed in SurrLayeredModels. Its use in ApproximationInterface is currently limited to header output.

6.8.2.3 DakotaArray<DakotaApproximation> ApproximationInterface::functionSurfaces [private]

list of approximations, one per response function. This formulation allows the use of mixed approximations (i.e., different approximations used for different response functions), although the input specification is not currently general enough to support it.

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

- ApproximationInterface.H
- ApproximationInterface.C
6.9 BaseConstructor Struct Reference

Dummy struct for overloading letter-envelope constructors.

Public Methods

- **BaseConstructor** (int=0)
  
  C++ structs can have constructors.

6.9.1 Detailed Description

Dummy struct for overloading letter-envelope constructors.
BaseConstructor is used to overload the constructor for the base class portion of letter objects. It avoids infinite recursion (Coplien p.139) in the letter-envelope idiom by preventing the letter from instantiating another envelope. Putting this struct here (rather than in a header of a class that uses it) avoids problems with circular dependencies.

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

- ProblemDescDB.H
6.10 BranchBndStrategy Class Reference

Strategy for mixed integer nonlinear programming using the PICO parallel branch and bound engine.

Inheritance diagram for BranchBndStrategy::

```
DakotaStrategy
  BranchBndStrategy
```

Public Methods

- **BranchBndStrategy** (ProblemDescDB &problem_db)
  constructor.

- **~BranchBndStrategy** ()
  destructor.

- void **run_strategy** ()
  Performs the branch and bound strategy by executing selectedIterator on userDefinedModel multiple times in parallel for different variable bounds within the model.

Private Attributes

- **DakotaModel userDefinedModel**
  the model used by the iterator.

- **DakotaIterator selectedIterator**
  the iterator used by BranchBndStrategy.

- **int numIteratorServers**
  number of concurrent iterator partitions.

- **int numRootSamples**
  number of samples to perform at the root of the branching structure.

- **int numNodeSamples**
  number of samples to perform at each node of the branching structure.

- **MPI_Comm picoComm**
  MPI intracommunicator for PICO hub processors (strategy and iterator masters).

- **int picoCommRank**
processor rank in picoComm.

- int picoCommSize
  number of processors in picoComm.

- int argC
  dummy argument count passed to pico classes in init(), readAll(), and readAndBroadcast().

- char ** argV
  dummy argument vector passed to pico classes in init(), readAll(), and readAndBroadcast().

- DoubleVector picoLowerBnds
  global lower bounds for merged continuous & discrete design variables passed to PICO (copied from user-DefinedModel).

- DoubleVector picoUpperBnds
  global upper bounds for merged continuous & discrete design variables passed to PICO (copied from user-DefinedModel).

- IntVector picoListOfIntegers
  key to the discrete variables which have been relaxed and merged into the continuous variables and bounds arrays (indices in the combined arrays).

### 6.10.1 Detailed Description

Strategy for mixed integer nonlinear programming using the PICO parallel branch and bound engine.

This strategy combines the PICO branching engine with nonlinear programming optimizers from DAKOTA (e.g., DOT, NPSOL, OPT++) to solve mixed integer nonlinear programs. The discrete variables in the problem must support relaxation, i.e., they must be able to assume nonintegral values during the solution process. PICO selects solution "branches", each of which constrains the problem to lie within different variable bounds. The series of branches selected is designed to drive integer variables to their integral values. For each of the branches, a nonlinear DAKOTA optimizer is used to solve the optimization problem and return the solution to PICO. If this solution has all of the integer variables at integral values, then it provides an upper bound on the true solution. This bound can be used to prune other branches, since there is no need to further investigate a branch which does not yet have integral values for the integer variables and which has an objective function worse than the bound. In linear programs, the bounding and pruning processes are rigorous and will lead to the exact global optimum. In nonlinear problems, the bounding and pruning processes are heuristic, i.e. they will find local optima but the global optimum may be missed. PICO supports parallelism between "hubs," each of which drives a concurrent iterator partition in DAKOTA (and each of these iterator partitions may have lower levels of nested parallelism). This complexity is hidden from PICO through the use of picoComm, which contains the set of master iterator processors, one from each iterator partition. Thus, PICO can schedule jobs among single-processor hubs in its normal manner, unaware of the nested parallelism complexities that may occur within each nonlinear optimization.

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

- BranchBndStrategy.H
- BranchBndStrategy.C
6.11 COLINApplication Class Template Reference

Public Methods

- **COLINApplication** (DakotaModel &model, DakotaResponse(*multiobj_mod_ptr)(const DakotaResponse &))
  
  destructor.

- ~**COLINApplication** ()

- void **DoEval** (DomainT &point, ResponseT *response, bool synch_flag)
  
  launch a function evaluation either synchronously or asynchronously.

- void **synchronize** ()
  
  blocking retrieval of all pending jobs.

- void **next_eval** (int &id)
  
  nonblocking query and retrieval of a job if completed.

- void **dakota_asynch_flag** (const bool &asynch_flag)
  
  Try to terminate a function evaluation TODO: supported by DAKOTA?

Private Methods

- void **map_response** (ResponseT &colin_response, const DakotaResponse &dakota_response)

Private Attributes

- **DakotaModel** & userDefinedModel
  
  reference to the COLINOptimizer’s model passed in the constructor.

- DakotaIntArray **activeSetVector**
  
  copy/conversion of the COLIN request vector.

- bool **dakotaModelAsynchFlag**
  
  a flag for asynchronous DAKOTA evaluations.

- DakotaResponseList **dakotaResponseList**
  
  list of DAKOTA responses returned by synchronize_nowait().

- DakotaIntList **dakotaCompletionsList**
  
  list of DAKOTA completions returned by synchronize_nowait_completions().

- size_t **numObjFns**
  
  number of objective functions.
- size_t numNonlinCons
  number of nonlinear constraints.

- DakotaResponse(* multiobjModifyPtr )(const DakotaResponse &)
  function pointer to DakotaOptimizer::multi_objective_modify() for reducing multiple objective functions to a single function.

- int num_real_params
- int num_integer_params
- DakotaVariables dakota_vars

### 6.11.1 Detailed Description

**template<class DomainT, class ResponseT> class COLINApplication< DomainT, ResponseT >**

COLINApplication is a DAKOTA class that is derived from COLIN’s OptApplication hierarchy. It redefines a variety of virtual COLIN functions to use the corresponding DAKOTA functions. This is a more flexible algorithm library interfacing approach than can be obtained with the function pointer approaches used by NPSOLOptimizer and SNLLOptimizer.

### 6.11.2 Member Function Documentation

#### 6.11.2.1 template<class DomainT, class ResponseT> void COLINApplication< DomainT, ResponseT >::DoEval (DomainT & pt, ResponseT * prob_response, bool synch_flag)

launch a function evaluation either synchronously or asynchronously.

Converts the DomainT variables and request vector to DAKOTA variables and active set vector, performs a DAKOTA function evaluation with synchronization governed by synch_flag, and then copies the DakotaResponse data to the ResponseT response (synchronous) or bookkeeps the response object (asynchronous).

#### 6.11.2.2 template<class DomainT, class ResponseT> void COLINApplication< DomainT, ResponseT >::synchronize ()

blocking retrieval of all pending jobs.

Blocking synchronize of asynchronous DAKOTA jobs followed by conversion of the DakotaResponse objects to ResponseT response objects.

#### 6.11.2.3 template<class DomainT, class ResponseT> void COLINApplication< DomainT, ResponseT >::next_eval (int & id)

nonblocking query and retrieval of a job if completed.

Nonblocking job retrieval. Finds a completion (if available), populates the COLIN response, and sets id to the completed job’s id. Else set id = -1.
6.11.2.4 template<class DomainT, class ResponseT> void COLINApplication< DomainT,
ResponseT >::dakota_asynch_flag (const bool & asynch_flag) [inline]

Try to terminate a function evaluation TODO: supported by DAKOTA?
This function is needed to publish the iterator’s asynchFlag at run time (asynchFlag not available at construction).

6.11.2.5 template<class DomainT, class ResponseT> void COLINApplication< DomainT,
ResponseT >::map_response (ResponseT & colin_response, const DakotaResponse &
dakota_response) [private]

map_response Maps a DakotaResponse object into a ResponseT class that is compatible with COLIN.
The documentation for this class was generated from the following file:

- COLINApplication.H
6.12 COLINO Optimizer Class Template Reference

Wrapper class for optimizers defined using COLIN.

Inheritance diagram for COLINOOptimizer:

```
DakotaOptimizer
    |________________________|
    |                         |
    | DakotatIterator         |
    |                         |
    | DakotOptLeastSq         |
    |                         |
    | DakotOptimizer          |
    |                         |
    | COLINOOptimizer         |
```

Public Methods

- **COLINO Optimizer** (DakotaModel &model)  
  *constructor.*

- **~COLINO Optimizer** ()  
  *destructor.*

- **void find optimum** ()  
  *Performs the iterations to determine the optimal solution.*

Protected Methods

- **virtual void set_rng** ()
- **virtual void set_initial_point** (ColinPoint &pt)
- **virtual void get_min_point** (ColinPoint &pt)
- **virtual void set_method_options** ()  
  *sets options for the methods based on user specifications.*

- **void set_standard_method_options** ()

Protected Attributes

- **OptimizerT * optimizer**  
  *Pointer to COLIN base optimizer object.*

- **OptProblem< ColinPoint > problem**  
  *pointer to COLIN problem object.*
6.12 COLINOptimizer Class Template Reference

- RNG * rng
  RNG ptr.

6.12.1 Detailed Description

template<class OptimizerT> class COLINOptimizer< OptimizerT >

Wrapper class for optimizers defined using COLIN.
The COLINOptimizer class provides a templated wrapper for COLIN, a Sandia-developed C++ optimization interface library. A variety of COLIN optimizers are defined in the COLINY optimization library, which contains the optimization components from the old SGOPT library. COLINY contains optimizers such as genetic algorithms, pattern search methods, and other nongradient-based techniques. COLINOptimizer uses a COLINApplication object to perform the function evaluations.

The user input mappings are as follows: max_iterations, max_function_evaluations, convergence_tolerance, solution_accuracy and max_cpu_time are mapped into COLIN’s max_iters, max_neval, ftol, accuracy, and max_time data attributes. An output setting of verbose is passed to COLIN’s set_output() function and a setting of debug activates output of method initialization and sets the COLIN debug attribute to 10000. COLIN methods assume asynchronous operations whenever the algorithm has independent evaluations which can be performed simultaneously (implicit parallelism). Therefore, parallel configuration is not mapped into the method, rather it is used in COLINApplication to control whether or not an asynchronous evaluation request from the method is honored by the model (exception: pattern search exploratory moves is set to best_all for parallel function evaluations). Refer to [Hart, W.E., 1997] for additional information on COLIN objects and controls.

6.12.2 Member Function Documentation

6.12.2.1 template<class OptimizerT> void COLINOptimizer< OptimizerT >::find_optimum ()  
  [virtual]

Performs the iterations to determine the optimal solution.
find_optimum redefines the DakotaOptimizer virtual function to perform the optimization using COLIN. It first sets up the problem data, then executes minimize() on the COLIN optimizer, and finally catalogues the results.
Implements DakotaOptimizer.

6.12.2.2 template<class OptimizerT> void COLINOptimizer< OptimizerT >::set_standard_method_options ()  
  [protected]

set_standard_method_options propagates standard DAKOTA user input to the optimizer.
The documentation for this class was generated from the following file:
- COLINOptimizer.H

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (c) 1997-2002
6.13 ColinPoint Class Reference

Public Attributes

- \texttt{vector<double> rvec}
- \texttt{vector<int> ivec}

6.13.1 Detailed Description

A class containing a vector of doubles and integers.

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

- \texttt{COLINOptimizer.H}
6.14 CommandLineHandler Class Reference

Utility class for managing command line inputs to DAKOTA.

Inheritance diagram for CommandLineHandler:

```
  CommandLineHandler
    GetLongOpt
    CommandLineHandler
```

Public Methods

- **CommandLineHandler ()**
  Constructor.

- **~CommandLineHandler ()**
  Destructor.

- **void check_usage (int argc, char **argv)**
  Verifies that DAKOTA is called with the correct command usage. Prints a descriptive message and exits the program if incorrect.

- **int read_restart_evals () const**
  Returns the number of evaluations to be read from the restart file (as specified on the DAKOTA command line) as an integer instead of a const char*.

6.14.1 Detailed Description

Utility class for managing command line inputs to DAKOTA.

CommandLineHandler provides additional functionality that is specific to DAKOTA's needs for the definition and parsing of command line options. Inheritance is used to allow the class to have all the functionality of the base class, GetLongOpt.

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

- CommandLineHandler.H
- CommandLineHandler.C
6.15 CommandShell Class Reference

Utility class which defines convenience operators for spawning processes with system calls.

Public Methods

- **CommandShell ()**
  
  *constructor.*

- **~CommandShell ()**
  
  *destructor.*

- **CommandShell & operator<< (const char *string)**
  
  *adds string to unixCommand.*

- **CommandShell & operator<< (CommandShell &(*f)(CommandShell &))**
  
  *allows passing of the flush function to the shell using <<.*

- **CommandShell & flush ()**
  
  *“flushes” the shell; i.e. executes the unixCommand.*

- **void asynch_flag (const bool flag)**
  
  *set the asynchFlag.*

- **bool asynch_flag () const**
  
  *get the asynchFlag.*

- **void suppress_output_flag (const bool flag)**
  
  *set the suppressOutputFlag.*

- **bool suppress_output_flag () const**
  
  *get the suppressOutputFlag.*

Private Attributes

- **DakotaString unixCommand**
  
  *the command string that is constructed through one or more << insertions and then executed by flush.*

- **bool asynchFlag**
  
  *flags nonblocking operation (background system calls).*

- **bool suppressOutputFlag**
  
  *flags suppression of shell output (no command echo).*
6.15 CommandShell Class Reference

6.15.1 Detailed Description

Utility class which defines convenience operators for spawning processes with system calls. The CommandShell class wraps the C system() utility and defines convenience operators for building a command string and then passing it to the shell.

6.15.2 Member Function Documentation

6.15.2.1 CommandShell & CommandShell::flush()

“flushes” the shell; i.e. executes the unixCommand.

Executes the unixCommand by passing it to system(). Appends an ”&” if asynchFlag is set (background system call) and echos the unixCommand to Cout if suppressOutputFlag is not set.

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

- CommandShell.H
- CommandShell.C
6.16 ConcurrentStrategy Class Reference

Strategy for multi-start iteration or pareto set optimization.

Inheritance diagram for ConcurrentStrategy::

```
DakotaStrategy
     |^
     v
ConcurrentStrategy
```

Public Methods

- **ConcurrentStrategy** (ProblemDescDB &problem_db)
  
  constructor.

- **~ConcurrentStrategy** ()
  
  destructor.

- void **run_strategy** ()
  
  Performs the concurrent strategy by executing selectedIterator on userDefinedModel multiple times in parallel for different settings within the iterator or model.

Private Attributes

- **DakotaModel userDefinedModel**
  
  the model used by the iterator.

- **DakotaIterator selectedIterator**
  
  the iterator used by the concurrent strategy.

- int **numIteratorServers**
  
  number of concurrent iterator partitions.

- int **numIteratorJobs**
  
  total number of iterator executions to schedule over the servers.

- **DakotaRealVectorArray parameterSets**
  
  an array of parameter set vectors (either multistart variable sets or pareto multiobjective weighting sets) to be performed.

- bool **multiStartFlag**
  
  a flag for distinguishing multi-start from Pareto set.
6.16 ConcurrentStrategy Class Reference

- bool strategyDedicatedMasterFlag
  signals ded. master partitioning.

- int iteratorServerId
  identifier for an iterator server.

6.16.1 Detailed Description

Strategy for multi-start iteration or pareto set optimization.

This strategy maintains two concurrent iterator capabilities. First, a general capability for running an iterator multiple times from different starting points is provided (often used for multi-start optimization, but not restricted to optimization). Second, a simple capability for mapping the "pareto frontier" (the set of optimal solutions in mutiobjective formulations) is provided. This pareto set is mapped through running an optimizer multiple times for different sets of multiobjective weightings.

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

- ConcurrentStrategy.H
- ConcurrentStrategy.C
6.17 CONMINOptimizer Class Reference

Wrapper class for the CONMIN optimization library.

Inheritance diagram for CONMINOptimizer::

```
CONMINOptimizer
\|-- DakotaIterator
|   \|-- DakotaOptLeastSq
|     \|-- DakotaOptimizer
        \|-- CONMINOptimizer
```

Public Methods

- **CONMINOptimizer** (DakotaModel &model)
  constructor.

- **~CONMINOptimizer**()
  destructor.

- **void** findOptimum()
  Used within the optimizer branch for computing the optimal solution. Redefines the run iterator virtual function for the optimizer branch.

Private Methods

- **void** allocate_workspace()
  Allocates workspace for the optimizer.

Private Attributes

- **int** conminInfo
  INFO from CONMIN manual.

- **int** printControl
  IPRINT from CONMIN manual (controls output verbosity).

- **int** optimizationType
  MINMAX from DOT manual (minimize or maximize).
- **DakotaRealVector localConstraintValues**
  array of nonlinear constraint values passed to CONMIN.

- **DakotaSizetList constraintMappingIndices**
  a list of indices for referencing the corresponding DakotaResponse constraints used in computing the CONMIN constraints.

- **DakotaRealList constraintMappingMultipliers**
  a list of multipliers for mapping the DakotaResponse constraints to the CONMIN constraints.

- **DakotaRealList constraintMappingOffsets**
  a list of offsets for mapping the DakotaResponse constraints to the CONMIN constraints.

- **int N1**
  Size variable for CONMIN arrays. See CONMIN manual.

- **int N2**
  Size variable for CONMIN arrays. See CONMIN manual.

- **int N3**
  Size variable for CONMIN arrays. See CONMIN manual.

- **int N4**
  Size variable for CONMIN arrays. See CONMIN manual.

- **int N5**
  Size variable for CONMIN arrays. See CONMIN manual.

- **int NFDG**
  Finite difference flag.

- **int IPRINT**
  Flag to control amount of output data.

- **int ITMAX**
  Flag to specify the maximum number of iterations.

- **Real FDCH**
  Relative finite difference step size.

- **Real FDCHM**
  Absolute finite difference step size.

- **Real CT**
  Constraint thickness parameter.

- **Real CTMIN**
  Minimum absolute value of CT used during optimization.
- **Real CTL**  
  *Constraint thickness parameter for linear and side constraints.*

- **Real CTLMIN**  
  *Minimum value of CTL used during optimization.*

- **Real DELFUN**  
  *Relative convergence criterion threshold.*

- **Real DABFUN**  
  *Absolute convergence criterion threshold.*

- **Real * conminDesVars**  
  *Array of design variables used by CONMIN (length N1 = numdv+2).*

- **Real * conminLowerBnds**  
  *Array of lower bounds used by CONMIN (length N1 = numdv+2).*

- **Real * conminUpperBnds**  
  *Array of upper bounds used by CONMIN (length N1 = numdv+2).*

- **Real * S**  
  *Internal CONMIN array.*

- **Real * G1**  
  *Internal CONMIN array.*

- **Real * G2**  
  *Internal CONMIN array.*

- **Real * B**  
  *Internal CONMIN array.*

- **Real * C**  
  *Internal CONMIN array.*

- **int * MS1**  
  *Internal CONMIN array.*

- **Real * SCAL**  
  *Internal CONMIN array.*

- **Real * DF**  
  *Internal CONMIN array.*

- **Real * A**  
  *Internal CONMIN array.*

- **int * ISC**  
  *Internal CONMIN array.*
6.17 CONMINOptimizer Class Reference

- int * IC
  
  Internal CONMIN array.

6.17.1 Detailed Description

Wrapper class for the CONMIN optimization library.

The CONMINOptimizer class provides a wrapper for CONMIN, a Public-domain Fortran 77 optimization library written by Gary Vanderplaats under contract to NASA Ames Research Center. The CONMIN User’s Manual is contained in NASA Technical Memorandum X-62282, 1978. CONMIN uses a reverse communication mode, which avoids the static function and static attribute issues that arise with function pointer designs (see NPSOLOptimizer and SNLLOptimizer).

The user input mappings are as follows: max_iterations is mapped into CONMIN’s ITMAX parameter, max_function_evaluations is implemented directly in the find_optimum() loop since there is no CONMIN parameter equivalent, convergence_tolerance is mapped into CONMIN’s DELFUN and DABFUN parameters, output verbosity is mapped into CONMIN’s IPRINT parameter (verbose: IPRINT = 4; quiet: IPRINT = 2), gradient mode is mapped into CONMIN’s NFDG parameter, and finite difference step size is mapped into CONMIN’s FDCH and FDCHM parameters. Refer to [Vanderplaats, 1978] for additional information on CONMIN parameters.

6.17.2 Member Data Documentation

6.17.2.1 int CONMINOptimizer::conminInfo [private]

INFO from CONMIN manual.

Information requested by CONMIN: 1 = evaluate objective and constraints, 2 = evaluate gradients of objective and constraints.

6.17.2.2 int CONMINOptimizer::printControl [private]

IPRINT from CONMIN manual (controls output verbosity).

Values range from 0 (nothing) to 4 (most output). 0 = nothing, 1 = initial and final function information, 2 = all of #1 plus function value and design vars at each iteration, 3 = all of #2 plus constraint values and direction vectors, 4 = all of #3 plus gradients of the objective function and constraints, 5 = all of #4 plus proposed design vector, plus objective and constraint functions from the 1-D search.

6.17.2.3 int CONMINOptimizer::optimizationType [private]

MINMAX from DOT manual (minimize or maximize).

Values of 0 or -1 (minimize) or 1 (maximize).

6.17.2.4 DakotaRealVector CONMINOptimizer::localConstraintValues [private]

array of nonlinear constraint values passed to CONMIN.
This array must be of nonzero length (sized with localConstraintArraySize) and must contain only one-sided inequality constraints which are \( \leq 0 \) (which requires a transformation from 2-sided inequalities and equalities).

### 6.17.2.5 DakotaSizetList CONMINOptimizer::constraintMappingIndices [private]

A list of indices for referencing the corresponding DakotaResponse constraints used in computing the CONMIN constraints.

The length of the list corresponds to the number of CONMIN constraints, and each entry in the list points to the corresponding DAKOTA constraint.

### 6.17.2.6 DakotaRealList CONMINOptimizer::constraintMappingMultipliers [private]

A list of multipliers for mapping the DakotaResponse constraints to the CONMIN constraints.

The length of the list corresponds to the number of CONMIN constraints, and each entry in the list contains a multiplier for the DAKOTA constraint identified with constraintMappingIndices. These multipliers are currently +1 or -1.

### 6.17.2.7 DakotaRealList CONMINOptimizer::constraintMappingOffsets [private]

A list of offsets for mapping the DakotaResponse constraints to the CONMIN constraints.

The length of the list corresponds to the number of CONMIN constraints, and each entry in the list contains an offset for the DAKOTA constraint identified with constraintMappingIndices. These offsets involve inequality bounds or equality targets, since CONMIN assumes constraint allowables = 0.

### 6.17.2.8 int CONMINOptimizer::N1 [private]

Size variable for CONMIN arrays. See CONMIN manual.

\( N1 = \text{number of variables} + 2 \)

### 6.17.2.9 int CONMINOptimizer::N2 [private]

Size variable for CONMIN arrays. See CONMIN manual.

\( N2 = \text{number of constraints} + 2 \times (\text{number of variables}) \)

### 6.17.2.10 int CONMINOptimizer::N3 [private]

Size variable for CONMIN arrays. See CONMIN manual.

\( N3 = \text{Maximum possible number of active constraints} \)

### 6.17.2.11 int CONMINOptimizer::N4 [private]

Size variable for CONMIN arrays. See CONMIN manual.

\( N4 = \text{Maximum}(N3, \text{number of variables}) \)
6.17.2.12  int CONMINOptimizer::N5  [private]

Size variable for CONMIN arrays. See CONMIN manual.
N5 = 2*(N4)

6.17.2.13  Real CONMINOptimizer::CT  [private]

Constraint thickness parameter.
The value of CT decreases in magnitude during optimization.

6.17.2.14  Real* CONMINOptimizer::S  [private]

Internal CONMIN array.
Move direction in N-dimensional space.

6.17.2.15  Real* CONMINOptimizer::G1  [private]

Internal CONMIN array.
Temporary storage of constraint values.

6.17.2.16  Real* CONMINOptimizer::G2  [private]

Internal CONMIN array.
Temporary storage of constraint values.

6.17.2.17  Real* CONMINOptimizer::B  [private]

Internal CONMIN array.
Temporary storage for computations involving array S.

6.17.2.18  Real* CONMINOptimizer::C  [private]

Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.17.2.19  int* CONMINOptimizer::MS1  [private]

Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.17.2.20  Real* CONMINOptimizer::SCAL  [private]

Internal CONMIN array.
Vector of scaling parameters for design parameter values.
6.17.2.21 Real* CONMINOptimizer::DF [private]

Internal CONMIN array.
Temporary storage for analytic gradient data.

6.17.2.22 Real* CONMINOptimizer::A [private]

Internal CONMIN array.
Temporary 2-D array for storage of constraint gradients.

6.17.2.23 int* CONMINOptimizer::ISC [private]

Internal CONMIN array.
Array of flags to identify linear constraints. (not used in this implementation of CONMIN)

6.17.2.24 int* CONMINOptimizer::IC [private]

Internal CONMIN array.
Array of flags to identify active and violated constraints.

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

- CONMINOptimizer.H
- CONMINOptimizer.C
6.18  CtelRegexp Class Reference

Public Types

- enum RStatus
  
  ```
  GOOD = 0, EXP_TOO_BIG, OUT_OF_MEM, TOO_MANY_PAR, UNMATCH_PAR, STARPLUS_EMPTY, STARPLUS_NESTED, INDEX_RANGE, INDEXMATCH, STARPLUS NOTHING, TRAILING, INT_ERROR, BAD_PARAM, BAD_OPCODE
  ```

  *Error codes reported by the engine - Most of these codes never really occurs with this implementation.*

Public Methods

- **CtelRegexp** (const std::string &pattern)
  
  Constructor - compile a regular expression.

- **~CtelRegexp** ()
  
  Destructor.

- bool compile (const std::string &pattern)
  
  Compile a new regular expression.

- std::string match (const std::string &str)
  
  matches a particular string; this method returns a string that is a sub-string matching with the regular expression.

- bool match (const std::string &str, size_t *start, size_t *size)
  
  another form of matching; returns the indexes of the maching.

- RStatus getStatus ()
  
  Get status.

- const std::string & getStatusMsg ()
  
  Get status message.

- void clearErrors ()
  
  Clear all errors.

- const std::string & getRe ()
  
  Return regular expression pattern.

- bool split (const std::string &str, std::vector<std::string> &all_matches)
  
  Split.
Private Methods

- **CtelRegexp** (const CtelRegexp &)
  
  Private copy constructor.

- **CtelRegexp & operator=** (const CtelRegexp &)
  
  Private assignment operator.

Private Attributes

- std::string **strPattern**
  
  STL string to hold pattern.

- regexp * **r**
  
  Pointer to regexp.

- **RStatus status**
  
  Return status, enumerated type.

- std::string **statusMsg**
  
  STL string to hold status message.

6.18.1 Detailed Description

DESCRIPTION: Wrapper for the Regular Expression engine( regexp ) released by Henry Spencer of the University of Toronto.

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

- **CtelReg Exp.H**
- **CtelReg Exp.C**
6.19 DACEIterator Class Reference

Wrapper class for the DDACE design of experiments library.

Inheritance diagram for DACEIterator::

```
DACEIterator
    ^
    |  
DakotaIterator
```

Public Methods

- **DACEIterator (DakotaModel &model)**
  primary constructor for building a standard iterator.

- **DACEIterator (DakotaModel &model, int samples, int symbols, int seed, const DakotaString &sampling_method)**
  alternate constructor for an iterator used for building approximations (inactive).

- **~DACEIterator ()**
  destructor.

- **void run_iterator ()**
  run the iterator.

- **const DakotaVariables & iterator_variable_results () const**
  return the final iterator solution (variables).

- **const DakotaResponse & iterator_response_results () const**
  return the final iterator solution (response).

- **void print_iterator_results (ostream &s) const**
  print the final iterator results.

- **void sampling_reset (int min_samples, bool all_data_flag, bool stats_flag)**
  reset sampling iterator.

- **const DakotaString & sampling_scheme () const**
  return sampling name.

- **void update_best (const DakotaRealVector &vars, const DakotaResponse &response, const int eval_num)**
  compares current evaluation to best evaluation and updates best.
Private Methods

- void resolve_samples_symbols ()
  
  convenience function for resolving number of samples and number of symbols from input.

Private Attributes

- DakotaString daceMethod
  
  oas, lhs, oa_lhs, random, box, behnken, central, composite, or grid.

- int numSamples
  
  number of samples to be evaluated.

- int numSymbols
  
  number of symbols to be used in generating the sample set (inversely related to number of replications).

- const int originalSeed
  
  the user seed specification for the random number generator (allows repeatable results).

- int randomSeed
  
  current seed for the random number generator.

- bool allDataFlag
  
  flag which triggers the update of allVars/allResponses for use by DakotaIterator::all_variables() and DakotaIterator::all_responses().

- size_t numDACERuns
  
  counter for number of executions of run_iterator() for this object.

- bool varyPattern
  
  flag for continuing the random number sequence from a previous run_iterator() execution (e.g., for surrogate-based optimization) so that multiple executions are repeatable but not correlated.

- DakotaVariables bestVariables
  
  best variables found during the study.

- DakotaResponse bestResponses
  
  best responses found during the study.

- Real bestObjectiveFn
  
  best objective function found during the study.

- Real bestViolations
  
  best constraint violations found during the study. In the current approach, constraint violation reduction takes strict precedence over objective function reduction.

- size_t numObjectiveFunctions
  
  number of objective functions. Used in update_best.
6.19 DACEIterator Class Reference

- size_t numNonlinearIneqConstraints
  number of nonlinear inequality constraints. Used in updateBest.

- size_t numNonlinearEqConstraints
  number of nonlinear equality constraints. Used in updateBest.

- DakotaRealVector multiObjWeights
  vector of multiobjective weights. Used in updateBest.

- DakotaRealVector nonlinearIneqLowerBnds
  vector of nonlinear inequality constraint lower bounds. Used in updateBest.

- DakotaRealVector nonlinearIneqUpperBnds
  vector of nonlinear inequality constraint upper bounds. Used in updateBest.

- DakotaRealVector nonlinearEqTargets
  vector of nonlinear equality constraint targets. Used in updateBest.

6.19.1 Detailed Description

Wrapper class for the DDACE design of experiments library.

The DACEIterator class provides a wrapper for DDACE, a C++ design of experiments library from the Computational Sciences and Mathematics Research (CSMR) department at Sandia’s Livermore CA site. This class uses design and analysis of computer experiments (DACE) methods to sample the design space spanned by the bounds of a DakotaModel. It returns all generated samples and their corresponding responses as well as the best sample found.

6.19.2 Constructor & Destructor Documentation

6.19.2.1 DACEIterator::DACEIterator (DakotaModel & model)

primary constructor for building a standard iterator.

This constructor is called for a standard iterator built with data from probDescDB.

6.19.2.2 DACEIterator::DACEIterator (DakotaModel & model, int samples, int symbols, int seed, const DakotaString & samplingMethod)

alternate constructor for an iterator used for building approximations (inactive).

This constructor is currently inactive, since the old DACEIterator instantiations within ApproximationInterface have been replaced with more general facilities within LayeredModel.

6.19.3 Member Function Documentation
6.19.3.1 void DACEIterator::run_iterator () [virtual]

run the iterator.

This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from DakotaIterator.

6.19.3.2 void DACEIterator::resolve_samples_symbols () [private]

convenience function for resolving number of samples and number of symbols from input.

This function must define a combination of samples and symbols that is acceptable for a particular sampling algorithm. Users provide requests for these quantities, but this function must enforce any restrictions imposed by the sampling algorithms.

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

- DACEIterator.H
- DACEIterator.C
6.20 DakotaApproximation Class Reference

Base class for the approximation class hierarchy.

Inheritance diagram for DakotaApproximation:

```
DakotaApproximation
    ANNSurf
    HermiteSurf
    KrigingSurf
    MARSSurf
    RespSurf
    TaylorSurf
```

Public Methods

- **DakotaApproximation ()**
  
  default constructor.

- **DakotaApproximation (const DakotaString &approx_type, const ProblemDescDB &problem_db, const size_t &num_acv)**
  
  standard constructor for envelope.

- **DakotaApproximation (const DakotaApproximation &approx)**
  
  copy constructor.

- virtual **~DakotaApproximation ()**
  
  destructor.

- DakotaApproximation **operator= (const DakotaApproximation &approx)**
  
  assignment operator.

- virtual **Real get_value (const DakotaRealVector &x)**
  
  retrieve the approximate function value for a given parameter vector.

- virtual **const DakotaRealVector & get_gradient (const DakotaRealVector &x)**
  
  retrieve the approximate function gradient for a given parameter vector.

- virtual **int required_samples ()**
  
  return the minimum number of samples required to build the derived class approximation type in numVars dimensions.

- virtual **const DakotaRealVector & approximation_coefficients ()**
  
  return the coefficient array computed by find_coefficients().

- void **build (const DakotaRealVectorArray &vars_samples, const DakotaRealVector &fn_samples, const DakotaRealVectorArray &grad_samples)**
  
  build the surface from scratch. Populates currentPoints and invokes find_coefficients().

- void **add_point_rebuild (const DakotaRealVector &x, const Real &f, const DakotaRealVector &grad_f)**
add a new point to the approximation and rebuild it.

- **set_bounds** (const DakotaRealVector &lower, const DakotaRealVector &upper)
  
  set approximation lower and upper bounds (currently only used by graphics).

- **draw_surface** ()
  
  render the approximate surface using the 3D graphics (2 variable problems only).

- **num_variables** () const
  
  return the number of variables used in the approximation.

**Protected Methods**

- **DakotaApproximation** (BaseConstructor, const ProblemDescDB &problem_db, const size_t &num_acv)
  
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

- **virtual void find_coefficients** ()
  
  calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

**Protected Attributes**

- **int numVars**
  
  number of variables in the approximation.

- **int numCurrentPoints**
  
  number of points in the currentPoints list.

- **int numSamples**
  
  number of samples passed to build() to construct the approximation.

- **bool gradientFlag**
  
  flag signaling the use of gradient data in global approximation builds as indicated by the user’s use_gradients specification.

- **bool verboseFlag**
  
  flag for verbose approximation output.

- **DakotaRealVector gradVector**
  
  gradient of the approximation with respect to the variables.

- **DakotaList< SurrogateDataPoint > currentPoints**
  
  list of samples used to build the approximation.

- **DakotaString approxType**
  
  approximation type (long form for diagnostic I/O).
Private Methods

- DakotaApproximation * get_approx (const DakotaString &approx_type, const ProblemDescDB &problem_db, const size_t &num_acv)
  
  Used only by the envelope constructor to initialize approxRep to the appropriate derived type.

- void add_point (const DakotaRealVector &x, const Real &f, const DakotaRealVector &grad_f)
  
  add a new point to the approximation (used by build & add_point rebuild).

Private Attributes

- DakotaRealVector approxLowerBounds
  
  approximation lower bounds (used only by 3D graphics).

- DakotaRealVector approxUpperBounds
  
  approximation upper bounds (used only by 3D graphics).

- DakotaApproximation * approxRep
  
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  
  number of objects sharing approxRep.

6.20.1 Detailed Description

Base class for the approximation class hierarchy.

The DakotaApproximation class is the base class for the data fit surrogate class hierarchy in DAKOTA. One instance of a DakotaApproximation must be created for each function to be approximated (a vector of DakotaApproximations is contained in ApproximationInterface). For memory efficiency and enhanced polymorphism, the approximation hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaApproximation) serves as the envelope and one of the derived classes (selected in DakotaApproximation::get_approximation()) serves as the letter.

6.20.2 Constructor & Destructor Documentation

6.20.2.1 DakotaApproximation::DakotaApproximation ()

default constructor.

The default constructor is used in List&lt;DakotaApproximation&gt; instantiations. approxRep is NULL in this case (problem_db is needed to build a meaningful DakotaModel object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.
6.20.2.2 DakotaApproximation::DakotaApproximation (const DakotaString & approx_type, const ProblemDescDB & problem_db, const size_t & num_acv)

standard constructor for envelope.
Envelope constructor only needs to extract enough data to properly execute get_approx, since DakotaApproximation(BaseConstructor, problem_db) builds the actual base class data for the derived approximations.

6.20.2.3 DakotaApproximation::DakotaApproximation (const DakotaApproximation & approx)

copy constructor.
Copy constructor manages sharing of approxRep and incrementing of referenceCount.

6.20.2.4 DakotaApproximation::~DakotaApproximation () [virtual]
destructor.
Destructor decrements referenceCount and only deletes approxRep when referenceCount reaches zero.

6.20.2.5 DakotaApproximation::DakotaApproximation (BaseConstructor, const ProblemDescDB & problem_db, const size_t & num_acv) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor is the one which must build the base class data for all derived classes. get_approx() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling get_approx() again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaApproximation).

6.20.3 Member Function Documentation

6.20.3.1 DakotaApproximation DakotaApproximation::operator= (const DakotaApproximation & approx)

assignment operator.

6.20.3.2 DakotaApproximation * DakotaApproximation::get_approx (const DakotaString & approx_type, const ProblemDescDB & problem_db, const size_t & num_acv) [private]

Used only by the envelope constructor to initialize approxRep to the appropriate derived type.
Used only by the envelope constructor to initialize approxRep to the appropriate derived type, as given by the approx_type parameter.
The documentation for this class was generated from the following files:

- DakotaApproximation.H
- DakotaApproximation.C
6.21 DakotaArray Class Template Reference

Template class for the Dakota bookkeeping array.

Public Methods

- DakotaArray ()
  Default constructor.

- DakotaArray (size_t size)
  Constructor which takes an initial size.

- DakotaArray (size_t size, const T &initial_val)
  Constructor which takes an initial size and an initial value.

- DakotaArray (const DakotaArray &a)
  Copy constructor.

- DakotaArray (const T *p, size_t size)
  Constructor, creates array of size, with initial value T p.

- ~DakotaArray ()
  Destructor.

- DakotaArray<T> & operator= (const DakotaArray<T> &a)
  Normal const assignment operator.

- DakotaArray<T> & operator= (DakotaArray<T> &a)
  Normal assignment operator.

- DakotaArray<T> & operator= (const T &ival)
  Sets all elements in self to the value ival.

- operator T * () const
  Converts the DakotaArray to a standard C-style array. Use with care!

- T & operator[] (int i)
  alternate bounds-checked indexing operator for int indices.

- const T & operator[] (int i) const
  alternate bounds-checked const indexing operator for int indices.

- T & operator[] (size_t i)
  Index operator, returns the ith value of the array.

- const T & operator[] (size_t i) const
Index operator const, returns the ith value of the array.

- T & operator(size_t i) (index operator, not bounds checked)

- const T & operator(size_t i) const (index operator const, not bounds checked)

- void print(ostream &s) const (prints a DakotaArray to an output stream)

- void read(UnPackBuffer &s) (reads a DakotaArray from a buffer after an MPI receive)

- void print(PackBuffer &s) const (writes a DakotaArray to a buffer prior to an MPI send)

- size_t length() const (returns size of array)

- void reshape(size_t sz) (resizes array to size sz)

- const T * data() const (returns pointer T* to continuous data)

- void testClass() (class unit test method)

### 6.21.1 Detailed Description

**template<class T> class DakotaArray<T>**

Template class for the Dakota bookkeeping array.

An array class template that provides additional functionality that is specific to Dakota's needs. The DakotaArray class adds additional functionality needed by Dakota to the inherited base array class. The DakotaArray class can inherit from either the STL or RW vector classes.

### 6.21.2 Constructor & Destructor Documentation

#### 6.21.2.1 template<class T> DakotaArray<T>::DakotaArray(const T * p, size_t size) [inline]

Constructor, creates array of size, with initial value <T> p.

Assigns size values from p into array.
6.21.3 Member Function Documentation

6.21.3.1 template<class T> DakotaArray<T> & DakotaArray<T>::operator= (const T & ival) [inline]

Sets all elements in self to the value ival.
Assigns all values of array to the value passed in as ival. For the Rogue Wave case utilizes base class operator=(ival),i while for the ANSI case uses the STL assign() method.

6.21.3.2 template<class T> DakotaArray<T>::operator T * () const [inline]

Converts the DakotaArray to a standard C-style array. Use with care!
The operator() returns a c style pointer to the data within the array. Calls the data() method. USE WITH CARE.

6.21.3.3 ]
template<class T> T & DakotaArray<T>::operator[] (size_t i) [inline]

Index operator, returns the ith value of the array.
Index operator; calls the STL method at() which is bounds checked. Mimics the RW vector class. Note: the at() method is not supported by the _GNUC__STL implementation and by SGI builds omitting exceptions (e.g., SIERRA).

6.21.3.4 ]
template<class T> const T & DakotaArray<T>::operator[] (size_t i) const [inline]

Index operator const, returns the ith value of the array.
A const version of the index operator; calls the STL method at() which is bounds checked. Mimics the RW vector class. Note: the at() method is not supported by the _GNUC__STL implementation and by SGI builds omitting exceptions (e.g., SIERRA).

6.21.3.5 template<class T> T & DakotaArray<T>::operator() (size_t i) [inline]

Index operator, not bounds checked.
Non bounds check index operator, calls the STL operator[] which is not bounds checked. Needed to mimic the RW vector class

6.21.3.6 template<class T> const T & DakotaArray<T>::operator() (size_t i) const [inline]

Index operator const, not bounds checked.
A const version of the non-bounds check index operator, calls the STL operator[] which is not bounds checked. Needed to mimic the RW vector class
6.21.3.7 \texttt{template<class T> const T * DakotaArray\textless T \textgreater \::data () const [inline]}

Returns pointer T* to continuous data.

Returns a C style pointer to the data within the array. USE WITH CARE. Needed to mimic RW vector class, is used in the operator(). Uses the STL front method.

6.21.3.8 \texttt{template<class T> void DakotaArray\textless T \textgreater \::testClass ()}

Class unit test method.

Unit test method for the DakotaArray class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an unexpected answer is received.

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

- DakotaArray.H
# 6.22 DakotaBaseVector Class Template Reference

Base class for the DakotaMatrix and DakotaVector classes.

Inheritance diagram for DakotaBaseVector:

```
DakotaBaseVector
    ▲
   │
DakotaVector
```

## Public Methods

- **DakotaBaseVector ()**
  
  *Default constructor.*

- **DakotaBaseVector (size_t size)**
  
  *Constructor, creates vector of size.*

- **DakotaBaseVector (size_t size, const T &initial_val)**
  
  *Constructor, creates vector of size with initial value of initial_val.*

- **~DakotaBaseVector ()**
  
  *Destructor.*

- **DakotaBaseVector (const DakotaBaseVector<T> &a)**
  
  *Copy constructor.*

- **DakotaBaseVector<T> & operator= (const DakotaBaseVector<T> &a)**
  
  *Normal assignment operator.*

- **DakotaBaseVector<T> & operator= (const T &ival)**
  
  *Assigns all values of vector to ival.*

- **T & operator[] (int i)**
  
  *alternate bounds-checked indexing operator for int indices.*

- **const T & operator[] (int i) const**
  
  *alternate bounds-checked const indexing operator for int indices.*

- **T & operator[] (size_t i)**
  
  *Returns the object at index i, (can use as lvalue).*

- **const T & operator[] (size_t i) const**
  
  *Returns the object at index i, const (can’t use as lvalue).*
6.22 DakotaBaseVector Class Template Reference

- **T & operator(size_t i)**
  
  Index operator, not bounds checked.

- **const T & operator(size_t i) const**
  
  Index operator const, not bounds checked.

- **size_t length() const**
  
  Returns size of vector.

- **void reshape(size_t sz)**
  
  Resizes vector to size sz.

- **const T * data() const**
  
  Returns const pointer to standard C array. Use with care.

Protected Methods

- **T * array() const**
  
  Returns pointer to standard C array. Use with care.

6.22.1 Detailed Description

template<class T> class DakotaBaseVector<T>

Base class for the DakotaMatrix and DakotaVector classes.

The DakotaBaseVector class is the base class for the DakotaMatrix class. It is used to define a common vector interface for both the STL and RW vector classes. If the STL version is based on the valarray class then some basic vector operations such as +, * are available.

6.22.2 Constructor & Destructor Documentation

6.22.2.1 template<class T> DakotaBaseVector<T>::DakotaBaseVector(size_t size, const T & initial_val) [inline]

Constructor, creates vector of size with initial value of initial_val.

Constructor which takes an initial size and an initial value, allocates an area of initial size and initializes it with input value. Calls base class constructor.

6.22.3 Member Function Documentation
6.22.3.1  

template<class T> T & DakotaBaseVector<T>::operator[](size_t i)  [inline]

Returns the object at index i, (can use as lvalue).

Index operator, calls the STL method at() which is bounds checked. Mimics the RW vector class. Note: the at() method is not supported by the __GNUC__ STL implementation and by SGI builds omitting exceptions (e.g., SIERRA).

6.22.3.2  

template<class T> const T & DakotaBaseVector<T>::operator[](size_t i) const  [inline]

Returns the object at index i, const (can’t use as lvalue).

Const versions of the index operator calls the STL method at() which is bounds checked. Mimics the RW vector class. Note: the at() method is not supported by the __GNUC__ STL implementation and by SGI builds omitting exceptions (e.g., SIERRA).

6.22.3.3  template<class T> T & DakotaBaseVector<T>::operator()(size_t i)  [inline]

Index operator, not bounds checked.

Non bounds check index operator, calls the STL operator[] which is not bounds checked. Needed to mimic the RW vector class

6.22.3.4  template<class T> const T & DakotaBaseVector<T>::operator()(size_t i) const  [inline]

Index operator const, not bounds checked.

Const version of the non-bounds check index operator, calls the STL operator[] which is not bounds checked. Needed to mimic the RW vector class

6.22.3.5  template<class T> size_t DakotaBaseVector<T>::length() const  [inline]

Returns size of vector.

Returns the length of the array by calling the STL size method. Needed to mimic the RW vector class

6.22.3.6  template<class T> void DakotaBaseVector<T>::reshape(size_t sz)  [inline]

Resizes vector to size sz.

Resizes the array to size sz by calling the STL resize method. Needed to mimic the RW vector class

6.22.3.7  template<class T> const T * DakotaBaseVector<T>::data() const  [inline]

Returns const pointer to standard C array. Use with care.

Returns a const pointer to the data within the array. USE WITH CARE. Needed to mimic RW vector class.
6.22.3.8 template<class T> T * DakotaBaseVector<T>::array () const [inline, protected]

Returns pointer to standard C array. Use with care.

Returns a non-const pointer to the data within the array. Non-const version of data() used by derived classes.

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

- DakotaBaseVector.H
6.23 DakotaBiStream Class Reference

The binary input stream class. Overloads the >> operator for all data types.

Public Methods

- **DakotaBiStream ()**
  
  Default constructor, need to open.

- **DakotaBiStream (const char *s)**
  
  Constructor takes name of input file.

- **DakotaBiStream (const char *s, std::ios_base::openmode mode)**
  
  Constructor takes name of input file, mode.

- **DakotaBiStream (const char *s, int mode)**
  
  Constructor takes name of input file, mode.

- **~DakotaBiStream ()**
  
  Destructor, calls xdr destroy to delete xdr stream.

- **DakotaBiStream & operator>> (DakotaString &ds)**
  
  Binary Input stream operator>>.

- **DakotaBiStream & operator>> (char *s)**
  
  Input operator, reads char* from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (char &c)**
  
  Input operator, reads char from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (int &i)**
  
  Input operator, reads int* from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (long &l)**
  
  Input operator, reads long from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (short &s)**
  
  Input operator, reads short from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (bool &b)**
  
  Input operator, reads bool from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (double &d)**
  
  Input operator, reads double from binary stream DakotaBiStream.

- **DakotaBiStream & operator>> (float &f)**
6.23 DakotaBiStream Class Reference

Input operator, reads float from binary stream DakotaBiStream.

- DakotaBiStream & operator>>(unsigned char &c)
  Input operator, reads unsigned char from binary stream DakotaBiStream.

- DakotaBiStream & operator>>(unsigned int &i)
  Input operator, reads unsigned int from binary stream DakotaBiStream.

- DakotaBiStream & operator>>(unsigned long &l)
  Input operator, reads unsigned long from binary stream DakotaBiStream.

- DakotaBiStream & operator>>(unsigned short &s)
  Input operator, reads unsigned short from binary stream DakotaBiStream.

Private Attributes

- XDR xdrInBuf
  XDR input stream buffer.

- char inBuf [MAX_NETOBJ_SZ]
  Buffer to hold data as it is read in.

6.23.1 Detailed Description

The binary input stream class. Overloads the >> operator for all data types.
The DakotaBiStream class is a binary input class which overloads the >> operator for all standard data types(int, char, float, etc). The class relies on the methods within the ifstream base class. The DakotaBiStream class inherits from the ifstream class. If available, the class utilize rpc/xdr to construct machine independent binary files. These Dakota restart files can be moved from host to host. The motivation to develop these classes was to replace the Rogue wave classes which Dakota historically used for binary I/O.

6.23.2 Constructor & Destructor Documentation

6.23.2.1 DakotaBiStream::DakotaBiStream ()

Default constructor, need to open.
Default constructor, allocates xdr stream, but does not call the open method. The open method must be called before stream can be read.

6.23.2.2 DakotaBiStream::DakotaBiStream (const char * s)

Constructor takes name of input file.
Constructor which takes a char* filename. Calls the base class open method with the filename and no other arguments. Also allocates the xdr stream.
6.23.2.3 DakotaBiStream::DakotaBiStream (const char * s, std::ios_base::openmode mode)

Constructor takes name of input file, mode.
Constructor which takes a char* filename and int flags. Calls the base class open method with the filename and flags as arguments. Also allocates xdr stream.

6.23.2.4 DakotaBiStream::~DakotaBiStream ()

Destructor, calls xdr_destroy to delete xdr stream.
Destructor, destroys the xdr stream allocated in constructor

6.23.3 Member Function Documentation

6.23.3.1 DakotaBiStream & DakotaBiStream::operator>>(DakotaString & ds)

Binary Input stream operator>>.
The DakotaString input operator must first read both the xdr buffer size and the size of the string written. Once these our read it can then read and convert the DakotaString correctly.

6.23.3.2 DakotaBiStream & DakotaBiStream::operator>>(char * s)

Input operator, reads char* from binary stream DakotaBiStream.
Reading char array is a special case. The method has no way of knowing if the length to the input array is large enough, it assumes it is one char longer than actual string, (Null terminator added). As with the DakotaString the size of the xdr buffer as well as the char array size written must be read from the stream prior to reading and converting the char array.
The documentation for this class was generated from the following files:

- DakotaBinStream.H
- DakotaBinStream.C
6.24 DakotaBoStream Class Reference

The binary output stream class. Overloads the << operator for all data types.

Public Methods

- **DakotaBoStream ()**
  Default constructor, need to open.

- **DakotaBoStream (const char *s)**
  Constructor takes name of input file.

- **DakotaBoStream (const char *s, std::ios_base::openmode mode)**
  Constructor takes name of input file, mode.

- **DakotaBoStream (const char *s, int mode)**
  Constructor takes name of input file, mode.

- **~DakotaBoStream ()**
  Destructor, calls xdr::destroy to delete xdr stream.

- **void testClass ()**
  Performs unit testing for the DakotaBoStream class.

- **DakotaBoStream & operator<<(const DakotaString &ds)**
  Binary Output stream operator<<.

- **DakotaBoStream & operator<<(const char *s)**
  Output operator, writes char* TO binary stream DakotaBoStream.

- **DakotaBoStream & operator<<(const char &c)**
  Output operator, writes char to binary stream DakotaBoStream.

- **DakotaBoStream & operator<<(const int &i)**
  Output operator, writes int to binary stream DakotaBoStream.

- **DakotaBoStream & operator<<(const long &l)**
  Output operator, writes long to binary stream DakotaBoStream.

- **DakotaBoStream & operator<<(const short &s)**
  Output operator, writes short to binary stream DakotaBoStream.

- **DakotaBoStream & operator<<(const bool &b)**
  Output operator, writes bool to binary stream DakotaBoStream.

- **DakotaBoStream & operator<<(const double &d)**
  Output operator, writes double to binary stream DakotaBoStream.
Output operator, writes double to binary stream DakotaBoStream.

- DakotaBoStream & operator<< (const float &f)
  
  Output operator, writes float to binary stream DakotaBoStream.

- DakotaBoStream & operator<< (const unsigned char &c)
  
  Output operator, writes unsigned char to binary stream DakotaBoStream.

- DakotaBoStream & operator<< (const unsigned int &i)
  
  Output operator, writes unsigned int to binary stream DakotaBoStream.

- DakotaBoStream & operator<< (const unsigned long &l)
  
  Output operator, writes unsigned long to binary stream DakotaBoStream.

- DakotaBoStream & operator<< (const unsigned short &s)
  
  Output operator, writes unsigned short to binary stream DakotaBoStream.

Private Attributes

- XDR xdrOutBuf

  XDR output stream buffer.

- char outBuf [MAX_NETOBJ_SZ]

  Buffer to hold converted data before it is written.

6.24.1 Detailed Description

The binary output stream class. Overloads the << operator for all data types.

The DakotaBoStream class is a binary output classes which overloads the << operator for all standard data types (int, char, float, etc). The class relies on the built in write methods within the ostream base classes. DakotaBoStream inherits from the ofstream class. The motivation to develop this class was to replace the Rogue wave class which Dakota historically used for binary I/O. If available, the class utilize rpc/xdr to construct machine independent binary files. These Dakota restart files can be moved between hosts.

6.24.2 Constructor & Destructor Documentation

6.24.2.1 DakotaBoStream::DakotaBoStream ()

Default constructor, need to open.

Default constructor allocates the xdr stream but does not call the open() method. The open() method must be called before stream can be written to.
### 6.24.2.2 DakotaBoStream::DakotaBoStream (const char * s)

Constructor takes name of input file.

Constructor, takes char * filename as argument. Calls base class open method with filename and no other arguments. Also allocates xdr stream.

### 6.24.2.3 DakotaBoStream::DakotaBoStream (const char * s, std::ios_base::openmode mode)

Constructor takes name of input file, mode.

Constructor, takes char * filename and int flags as arguments. Calls base class open method with filename and flags as arguments. Also allocates xdr stream. Note: If no rpc/xdr support xdr calls are ifdef’d out.

### 6.24.3 Member Function Documentation

#### 6.24.3.1 void DakotaBoStream::testClass ()

Performs unit testing for the DakotaBoStream class.

Unit test method for the DakotaBinStream class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an expected answer is not received.

#### 6.24.3.2 DakotaBoStream & DakotaBoStream::operator<< (const DakotaString & ds)

Binary Output stream operator<<.

The DakotaString operator<< must first write the xdr buffer size and the original string size to the stream. The input operator needs this information to be able to correctly read and convert the DakotaString.

#### 6.24.3.3 DakotaBoStream & DakotaBoStream::operator<< (const char * s)

Output operator, writes char* TO binary stream DakotaBoStream.

The output of char* is the same as the output of the DakotaString. The size of the xdr buffer and the size of the string must be written first, then the string itself.

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

- DakotaBinStream.H
- DakotaBinStream.C
6.25 DakotaGraphics Class Reference

The DakotaGraphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.

Public Methods

- **DakotaGraphics ()**
  constructor.

- **~DakotaGraphics ()**
  destructor.

- void **create_plots_2d** (const DakotaVariables &vars, const DakotaResponse &response)
  creates the 2d graphics window and initializes the plots.

- void **create_tabular_datastream** (const DakotaVariables &vars, const DakotaResponse &response, const DakotaString &tabular_data_file)
  opens the tabular data file stream and prints the headings.

- void **add_datapoint** (const DakotaVariables &vars, const DakotaResponse &response)
  adds data to the 2d graphics and tabular data file.

- void **show_data_3d** (DakotaRealArray &X, DakotaRealArray &Y, DakotaRealMatrix &F)
  generate a new 3d plot for F(X,Y).

- void **close ()**
  close graphics windows and tabular datastream.

Private Attributes

- Graphics2D * **graphics2D**
  pointer to the 2D graphics object.

- bool **win2dOn**
  flag to indicate if 2D graphics window is active.

- bool **win3dOn**
  flag to indicate if 3D graphics window is active.

- int **graphicsCntr**
  used for x axis values in 2D graphics and for 1st column in tabular data.

- bool **tabularDataFlag**
  flag to indicate if tabular data stream is active.
6.25 DakotaGraphics Class Reference

- ofstream tabularDataFStream
  
  file stream for tabulation of graphics data within compute\_response.

### 6.25.1 Detailed Description

The DakotaGraphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.

There is only one DakotaGraphics object (dakotaGraphics) and it is global (for convenient access from strategies, models, and approximations).

### 6.25.2 Member Function Documentation

#### 6.25.2.1 void DakotaGraphics::create\_plots\_2d (const DakotaVariables & vars, const DakotaResponse & response)

creates the 2d graphics window and initializes the plots.

Sets up a single event loop for duration of the dakotaGraphics object, continuously adding data to a single window. There is no reset. To start over with a new data set, you need a new object (delete old and instantiate new).

#### 6.25.2.2 void DakotaGraphics::create\_tabular\_datastream (const DakotaVariables & vars, const DakotaResponse & response, const DakotaString & tabular\_data\_file)

opens the tabular data file stream and prints the headings.

Opens the tabular data file stream and prints headings, one for each continuous and discrete variable and one for each response function, using the variable and response function labels. This tabular data is used for post-processing of DAKOTA results in Matlab, Tecplot, etc.

#### 6.25.2.3 void DakotaGraphics::add\_datapoint (const DakotaVariables & vars, const DakotaResponse & response)

adds data to the 2d graphics and tabular data file.

Adds data to each 2d plot and each tabular data column (one for each active variable and for each response function). graphicsCntr is used for the x axis in the graphics and the first column in the tabular data.

#### 6.25.2.4 void DakotaGraphics::show\_data\_3d (DakotaRealArray & X, DakotaRealArray & Y, DakotaRealMatrix & F)

generate a new 3d plot for F(X,Y).

3D plotting clears data set and builds from scratch each time show\_data3d is called. This still involves an event loop waiting for a mouse click (right button) to continue. X = 1-D x grid values only. Y = 1-D Y grid values only. Note that X and Y are not \((X,Y)\) pairs. \(F = 2\)-d grid of values for a single function for all \((X,Y)\) combinations.
The documentation for this class was generated from the following files:

- DakotaGraphics.H
- DakotaGraphics.C
6.26 DakotaInterface Class Reference

Base class for the interface class hierarchy.

Inheritance diagram for DakotaInterface:

```
DakotaInterface
  ↓
ApplicationInterface  ApproximationInterface
  ↓
DirectFnApplicInterface  ForkApplicInterface  SysCallApplicInterface
```

Public Methods

- DakotaInterface ()
  *default constructor.*

- DakotaInterface (ProblemDescDB &problem_db, const size_t &num_acv, const size_t &num_fns)
  *standard constructor for envelope.*

- DakotaInterface (const DakotaInterface &interface)
  *copy constructor.*

- virtual ~DakotaInterface ()
  *destructor.*

- DakotaInterface operator=(const DakotaInterface &interface)
  *assignment operator.*

- virtual void map (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, const bool asynch_flag=0)
  *the function evaluator: provides a "mapping" from the variables to the responses.*

- virtual const DakotaResponseArray & synch ()
  *recovers data from a series of asynchronous evaluations (blocking).*

- virtual const DakotaResponseList & synch_nowait ()
  *recovers data from a series of asynchronous evaluations (nonblocking).*

- virtual void serve_evaluations ()
  *evaluation server function for multiprocessor executions.*

- virtual void stop_evaluation_servers ()
  *send messages from iterator rank 0 to terminate evaluation servers.*
virtual void \texttt{init	extunderscore communicators} (const DakotaIntArray &message\_lengths, const int &max\_iterator, &concurrency)

allocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

virtual void \texttt{free	extunderscore communicators} ()

disable communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

virtual void \texttt{init\_serial} ()

reset certain defaults for serial interface objects.

virtual int \texttt{asynch\_local\_evaluation\_concurrency} () const

return the user-specified concurrency for asynch local evaluations.

virtual DakotaString \texttt{interface\_synchronization} () const

return the user-specified interface synchronization.

virtual int \texttt{minimum\_samples} () const

returns the minimum number of samples required to build a particular \texttt{ApproximationInterface} (used by SurrLayeredModels).

virtual void \texttt{build\_global\_approximation} (DakotaIterator &dace\_iterator, const DakotaRealVector &lower\_bnds, const DakotaRealVector &upper\_bnds)

builds a global approximation for use as a surrogate.

virtual void \texttt{build\_local\_approximation} (DakotaModel &actual\_model)

builds a local approximation for use as a surrogate.

virtual void \texttt{update\_approximation} (const DakotaRealVector &x\_star, const DakotaResponse &response\_star)

updates an existing global approximation with new data.

virtual const DakotaRealVectorArray & \texttt{approximation\_coefficients} ()

retrieve the approximation coefficients from each \texttt{DakotaApproximation} within an \texttt{ApproximationInterface}.

void \texttt{assign\_rep} (DakotaInterface *interface\_rep)

replaces existing letter with a new one.

const DakotaIntList & \texttt{synch\_nowait\_completions} ()

returns id’s matching response list from \texttt{synch\_nowait}().

const DakotaString & \texttt{interface\_type} () const

returns the interface type.

int \texttt{total\_eval\_counter} () const

returns the total number of evaluations of the interface.

int \texttt{new\_eval\_counter} () const
returns the number of new (nonduplicate) evaluations of the interface.

- bool multi_proc_eval_flag () const
  returns a flag signaling the use of multiprocessor evaluation partitions.

- bool iterator_dedicated_master_flag () const
  returns a flag signaling the use of a dedicated master processor for iterator scheduling.

Protected Methods

- DakotaInterface (BaseConstructor, const ProblemDescDB &problem_db)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

Protected Attributes

- DakotaString interfaceType
  interface type may be (1) application: system, fork, direct, or grid; or (2) approximation: ann, rsm, mars, hermite, ksm, mpa, taylor, or hierarchical.

- int fnEvalId
  total evaluation counter.

- int newFnEvalId
  new (non-duplicate) evaluation counter.

- DakotaIntList beforeSynchIdList
  bookkeeps fnEvalId’s of all asynchronous evaluations (new & duplicate).

- DakotaResponseArray rawResponseArray
  The complete array of responses returned after a blocking schedule of asynchronous evaluations.

- DakotaResponseList rawResponseList
  The partial list of responses returned after a nonblocking schedule of asynchronous evaluations.

- DakotaIntList completionList
  identifies the responses in rawResponseList for nonblocking schedules.

- bool multiProcEvalFlag
  flag for multiprocessor evaluation partitions (evalComm).

- bool iteratorDedMasterFlag
  flag for dedicated master partitioning at the iterator level.

- bool silentFlag
  flag for really quiet (silent) interface output.
- bool quietFlag
  flag for quiet interface output.

- bool verboseFlag
  flag for verbose interface output.

- bool debugFlag
  flag for really verbose (debug) interface output.

Private Methods

- DakotaInterface * get_interface (ProblemDescDB &problem_db, const size_t &num_acv, const size_t &num_fns)
  Used by the envelope to instantiate the correct letter class.

Private Attributes

- DakotaInterface * interfaceRep
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  number of objects sharing interfaceRep.

6.26.1 Detailed Description

Base class for the interface class hierarchy.

The DakotaInterface class hierarchy provides the part of a DakotaModel that is responsible for mapping a set of DakotaVariables into a set of DakotaResponses. The mapping is performed using either a simulation-based application interface or a surrogate-based approximation interface. For memory efficiency and enhanced polymorphism, the interface hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaInterface) serves as the envelope and one of the derived classes (selected in DakotaInterface::get_interface()) serves as the letter.

6.26.2 Constructor & Destructor Documentation

6.26.2.1 DakotaInterface::DakotaInterface ()

default constructor.

used in DakotaModel envelope class instantiations
6.26.2.2 DakotaInterface::DakotaInterface (ProblemDescDB & problem_db, const size_t & num_acv, const size_t & num_fns)

standard constructor for envelope.

Used in DakotaModel instantiation to build the envelope. This constructor only needs to extract enough data to properly execute get_interface, since DakotaInterface::DakotaInterface(BaseConstructor, problem_db) builds the actual base class data inherited by the derived interfaces.

6.26.2.3 DakotaInterface::DakotaInterface (const DakotaInterface & interface)

copy constructor.

Copy constructor manages sharing of interfaceRep and incrementing of referenceCount.

6.26.2.4 DakotaInterface::~DakotaInterface () [virtual]
destructor.

Destructor decrements referenceCount and only deletes interfaceRep if referenceCount is zero.

6.26.2.5 DakotaInterface::DakotaInterface (BaseConstructor, const ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

This constructor is the one which must build the base class data for all inherited interfaces. get_interface() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_interface() again). Since this is the letter and the letter IS the representation, interfaceRep is set to NULL (an uninitialized pointer causes problems in ~DakotaInterface).

6.26.3 Member Function Documentation

6.26.3.1 DakotaInterface DakotaInterface::operator= (const DakotaInterface & interface)

assignment operator.


6.26.3.2 void DakotaInterface::assign_rep (DakotaInterface * interface_rep)

replaces existing letter with a new one.

Similar to the assignment operator, the assign_rep() function decrements referenceCount for old interfaceRep, assigns the new interfaceRep, and increments referenceCount for new interfaceRep. It is different in the sense that it is used for publishing derived class letter objects, as opposed to a base class envelope object.
6.26.3  DakotaInterface: get_interface (ProblemDescDB & problem.db, const size_t & num_acv, const size_t & num_fns) [private]

Used by the envelope to instantiate the correct letter class.
used only by the envelope constructor to initialize interfaceRep to the appropriate derived type, as given by
the interfaceType attribute.

6.26.4  Member Data Documentation

6.26.4.1  DakotaResponseArray DakotaInterface::rawResponseArray  [protected]

The complete array of responses returned after a blocking schedule of asynchronous evaluations.
The array is the raw set of responses corresponding to all asynchronous map calls. This raw array is
postprocessed (i.e., finite difference gradients merged) in DakotaModel::synchronize() where it becomes
responseArray.

6.26.4.2  DakotaResponseList DakotaInterface::rawResponseList  [protected]

The partial list of responses returned after a nonblocking schedule of asynchronous evaluations.
The list is a partial set of completions which must be identified through the use of completionList. Post-
processing from raw to combined form (i.e., finite difference gradient merging) is not currently supported
in DakotaModel::synchronize_nowait().

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

- DakotaInterface.H
- DakotaInterface.C
6.27 DakotaIterator Class Reference

Base class for the iterator class hierarchy.

Inheritance diagram for DakotaIterator::

```
Base class for the iterator class hierarchy.

Public Methods

- DakotaIterator ()
  default constructor.

- DakotaIterator (DakotaModel &model)
  standard constructor for envelope.

- DakotaIterator (const DakotaIterator &iterator)
  copy constructor.

- virtual ~DakotaIterator ()
  destructor.

- DakotaIterator operator= (const DakotaIterator &iterator)
  assignment operator.

- virtual void run_iterator ()
  run the iterator.

- virtual const DakotaVariables & iterator_variable_results () const
  return the final iterator solution (variables).

- virtual const DakotaResponse & iterator_response_results () const
  return the final iterator solution (response).

- virtual void print_iterator_results (ostream &s) const
  print the final iterator results.
```
virtual void multi\_objective\_weights (const DakotaRealVector &multi\_obj\_wts) 
set the relative weightings for multiple objective functions. Used by ConcurrentStrategy for Pareto set optimization.

virtual void sampling\_reset (int min\_samples, bool all\_data\_flag, bool stats\_flag) 
reset sampling iterator.

virtual const DakotaString & sampling\_scheme () const 
return sampling name.

void user\_defined\_model (const DakotaModel &the\_model) 
set the model.

DakotaModel & user\_defined\_model () const 
return the model.

const DakotaString & method\_name () const 
return the method name.

int maximum\_concurrency () const 
return the maximum concurrency supported by the iterator.

const DakotaVariablesArray & all\_variables () const 
return the complete set of evaluated variables.

const DakotaRealVectorArray & all\_c\_variables () const 
return the complete set of evaluated continuous variables.

const DakotaResponseArray & all\_responses () const 
return the complete set of computed responses.

const DakotaRealVectorArray & all\_fn\_responses () const 
return the complete set of computed function responses.

bool is\_null () const 
function to check iteratorRep (does this envelope contain a letter).

Protected Methods

- **DakotaIterator (BaseConstructor, DakotaModel \&model)**
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

- **DakotaIterator (NoDBBaseConstructor, DakotaModel \&model)**
  base class for iterator classes constructed on the fly (no DB queries).

- virtual void update\_best (const DakotaRealVector \&vars, const DakotaResponse \&response, const int eval\_num)
void evaluate parameter sets (bool vars_flag, bool resp_flag, bool fns_flag, bool best_flag)
perform function evaluations to map parameter sets (allVariables/allCVariables/allDVariables) into response sets (allResponses/allFnResponses/allGradResponses).

Protected Attributes

- DakotaModel & userDefinedModel
class member reference for the model passed into the constructor.

- const ProblemDescDB & probDescDB
class member reference to the problem description database.

- DakotaString methodName
name of the iterator (the user’s method spec).

- int maxIterations
maximum number of iterations for the iterator.

- int maxFunctionEvals
maximum number of fn evaluations for the iterator.

- int numFunctions
number of response functions.

- int maxConcurrency
maximum coarse-grained concurrency.

- int numContinuousVars
number of active continuous vars.

- int numDiscreteVars
number of active discrete vars.

- int numVars
total number of vars. (active and inactive).

- DakotaIntArray activeSetVector
this vector tracks the data requirements for the response functions. It uses a 0 value for inactive functions and, for active functions, sums 1 for value, 2 for gradient, and 4 for Hessian.

- DakotaString gradientType
type of gradient data: "analytic", "numerical", "mixed", or "none".

- DakotaString hessianType
type of Hessian data: "analytic" or "none".

- DakotaString finiteDiffType
type of finite difference interval: "central" or "forward".

- **DakotaString methodSource**
  source of finite difference routine: "dakota" or "vendor".

- **Real finiteDiffStepSize**
  relative finite difference step size.

- **DakotaIntList mixedGradAnalyticIds**
  for mixed gradients, contains ids of functions with analytic gradients.

- **DakotaIntList mixedGradNumericalIds**
  for mixed gradients, contains ids of functions with numerical gradients.

- **bool silentOutput**
  flag for really quiet (silent) algorithm output.

- **bool quietOutput**
  flag for quiet algorithm output.

- **bool verboseOutput**
  flag for verbose algorithm output.

- **bool debugOutput**
  flag for really verbose (debug) algorithm output.

- **bool asynchFlag**
  copy of the model’s asynchronous evaluation flag.

- **DakotaVariablesArray allVariables**
  array of all variables evaluated.

- **DakotaRealVectorArray allCVariables**
  array of all continuous variables evaluated (subset of allVariables).

- **DakotaResponseArray allResponses**
  array of all responses computed.

- **DakotaRealVectorArray allFnResponses**
  array of all function responses computed (subset of allResponses).

- **DakotaStringArray allHeaders**
  array of headers to insert into output while evaluating allCVariables.

**Static Protected Attributes**

- **DakotaModel & staticModel = dummyModel**
  static model reference used by OPT++, NPSOL, NonDAMV.
Private Methods

- DakotalItera**tor** *get_iterator*(Dakota**Model** &model)
  
  Used by the envelope to instantiate the correct letter class.

- void populate_gradient_vars()
  
  Used only by constructor functions to define gradient variables for use within the iterator hierarchy.

Private Attributes

- DakotalItera**tor** *iteratorRep
  
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  
  number of objects sharing iteratorRep.

6.27.1 Detailed Description

Base class for the iterator class hierarchy.

The DakotalItera**tor** class is the base class for one of the primary class hierarchies in DAKOTA. The iterator hierarchy contains all of the iterative algorithms which use repeated execution of simulations as function evaluations. For memory efficiency and enhanced polymorphism, the iterator hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotalItera**tor**) serves as the envelope and one of the derived classes (selected in DakotalItera**tor**::get_iterator()) serves as the letter.

6.27.2 Constructor & Destructor Documentation

6.27.2.1 DakotalItera**tor**::DakotalItera**tor**()

default constructor.

The default constructor is used in Vector<DakotaItera**tor**> instantiations and for initialization of DakotalItera**tor** objects contained in Dakota**Strategy** derived classes (see derived class header files). iteratorRep is NULL in this case (a populated problem db is needed to build a meaningful DakotalItera**tor** object). This makes it necessary to check for NULL pointers in the copy constructor, assignment operator, and destructor.

6.27.2.2 DakotalItera**tor**::DakotalItera**tor**(Dakota**Model** & model)

standard constructor for envelope.

Used in iterator instantiations within strategy constructors. Envelope constructor only needs to extract enough data to properly execute get_iterator, since DakotalItera**tor**(Base**Constructor**, model) builds the actual base class data inherited by the derived iterators.
6.27.2.3 DakotaIterator::DakotaIterator (const DakotaIterator & iterator)

copy constructor.
Copy constructor manages sharing of iteratorRep and incrementing of referenceCount.

6.27.2.4 DakotaIterator::~DakotaIterator () [virtual]
destructor.
Destructor decrements referenceCount and only deletes iteratorRep when referenceCount reaches zero.

6.27.2.5 DakotaIterator::DakotaIterator (BaseConstructor, DakotaModel & model) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor builds the base class data for all inherited iterators. get_iterator() instantiates a derived class and the derived class selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_iterator() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaIterator).

6.27.2.6 DakotaIterator::DakotaIterator (NoDBBaseConstructor, DakotaModel & model) [protected]

base class for iterator classes constructed on the fly (no DB queries).
This constructor also builds base class data for inherited iterators. However, it is used for on-the-fly instantiations for which DB queries cannot be used (e.g., ApproximationInterface instantiation of DACEIterator or NonDProbability, AMV usage of optimizers, etc.). Therefore it only sets attributes taken from the incoming model.

6.27.3 Member Function Documentation

6.27.3.1 DakotaIterator DakotaIterator::operator= (const DakotaIterator & iterator)

assignment operator.

6.27.3.2 void DakotaIterator::run_iterator () [virtual]
run the iterator.
This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.
Reimplemented in DACEIterator.
6.27.3.3 void DakotaIterator::evaluate_parameter_sets (bool vars flag, bool resp flag, bool fns flag, bool best flag) [protected]

perform function evaluations to map parameter sets (allVariables/allCVariables/allDVariables) into response sets (allResponses/allFnResponses/allGradResponses).

Convenience function for derived classes with sets of function evaluations to perform (e.g., NonDSampling, DACEIterator, ParamStudy).

6.27.3.4 DakotaIterator * DakotaIterator::get_iterator (DakotaModel & model) [private]

Used by the envelope to instantiate the correct letter class.

Used only by the envelope constructor to initialize iteratorRep to the appropriate derived type, as given by the methodName attribute.

6.27.3.5 void DakotaIterator::populate_gradient_vars () [private]

Used only by constructor functions to define gradient variables for use within the iterator hierarchy.

Convenience function for constructors. Populates gradient and Hessian data attributes from the problem description database.

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

- DakotaIterator.H
- DakotaIterator.C
6.28 DakotaLeastSq Class Reference

Base class for the nonlinear least squares branch of the iterator hierarchy.

Inheritance diagram for DakotaLeastSq:

- DakotaIterator
- DakotaOptLeastSq
- DakotaLeastSq
- NLSSOLLeastSq
- SNLLLeastSq

Public Methods

- void run_iterator ()
  
  run the iterator.

- void print_iterator_results (ostream &s) const

Protected Methods

- DakotaLeastSq ()
  
  default constructor.

- DakotaLeastSq (DakotaModel &model)
  
  standard constructor.

- ~DakotaLeastSq ()
  
  destructor.

- virtual void minimize_residuals ()=0
  
  Used within the least squares branch for minimizing the sum of squares residuals. Redefines the run_iterator virtual function for the least squares branch.

Protected Attributes

- int numLeastSqTerms
  
  number of least squares terms.
6.28 DakotaLeastSq Class Reference

Static Protected Attributes

- `size_t staticNumLSqTerms`

  static copy of numLeastSqTerms used in static functions passed by function pointer (NLSSOL and OPT++).

6.28.1 Detailed Description

Base class for the nonlinear least squares branch of the iterator hierarchy.

The DakotaLeastSq class provides common data and functionality for NLSSOLLeastSq and SNLLLeastSq.

6.28.2 Constructor & Destructor Documentation

6.28.2.1 DakotaLeastSq::DakotaLeastSq (DakotaModel & model) [protected]

standard constructor.

This constructor extracts the inherited data for the least squares branch and performs sanity checking on gradient and constraint settings.

6.28.3 Member Function Documentation

6.28.3.1 void DakotaLeastSq::run_iterator () [inline, virtual]

run the iterator.

This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from DakotaIterator.

6.28.3.2 void DakotaLeastSq::print_iterator_results (ostream & s) const [virtual]

Redefines default iterator results printing to include optimization results (objective function and constraints).

Reimplemented from DakotaIterator.

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

- DakotaLeastSq.H
- DakotaLeastSq.C

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (©) 1997-2002
6.29 DakotaList Class Template Reference

Template class for the Dakota bookkeeping list.

Public Methods

- **DakotaList ()**
  *Default constructor.*

- **DakotaList (const DakotaList &a)**
  *Copy constructor.*

- **~DakotaList ()**
  *Destructor.*

- **template<class InputIter> DakotaList (InputIter first, InputIter last)**
  *Range constructor (member template).*

- **DakotaList<T> & operator= (const DakotaList<T> &a)**
  *Assignment operator.*

- **void testClass ()**
  *Class unit test method.*

- **void print (ostream &s) const**
  *Prints a DakotaList to an output stream.*

- **void read (UnPackBuffer &s)**
  *Reads a DakotaList from an UnPackBuffer after an MPI receive.*

- **void print (PackBuffer &s) const**
  *Prints a DakotaList to a PackBuffer prior to an MPI send.*

- **size_t entries () const**
  *Returns the number of items that are currently in the list.*

- **T get ()**
  *Removes and returns the first item in the list.*

- **T removeAt (size_t index)**
  *Removes and returns the item at the specified index.*

- **bool remove (const T &a)**
  *Removes the specified item from the list.*

- **void insert (const T &a)**
6.29 DakotaList Class Template Reference

Adds the item a to the end of the list.

- bool contains (const T &a) const
  Returns TRUE if list contains object a, returns FALSE otherwise.

- bool find (bool(*testFun)(const T &, void *), void *d, T &k) const
  Returns TRUE if the list contains an object which the user defined function finds and sets k to this object.

- size_t index (bool(*testFun)(const T &, void *), void *d) const
  Returns the index of object which the user defined test function finds.

- void sort (bool(*sortFun)(const T &, const T &))
  Sorts the list into an order based on the predefined sort function.

- size_t index (const T &a) const
  Returns the index of the object.

- size_t occurrencesOf (const T &a) const
  Returns the number of items in the list equal to object.

- bool isEmpty () const
  Returns TRUE if list is empty, returns FALSE otherwise.

- T & operator[ ] (size_t i)
  Returns the object at index i (can use as lvalue).

- const T & operator[ ] (size_t i) const
  Returns the object at index i, const (can’t use as lvalue).

6.29.1 Detailed Description

template<class T> class DakotaList<T>

Template class for the Dakota bookkeeping list.

The DakotaList is the common list class for Dakota. It inherits from either the RW list class or the STL list class. Extends the base list class to add Dakota specific methods Builds upon the previously existing DakotaValList class

6.29.2 Member Function Documentation

6.29.2.1 template<class T> void DakotaList<T>::testClass ()

Class unit test method.

Unit test method for the DakotaList class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an unexpected answer is received.
6.29.2.2  template<class T> T DakotaList<T>::get()

Removes and returns the first item in the list.
Remove and return item from front of list. Returns the object pointed to by the list::begin() iterator. It also deletes the first node by calling the list::pop_front() method. Note: get() is not the same as list::front() since the latter would return the 1st item but would not delete it.

6.29.2.3  template<class T> T DakotaList<T>::removeAt(size_t index)

Removes and returns the item at the specified index.
Removes the item at the index specified. Uses the STL advance() function to step to the appropriate position in the list and then calls the list::erase() method.

6.29.2.4  template<class T> bool DakotaList<T>::remove(const T &a)

Removes the specified item from the list.
Removes the first instance matching object a from the list (and therefore differs from the STL list::remove() which removes all instances). Uses the STL find() algorithm to find the object and the list::erase() method to perform the remove.

6.29.2.5  template<class T> void DakotaList<T>::insert(const T &a) [inline]

Adds the item a to the end of the list.
Insert item at the end of list, calls list::push_back() method which places the object at the end of the list.

6.29.2.6  template<class T> bool DakotaList<T>::contains(const T &a) const [inline]

Returns TRUE if list contains object a, returns FALSE otherwise.
Uses the STL find() algorithm to locate the first instance of object a. Returns true if an instance is found.

6.29.2.7  template<class T> bool DakotaList<T>::find(bool(*testFun)(const T &, void *), void *d, T &k) const

Returns TRUE if the list contains an object which the user defined function finds and sets k to this object.
Find the first item in the list which satisfies the test function. Sets k if the object is found.

6.29.2.8  template<class T> size_t DakotaList<T>::index(bool(*testFun)(const T &, void *), void *d) const

Returns the index of object which the user defined test function finds.
Returns the index of the first item in the list which satisfies the test function. Uses a single list traversal to both locate the object and return its index (generic algorithms would require two loop traversals).
6.29.2.9 \texttt{template<class T> void DakotaList\textless T\textgreater::sort (bool\* sortFun)(const T &, const T &)}

\[\text{inline}\]

Sorts the list into an order based on the predefined sort function.

The sort method utilizes the \texttt{SortCompare} functor and the base class \texttt{list::sort} algorithm to sort a list based on the incoming sorting function \texttt{sortFun}. Note that the functor-based sorting method of \texttt{std::list} is not supported by all compilers (e.g., SOLARIS, TFLOP) due to use of member templates, but a function pointer-based interface is available in some cases.

6.29.2.10 \texttt{template<class T> size\_t DakotaList\textless T\textgreater::index (const T & a) const}

Returns the index of the object.

Returns the index of the first item in the list which matches the object \texttt{a}. Uses a single list traversal to both locate the object and return its index (generic algorithms would require two loop traversals).

6.29.2.11 \texttt{template<class T> size\_t DakotaList\textless T\textgreater::occurrencesOf (const T & a) const}

\[\text{inline}\]

Returns the number of items in the list equal to object.

Uses the STL \texttt{count()} algorithm to return the number of occurrences of the specified object.

6.29.2.12 \texttt{template<class T> T & DakotaList\textless T\textgreater::operator[ ] (size\_t i)}

\texttt{template<const T &}}

Returns the object at index \texttt{i} (can use as lvalue).

Returns item at position \texttt{i} of the list by stepping through the list using forward or reverse STL iterators (depending on which end of the list is closer to the desired item). Once the object is found, it returns the value pointed to by the iterator.

This functionality is inefficient in 0-len loop-based list traversals and is being replaced by iterator-based list traversals in the main DAKOTA code. For isolated look-ups of a particular index, however, this approach is acceptable.

6.29.2.13 \texttt{template<class T> const T & DakotaList\textless T\textgreater::operator[ ] (size\_t i) const}

\texttt{template<const T &}}

Returns the object at index \texttt{i}, const (can’t use as lvalue).

Returns const item at position \texttt{i} of the list by stepping through the list using forward or reverse STL iterators (depending on which end of the list is closer to the desired item). Once the object is found it returns the value pointed to by the iterator.

This functionality is inefficient in 0-len loop-based list traversals and is being replaced by iterator-based list traversals in the main DAKOTA code. For isolated look-ups of a particular index, however, this approach is acceptable.

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

- DakotaList.H

---

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (c) 1997-2002
6.30 DakotaMatrix Class Template Reference

Template class for the Dakota numerical matrix.

Inheritance diagram for DakotaMatrix::

```
DakotaMatrix
  |________________|
  |             |
  | DakotaBaseVector< DakotaBaseVector< T > > |
  |             |
  |____________|
  DakotaMatrix
```

Public Methods

- **DakotaMatrix** (size_t numRows=0, size_t numCols=0)
  
  Constructor, takes number of rows, and number of columns as arguments.

- **~DakotaMatrix ()**
  
  Destructor.

- **DakotaMatrix< T > & operator= (const T &ival)**
  
  Sets all elements in the matrix to ival.

- **size_t numRows () const**
  
  Returns the number of rows for the matrix.

- **size_t numColumns () const**
  
  Returns the number of columns for the matrix.

- **void reshape2d (size_t numRows, size_t numCols)**
  
  Resizes the matrix to numRows by numCols.

- **void print (ostream &s) const**
  
  Prints a DakotaMatrix to an output stream.

- **void read (UnPackBuffer &s)**
  
  Reads a DakotaMatrix from an UnPackBuffer after an MPI receive.

- **void print (PackBuffer &s) const**
  
  Prints a DakotaMatrix to a PackBuffer prior to an MPI send.

- **void testClass ()**
  
  Class unit test method.
6.30 DakotaMatrix Class Template Reference

6.30.1 Detailed Description

template<class T> class DakotaMatrix<T>

Template class for the Dakota numerical matrix.

A matrix class template to provide 2D arrays of objects. The matrix is zero-based, rows: 0 to (numRows-1) and cols: 0 to (numColumns-1). The class supports overloading of the subscript operator allowing it to emulate a normal built-in 2D array type. The DakotaMatrix relies on the DakotaBaseVector template class to manage the differences between the Rogue Wave vector class and the STL vector class.

6.30.2 Member Function Documentation

6.30.2.1 template<class T> DakotaMatrix<T> & DakotaMatrix<T>::operator= (const T & val) [inline]

Sets all elements in the matrix to ival.

calls base class operator=(ival)

6.30.2.2 template<class T> void DakotaMatrix<T>::testClass ()

Class unit test method.

verifies the basic functionality of the DakotaMatrix class. The assert function is used to test the correctness of results.

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

- DakotaMatrix.H
6.31 DakotaModel Class Reference

Base class for the model class hierarchy.

Inheritance diagram for DakotaModel::

```
DakotaModel
   LayeredModel
      HierLayeredModel
      SurrLayeredModel
   NestedModel
   SingleModel
```

Public Methods

- **DakotaModel ()**
  
  *default constructor.*

- **DakotaModel (ProblemDescDB &problem_db)**
  
  *standard constructor for envelope.*

- **DakotaModel (const DakotaModel &model)**
  
  *copy constructor.*

- **virtual ~DakotaModel ()**
  
  *destructor.*

- **DakotaModel operator= (const DakotaModel &model)**
  
  *assignment operator.*

- **virtual DakotaModel & subordinate_model ()**
  
  *return the sub-model in nested and layered models.*

- **virtual DakotaIterator & subordinate_iterator ()**
  
  *return the sub-iterator in nested and layered models.*

- **virtual int maximum_concurrency () const**
  
  *used to return DACE iterator concurrency for SurrLayeredModels.*

- **virtual void build_approximation ()**
  
  *build the approximation in LayeredModels.*

- **virtual void update_approximation (const DakotaRealVector &x_star, const DakotaResponse &response_star)**
  
  *update the approximation in SurrLayeredModels with new data.*
virtual const DakotaRealVectorArray & approximation_coefficients ()
retrieve the approximation coefficients from each DakotaApproximation within a SurrLayeredModel.

virtual void compute_correction (const DakotaResponse &truth_response, const DakotaResponse &approx_response, const DakotaRealVector &c_vars)
compute correction factors for use in LayeredModels.

virtual void auto_correction (bool correction_flag)
manages automatic application of correction factors in LayeredModels.

virtual void apply_correction (DakotaResponse &approx_response, const DakotaRealVector &c_vars, bool quiet_flag=0)
apply correction factors to approx_response (for use in LayeredModels).

virtual DakotaString local_eval_synchronization ()
return derived model synchronization setting.

virtual void free_communicators ()
deallocate communicator partitions for a model.

virtual void serve ()
Service job requests received from the master. Completes when a termination message is received from stop_servers().

virtual void stop_servers ()
Executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete.

virtual const DakotaIntArray & synchronize_nowait_completions ()
Return completion id’s matching response list from synchronize_nowait.

virtual bool derived_master_overload () const
Return a flag indicating the combination of multiprocessor evaluations and a dedicated master iterator scheduling. Used in synchronous compute_response functions to prevent the error of trying to run a multi-processor job on the master.

virtual int total_eval_counter () const
Return the total evaluation count from the interface.

virtual int new_eval_counter () const
Return the new (non-duplicate) evaluation count from the interface.

void compute_response ()
Compute the DakotaResponse at currentVariables (default asv).

void compute_response (const DakotaIntArray &asv)
Compute the DakotaResponse at currentVariables (specified asv).

void async_compute_response ()
Spawn an asynchronous job (or jobs) that computes the value of the DakotaResponse at currentVariables (default asv).

- **void asynchComputeResponse** (const DakotaIntArray &asv)
  Spawn an asynchronous job (or jobs) that computes the value of the DakotaResponse at currentVariables (specified asv).

- **const DakotaResponseArray & synchronize()**
  Execute a blocking scheduling algorithm to collect the complete set of results from a group of asynchronous evaluations.

- **const DakotaResponseList & synchronize_nowait()**
  Execute a nonblocking scheduling algorithm to collect all available results from a group of asynchronous evaluations.

- **void initCommunicators** (const int &max_iterator_concurrency)
  allocate communicator partitions for a model.

- **size_t tv() const**
  return total number of vars.

- **size_t cv() const**
  return number of active continuous variables.

- **size_t dv() const**
  return number of active discrete variables.

- **size_t num_functions() const**
  return number of functions in currentResponse.

- **void active_variables** (const DakotaVariables &vars)
  set the active variables in currentVariables.

- **const DakotaRealVector & continuous_variables() const**
  return the active continuous variables from currentVariables.

- **void continuous_variables** (const DakotaRealVector &c_vars)
  set the active continuous variables in currentVariables.

- **const DakotaIntVector & discrete_variables() const**
  return the active discrete variables from currentVariables.

- **void discrete_variables** (const DakotaIntVector &d_vars)
  set the active discrete variables in currentVariables.

- **const DakotaStringArray & continuous_variable_labels() const**
  return the active continuous variable labels from currentVariables.

- **const DakotaStringArray & discrete_variable_labels() const**
  return the active discrete variable labels from currentVariables.
- void **inactive_continuous_variables** (const DakotaRealVector &i_c_vars)
  
  *set the inactive continuous variables in currentVariables.*

- void **inactive_discrete_variables** (const DakotaIntVector &i_d_vars)
  
  *set the inactive discrete variables in currentVariables.*

- const DakotaRealVector & **continuous_lower_bounds** () const
  
  *return the active continuous variable lower bounds from userDefinedVarConstraints.*

- void **continuous_lower_bounds** (const DakotaRealVector &c_l_bnds)
  
  *set the active continuous variable lower bounds in userDefinedVarConstraints.*

- const DakotaRealVector & **continuous_upper_bounds** () const
  
  *return the active continuous variable upper bounds from userDefinedVarConstraints.*

- void **continuous_upper_bounds** (const DakotaRealVector &c_u_bnds)
  
  *set the active continuous variable upper bounds in userDefinedVarConstraints.*

- const DakotaIntVector & **discrete_lower_bounds** () const
  
  *return the active discrete variable lower bounds from userDefinedVarConstraints.*

- void **discrete_lower_bounds** (const DakotaIntVector &d_l_bnds)
  
  *set the active discrete variable lower bounds in userDefinedVarConstraints.*

- const DakotaIntVector & **discrete_upper_bounds** () const
  
  *return the active discrete variable upper bounds from userDefinedVarConstraints.*

- void **discrete_upper_bounds** (const DakotaIntVector &d_u_bnds)
  
  *set the active discrete variable upper bounds in userDefinedVarConstraints.*

- void **inactive_continuous_lower_bounds** (const DakotaRealVector &i_c_l_bnds)
  
  *set the inactive continuous lower bounds in userDefinedVarConstraints.*

- void **inactive_continuous_upper_bounds** (const DakotaRealVector &i_c_u_bnds)
  
  *set the inactive continuous upper bounds in userDefinedVarConstraints.*

- void **inactive_discrete_lower_bounds** (const DakotaIntVector &i_d_l_bnds)
  
  *set the inactive discrete lower bounds in userDefinedVarConstraints.*

- void **inactive_discrete_upper_bounds** (const DakotaIntVector &i_d_u_bnds)
  
  *set the inactive discrete upper bounds in userDefinedVarConstraints.*

- const DakotaRealMatrix & **linear_ineq_constraint_coeffs** () const
  
  *return the linear inequality constraint coefficients.*

- size_t **num_linear_ineq_constraints** () const
  
  *return the number of linear inequality constraints.*

- size_t **num_linear_eq_constraints** () const
  
  *return the number of linear equality constraints.*

- const DakotaRealMatrix & **linear_eq_constraint_coeffs** () const
  
  *return the linear equality constraint coefficients.*
• const DakotaRealVector & linear_ineq_constraint_lower_bounds () const
  return the linear inequality constraint lower bounds.

• const DakotaRealVector & linear_ineq_constraint_upper_bounds () const
  return the linear inequality constraint upper bounds.

• const DakotaRealMatrix & linear_eq_constraint_coeffs () const
  return the linear equality constraint coefficients.

• const DakotaRealVector & linear_eq_constraint_targets () const
  return the linear equality constraint targets.

• const DakotaIntList & merged_integer_list () const
  return the list of discrete variables merged into a continuous array in currentVariables.

• const DakotaIntArray & message_lengths () const
  return the array of MPI packed message buffer lengths (messageLengths).

• const DakotaVariables & current_variables () const
  return the current variables (currentVariables).

• const DakotaResponse & current_response () const
  return the current response (currentResponse).

• const ProblemDescDB & probDescDB () const
  return the problem description database (probDescDB).

• const DakotaString & model_type () const
  return the model type (modelType).

• bool asynch_flag () const
  return the asynchronous evaluation flag (asynchEvalFlag).

• void asynch_flag (const bool flag)
  set the asynchronous evaluation flag (asynchEvalFlag).

• void activate_model_auto_graphics ()
  set modelAutoGraphicsFlag to activate posting of graphics data within compute_response/synchronize functions (automatic graphics posting in the model as opposed to graphics posting at the strategy level).

• const DakotaString & gradient_method () const
  return the gradient method (gradType).

• int gradient_concurrency () const
  return the gradient concurrency for use in parallel configuration logic.

• bool is_null () const
  function to check modelRep (does this envelope contain a letter).
Protected Methods

- **DakotaModel (BaseConstructor, ProblemDescDB &problem_db)**
  
  *constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).*

- virtual void derived_compute_response (const DakotaIntArray &asv)
  
  *portion of compute_response() specific to derived model classes.*

- virtual void derived_asynch_compute_response (const DakotaIntArray &asv)
  
  *portion of asynch_compute_response() specific to derived model classes.*

- virtual const DakotaResponseArray & derived_synchronize ()
  
  *portion of synchronize() specific to derived model classes.*

- virtual const DakotaResponseList & derived_synchronize_nowait ()
  
  *portion of synchronize_nowait() specific to derived model classes.*

- virtual void derived_init_communicators (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  
  *portion of init_communicators() specific to derived model classes.*

Protected Attributes

- **DakotaVariables currentVariables**
  
  *the set of current variables used by the model for performing function evaluations.*

- size_t numGradVars
  
  *the number of active continuous variables (used in the finite difference routines).*

- **DakotaResponse currentResponse**
  
  *the set of current responses that holds the results of model function evaluations.*

- size_t numFs
  
  *the number of functions in currentResponse.*

- **DakotaVarConstraints userDefinedVarConstraints**
  
  *Explicit constraints on variables are maintained in the DakotaVarConstraints class hierarchy. Currently, this includes linear constraints and bounds, but could be extended in the future to include other explicit constraints which (1) have their form specified by the user, and (2) are not catalogued in DakotaResponse since their form and coefficients are published to an iterator at startup.*

Private Methods

- **DakotaModel * get_model (ProblemDescDB &problem_db)**
  
  *Used by the envelope to instantiate the correct letter class.*
• int fd_gradients (const DakotaIntArray &map_asv, const DakotaIntArray &fd_grad_asv, const DakotaIntArray &original_asv, const int asynch_flag)
  evaluate numerical gradients using finite differences. This routine is selected with "method_source dakota" (the default method_source) in the numerical gradient specification.

• void synchronize_fd_gradients (const DakotaResponseArray &fd_grad_responses, DakotaResponse &new_response, const DakotaIntArray &fd_grad_asv, const DakotaIntArray &asv)
  combine results from an array of finite difference response objects (fd_grad_responses) into a single response (new_response).

• void update_response (DakotaResponse &new_response, const DakotaIntArray &fd_grad_asv, const DakotaIntArray &asv, const bool initial_map, DakotaRealVector &fn_vals_x0, DakotaRealMatrix &partial_fn_grads, const DakotaRealMatrix &new_fn_grads)
  overlay results to update a response object.

• void manage_asv (const DakotaIntArray &asv_in, DakotaIntArray &map_asv_out, DakotaIntArray &fd_grad_asv_out, int &use_fd_grad)
  Coordinates map() and fd_gradients() calls given an asv_in input.

**Private Attributes**

• DakotaModel * modelRep
  pointer to the letter (initialized only for the envelope).

• int referenceCount
  number of objects sharing modelRep.

• const ProblemDescDB & probDescDB
  class member reference to the problem description database. This reference is a const copy of the incoming problem db non-const reference and is only used in DakotaModel::prob_desc_db() (it is not inherited).

• const ParallelLibrary & parallelLib
  class member reference to the parallel library.

• DakotaIntArray messageLengths
  length of packed MPI buffers containing vars, vars and asv, response, and PRPair.

• DakotaString modelType
  type of model: single, nested, or layered.

• bool asynchFDFlag
  flags use of fd_gradients w/ asynch compute response.

• bool asynchEvalFlag
  flags asynch evaluations (local or distributed).

• bool modelAutoGraphicsFlag
  flag for posting of graphics data within compute_response (automatic graphics posting in the model as opposed to graphics posting at the strategy level).
- bool silentFlag
  flag for really quiet (silent) model output.

- bool quietFlag
  flag for quiet model output.

- DakotaVariablesList varsList
  history of vars populated in asynch_compute_response() and used in synchronize().

- DakotaList< DakotaIntArray > asvList
  if asynchFDFlag is set, transfers asv requests to synchronize.

- DakotaBoolList initialMapList
  transfers initial_map flag values from fd_gradients to synchronize_fd_gradients.

- DakotaBoolList dbFnsList
  transfers db_fns flag values from fd_gradients to synchronize_fd_gradients.

- DakotaResponseList dbResponseList
  transfers database captures from fd_gradients to synchronize_fd_gradients.

- DakotaRealList deltaList
  transfers deltas from fd_gradients to synchronize_fd_gradients.

- DakotaIntList numMapsList
  tracks the number of maps used in fd_gradients(). Used in synchronize() as a key for combining finite difference responses into numerical gradients.

- DakotaResponseArray responseArray
  used to return an array of responses for asynchronous evaluations. This array has the responses in final concatenated form. The similar array in DakotaInterface contains the raw responses.

- DakotaResponseList responseList
  used to return a list of responses for asynchronous evaluations. This list has the responses in final concatenated form. The similar list in DakotaInterface contains the raw responses.

- DakotaString gradType
  gradient type: none, numerical, analytic, mixed.

- DakotaString methodSrc
  method source: dakota, vendor.

- DakotaString intervalType
  interval type: forward, central.

- Real finiteDiffSS
  relative finite difference step size.

- DakotaIntList idAnalytic
  analytic fn id's for mixed gradients.
6.31.1 Detailed Description

Base class for the model class hierarchy.

The DakotaModel class is the base class for one of the primary class hierarchies in DAKOTA. The model hierarchy contains a set of variables, an interface, and a set of responses, and an iterator operates on the model to map the variables into responses using the interface. For memory efficiency and enhanced polymorphism, the model hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaModel) serves as the envelope and one of the derived classes (selected in DakotaModel::get_model()) serves as the letter.

6.31.2 Constructor & Destructor Documentation

6.31.2.1 DakotaModel::DakotaModel ()

default constructor.

The default constructor is used in Vector<DakotaModel> instantiations and for initialization of DakotaModel objects contained in DakotaStrategy derived classes (see derived strategy header files). modelRep is NULL in this case (a populated problem_db is needed to build a meaningful DakotaModel object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

6.31.2.2 DakotaModel::DakotaModel (ProblemDescDB & problem_db)

standard constructor for envelope.

Used in model instantiations within strategy constructors. Envelope constructor only needs to extract enough data to properly execute get_model, since DakotaModel(BaseConstructor, problem_db) builds the actual base class data for the derived models.

6.31.2.3 DakotaModel::DakotaModel (const DakotaModel & model)

copy constructor.

Copy constructor manages sharing of modelRep and incrementing of referenceCount.

6.31.2.4 DakotaModel::~DakotaModel () [virtual]

destructor.

Destructor decrements referenceCount and only deletes modelRep when referenceCount reaches zero.

6.31.2.5 DakotaModel::DakotaModel (BaseConstructor, ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

This constructor builds the base class data for all inherited models. get_model() instantiates a derived class and the derived class selects this base class constructor in its initialization list (to avoid the recursion of the
base class constructor calling `get_model()` again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaModel).

### 6.31.3 Member Function Documentation

#### 6.31.3.1 DakotaModel DakotaModel::operator= (const DakotaModel & model)

Assignment operator.

#### 6.31.3.2 DakotaString DakotaModel::local_eval_synchronization () [virtual]

return derived model synchronization setting.
SingleModels and HierLayeredModels redefine this virtual function. A default value of "synchronous" prevents async local operations for:

- NestedModels: a subIterator can support message passing parallelism, but not async local. Also, `ProblemDescDB`’s "interface.synchronization" will be bad if no optional interface (will contain last interface spec. parsed).
- SurrLayeredModels: while async evals on approximations will work due to some added bookkeeping, avoiding them is preferable.

Reimplemented in HierLayeredModel.

#### 6.31.3.3 void DakotaModel::init_communicators (const int & max_iterator_concurrency)

allocate communicator partitions for a model.
The `init_communicators()` and `derived_init_communicators()` functions are structured to avoid performing the messageLengths estimation more than once. `init_communicators()` (not virtual) performs the estimation and then forwards the results to `derived_init_communicators` (virtual) which uses the data in different contexts.

#### 6.31.3.4 DakotaModel * DakotaModel::get_model (ProblemDescDB & problem_db) [private]

Used by the envelope to instantiate the correct letter class.
Used only by the envelope constructor to initialize modelRep to the appropriate derived type, as given by the modelType attribute.

#### 6.31.3.5 int DakotaModel::fd_gradients (const DakotaIntArray & map_asv, const DakotaIntArray & fd_grad_asv, const DakotaIntArray & original_asv, const int asynch_flag) [private]

evaluate numerical gradients using finite differences. This routine is selected with "method_source dakota" (the default method_source) in the numerical gradient specification.
Compute finite difference gradients, put the data in currentResponse, and return the number of maps used by fd_gradients. This return value is used by async_compute_response() and synchronize() to track response arrays and it could be used to improve management of max_function_evaluations within the iterators.

6.31.3.6 void DakotaModel::synchronize_fd_gradients (const DakotaResponseArray & fd_grad_responses, DakotaResponse & new_response, const DakotaIntArray & fd_grad_asv, const DakotaIntArray & asv) [private]

combine results from an array of finite difference response objects (fd_grad_responses) into a single response (new_response).
Merge a vector of fd_grad_responses into a single new_response. This function is used both by compute_response() for the case of asynchronous fd_gradients() and by synchronize() for the case where one or more async_compute_response() calls has employed asynchronous fd_gradients().

6.31.3.7 void DakotaModel::update_response (DakotaResponse & new_response, const DakotaIntArray & fd_grad_asv, const DakotaIntArray & asv, const bool initial_map, DakotaRealVector & fn_vals_x0, DakotaRealMatrix & partial_fn_grads, const DakotaRealMatrix & new_fn_grads) [private]

overlay results to update a response object.
Overlay function value and numerical gradient data to populate new_response as governed by initial_map flag and asv vectors. If initial_map occurred, then add to the partial response object created by the map. If initial_map was not used, then only new_fn_grads should be present in the updated new_response. Convenience function used by fd_gradients for the synchronous case and by synchronize_fd_gradients for the asynchronous case.

6.31.3.8 void DakotaModel::manage_asv (const DakotaIntArray & asv_in, DakotaIntArray & map_asv_out, DakotaIntArray & fd_grad_asv_out, int & use_fd_gradients) [private]

Coordinates map() and fd_gradients() calls given an asv_in input.
Splits asv_in total request into map_asv_out for use by map() and fd_grad_asv_out for use by fd_gradients(), as governed by gradient specification.
The documentation for this class was generated from the following files:

- DakotaModel.H
- DakotaModel.C
6.32 DakotaNonD Class Reference

Base class for all nondeterministic iterators (the DAKOTA/UQ branch).

Inheritance diagram for DakotaNonD:

```
DakotaNonD
 |     |
 |     | NonDAdvMeanValue
 |     | NonDSampling
 |     |
 |     | NonDLHSSampling
 |     | NonDPCESampling
```

**Protected Methods**

- **DakotaNonD** (DakotaModel &model)
  
  constructor.

- **DakotaNonD** (NoDBaseConstructor, DakotaModel &model, int num_vars, const DakotaRealVector &lower_bnds, const DakotaRealVector &upper_bnds)
  
  alternate constructor for instantiations "on the fly".

- **~DakotaNonD** ()
  
  destructor.

- void **run_iterator** ()
  
  redefines the main iterator hierarchy virtual function to invoke quantify uncertainty.

- virtual void **quantify_uncertainty** ()=0
  
  performs a forward uncertainty propagation of parameter distributions into response statistics.

- const DakotaResponse & **iterator_response_results** () const
  
  return the final statistics from the nondeterministic iteration.

**Protected Attributes**

- DakotaRealVector **normalMeans**
  
  normal uncertain variable means.

- DakotaRealVector **normalStdDevs**
  
  normal uncertain variable standard deviations.
- DakotaRealVector `normalDistLowerBnds`
  normal uncertain variable distribution lower bounds.

- DakotaRealVector `normalDistUpperBnds`
  normal uncertain variable distribution upper bounds.

- DakotaRealVector `lognormalMeans`
  lognormal uncertain variable means.

- DakotaRealVector `lognormalStdDevs`
  lognormal uncertain variable standard deviations.

- DakotaRealVector `lognormalErrFacts`
  lognormal uncertain variable error factors.

- DakotaRealVector `lognormalDistLowerBnds`
  lognormal uncertain variable distribution lower bounds.

- DakotaRealVector `lognormalDistUpperBnds`
  lognormal uncertain variable distribution upper bounds.

- DakotaRealVector `uniformDistLowerBnds`
  uniform uncertain variable distribution lower bounds.

- DakotaRealVector `uniformDistUpperBnds`
  uniform uncertain variable distribution upper bounds.

- DakotaRealVector `loguniformDistLowerBnds`
  loguniform uncertain variable distribution lower bounds.

- DakotaRealVector `loguniformDistUpperBnds`
  loguniform uncertain variable distribution upper bounds.

- DakotaRealVector `weibullAlphas`
  weibull uncertain variable alphas.

- DakotaRealVector `weibullBetas`
  weibull uncertain variable betas.

- DakotaRealVectorArray `histogramBinPairs`
  histogram uncertain (x,y) bin pairs (continuous linear histogram).

- DakotaRealVectorArray `histogramPointPairs`
  histogram uncertain (x,y) point pairs (discrete histogram).

- DakotaRealMatrix `uncertainCorrelations`
  uncertain variable correlation matrix (rank correlations for sampling and correlation coefficients for analytic reliability).

- size_t `numNormalVars`
number of normal uncertain variables.

- size\_t numLognormalVars
  number of lognormal uncertain variables.

- size\_t numUniformVars
  number of uniform uncertain variables.

- size\_t numLoguniformVars
  number of loguniform uncertain variables.

- size\_t numWeibullVars
  number of weibull uncertain variables.

- size\_t numHistogramVars
  number of histogram uncertain variables.

- size\_t numUncertainVars
  total number of uncertain variables.

- size\_t numResponseFunctions
  number of response functions.

- DakotaRealVector meanStats
  means of response functions calculated in compute\_statistics().

- DakotaRealVector mean95CIDeltas
  Plus/minus deltas on response function means for 95% confidence intervals (calculated in compute\_statistics()).

- DakotaRealVector stdDevStats
  std deviations of response functions (calculated in compute\_statistics()).

- DakotaRealArray respThresh
  response thresholds for computing failure probabilities.

- DakotaRealVector probMoreThanThresh
  probabilities that response functions are greater than respThresh (calculated in compute\_statistics()).

- DakotaResponse finalStatistics
  final statistics from the uncertainty propagation used in strategies: response means, standard deviations, and probabilities of failure.

- bool correlationFlag
  flag for indicating if correlation exists among the uncertain variables.
6.32.1 Detailed Description

Base class for all nondeterministic iterators (the DAKOTA/UQ branch).

The base class for nondeterministic iterators consolidates uncertain variable data and probabilistic utilities for inherited classes.

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

- DakotaNonD.H
- DakotaNonD.C
6.33 DakotaOptimizer Class Reference

Base class for the optimizer branch of the iterator hierarchy.

Inheritance diagram for DakotaOptimizer::

```
DakotaOptimizer
    |           |           |           |
    |           |           |           |
    |           |           |           |
    |           |           |           |
    |           |           |           |
    DakotaOptLeastSq
    DakotaIterator
    COLINOptimizer
    CONMINOptimizer
    DOTOptimizer
    NPSOLOptimizer
    rSQPOptimizer
    SGLOPTOptimizer
    SNLLOptimizer
```

Public Methods

- void run_iterator ()
  
  *run the iterator.*

- void print_iterator_results (ostream &s) const

- void multi_objective_weights (const DakotaRealVector &multi_obj_wts)
  
  *set the relative weightings for multiple objective functions. Used by ConcurrentStrategy for Pareto set optimization.*

Protected Methods

- DakotaOptimizer ()
  
  *default constructor.*

- DakotaOptimizer (DakotaModel &model)
  
  *standard constructor.*

- ~DakotaOptimizer ()
  
  *destructor.*

- virtual void find_optimum ()=0
  
  *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

Static Protected Methods

- DakotaResponse multi_objective_modify (const DakotaResponse &raw_response)
  
  *maps multiple objective functions to a single objective for single-objective optimizers (static for use within NPSOL and OPT++ evaluator functions).*
Protected Attributes

- size_t numObjectiveFunctions
  number of objective functions.

- DakotaRealVector multiObjWeights
  user-specified weights for multiple objective functions.

Static Protected Attributes

- size_t staticNumObjFns
  static copy of numObjectiveFunctions used in static functions passed by function pointer (NPSOL and OPT++).

- DakotaRealVector staticMultiObjWeights
  static copy of multiObjWeights for use in multi_objective_modify().

6.33.1 Detailed Description

Base class for the optimizer branch of the iterator hierarchy.
The DakotaOptimizer class provides common data and functionality for DOTOptimizer, NPSOLOptimizer, SNLLOptimizer, and SGOPTOptimizer.

6.33.2 Constructor & Destructor Documentation

6.33.2.1 DakotaOptimizer::DakotaOptimizer (DakotaModel & model) [protected]

standard constructor.
This constructor extracts the inherited data for the optimizer branch and performs sanity checking on gradient and constraint settings.

6.33.3 Member Function Documentation

6.33.3.1 void DakotaOptimizer::run_iterator () [virtual]

run the iterator.
This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.
Reimplemented from DakotaIterator.
6.33.3.2  void DakotaOptimizer::print_iterator_results (ostream & s) const  [virtual]

Redefines default iterator results printing to include optimization results (objective function and constraints).
Reimplemented from DakotaIterator.

6.33.3.3  DakotaResponse DakotaOptimizer::multi_objective_modify (const DakotaResponse & raw_response)  [static, protected]

maps multiple objective functions to a single objective for single-objective optimizers (static for use within NPSOL and OPT++ evaluator functions).
This function is responsible for the mapping of multiple objective functions into a single objective for publishing to single-objective optimizers. Used in DOTOptimizer, NPSOLOptimizer, SNLLOptimizer, and SGOPTApplication on every function evaluation. The simple weighting approach (using staticMulti-ObjWeights) is the only technique supported currently. The weightings are used to scale function values, gradients, and Hessians as needed.
The documentation for this class was generated from the following files:

- DakotaOptimizer.H
- DakotaOptimizer.C
6.34 DakotaOptLeastSq Class Reference

Base class for the optimizer and least squares branches of the iterator hierarchy.

Inheritance diagram for DakotaOptLeastSq:

```
DakotaIterator
    \------\------
   |        |
   | DakotaOptLeastSq |
   |                  |
   | DakotaLeastSq |
   |               |
   |               |
   | NLSSOLLeastSq |
   |              |
   |               |
   |               |
   | SNLLLeastSq |
   |            |
   |            |
   | COLINOptimizer |
   |               |
   | CONMINOptimizer |
   |                |
   | DOTOptimizer |
   |              |
   | NPSOLOptimizer |
   |               |
   | rSQPOptimizer |
   |              |
   | SGOPTOptimizer |
   |               |
   | SNLLOptimizer |
```

Public Methods

- **const DakotaVariables & iterator_variable_results () const**
  
  *return the final iterator solution (variables).*

- **const DakotaResponse & iterator_response_results () const**
  
  *return the final iterator solution (response).*

Protected Methods

- **DakotaOptLeastSq ()**
  
  *default constructor.*

- **DakotaOptLeastSq (DakotaModel &model)**
  
  *standard constructor.*
• ~\texttt{DakotaOptLeastSq()} 
  \texttt{destructor}.

\textbf{Protected Attributes}

• \texttt{Real convergenceTol}
  \texttt{optimizer/least squares convergence tolerance.}

• \texttt{Real constraintTol}
  \texttt{optimizer/least squares constraint tolerance.}

• \texttt{size_t numNonlinearIneqConstraints}
  \texttt{number of nonlinear ineq constraints.}

• \texttt{DakotaRealVector nonlinearIneqLowerBnds}
  \texttt{nonlinear ineq constraint lower bounds.}

• \texttt{DakotaRealVector nonlinearIneqUpperBnds}
  \texttt{nonlinear ineq constraint upper bounds.}

• \texttt{Real bigRealBoundSize}
  \texttt{cutoff value for ineq constraint and continuous variable bounds.}

• \texttt{int bigIntBoundSize}
  \texttt{cutoff value for discrete variable bounds.}

• \texttt{size_t numNonlinearEqConstraints}
  \texttt{number of nonlinear eq constraints.}

• \texttt{DakotaRealVector nonlinearEqTargets}
  \texttt{nonlinear eq constraint targets.}

• \texttt{int numNonlinearConstraints}
  \texttt{total number of nonlinear constraints.}

• \texttt{int numConstraints}
  \texttt{total number of linear and nonlinear constraints (for DOT/CONMIN).}

• \texttt{size_t numLinearIneqConstraints}
  \texttt{number of linear ineq constraints.}

• \texttt{DakotaRealMatrix linearIneqConstraintCoeffs}
  \texttt{linear ineq constraint coefficients.}

• \texttt{DakotaRealVector linearIneqLowerBnds}
  \texttt{linear ineq constraint lower bounds.}
```markdown
- DakotaRealVector `linearIneqUpperBnds`  
  linear inequality constraint upper bounds.

- size_t `numLinearEqConstraints`  
  number of linear equality constraints.

- DakotaRealMatrix `linearEqConstraintCoeffs`  
  linear equality constraint coefficients.

- DakotaRealVector `linearEqTargets`  
  linear equality constraint targets.

- int `numLinearConstraints`  
  total number of linear constraints.

- bool `boundConstraintFlag`  
  convenience flag for denoting the presence of user-specified bound constraints. Used for method selection and error checking.

- bool `speculativeFlag`  
  flag for speculative gradient evaluations.

- bool `vendorNumericalGradFlag`  
  convenience flag for gradType=="numerical" && methodSource=="vendor".

- DakotaVariables `bestVariables`  
  best variables found in solution.

- DakotaResponse `bestResponses`  
  best responses found in solution.

### Static Protected Attributes

- size_t `staticNumContinuousVars`  
  static copy of numContinuousVars used in static functions passed by function pointer (NPSOL, NLSSOL, and OPT++) and in DakotaOptimizer::multi_objective_modify().

- size_t `staticNumNonlinearConstraints`  
  static copy of numNonlinearConstraints used in static functions passed by function pointer (NPSOL, NLSSOL, and OPT++) and in DakotaOptimizer::multi_objective_modify().

### 6.34.1 Detailed Description

Base class for the optimizer and least squares branches of the iterator hierarchy.

The DakotaOptLeastSq class provides common data and functionality for DakotaOptimizer and DakotaLeastSq.
```
6.34.2 Constructor & Destructor Documentation

6.34.2.1 DakotaOptLeastSq::DakotaOptLeastSq (DakotaModel & model) [protected]

standard constructor.

This constructor extracts inherited data for the optimizer and least squares branches and performs sanity checking on constraint settings.

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

- DakotaOptLeastSq.H
- DakotaOptLeastSq.C
6.35 DakotaResponse Class Reference

Container class for response functions and their derivatives. DakotaResponse provides the handle class.

Public Methods

- **DakotaResponse ()**
  
  *default constructor.*

- **DakotaResponse (int num_params, const ProblemDescDB &problem_db)**
  
  *standard constructor built from problem description database.*

- **DakotaResponse (int num_params, const DakotaIntArray &asv)**
  
  *alternate constructor using limited data.*

- **DakotaResponse (const DakotaResponse &response)**
  
  *copy constructor.*

- **~DakotaResponse ()**
  
  *destructor.*

- **DakotaResponse operator= (const DakotaResponse &response)**
  
  *assignment operator.*

- **size_t num_functions () const**
  
  *return the number of response functions.*

- **const DakotaIntArray & active_set_vector () const**
  
  *return the active set vector.*

- **void active_set_vector (const DakotaIntArray &asv)**
  
  *set the active set vector.*

- **const DakotaString & interface_id () const**
  
  *return the interface identifier.*

- **void interface_id (const DakotaString &id)**
  
  *set the interface identifier.*

- **const DakotaStringArray & fn_tags () const**
  
  *return the function identifier strings.*

- **void fn_tags (const DakotaStringArray &tags)**
  
  *set the function identifier strings.*

- **const DakotaRealVector & function_values () const**
return the function values.

- **void** function\_values (const DakotaRealVector &function\_vals)
  set the function values.

- **const** DakotaRealMatrix & function\_gradients () const
  return the function gradients.

- **void** function\_gradients (const DakotaRealMatrix &function\_grads)
  set the function gradients.

- **const** DakotaRealMatrixArray & function\_hessians () const
  return the function Hessians.

- **void** function\_hessians (const DakotaRealMatrixArray &function\_hessians)
  set the function Hessians.

- **void** read (istream &s)
  read a response object from an istream.

- **void** write (ostream &s) const
  write a response object to an ostream.

- **void** read\_annotated (istream &s)
  read a response object in annotated format from an istream.

- **void** write\_annotated (ostream &s) const
  write a response object in annotated format to an ostream.

- **void** read\_tabular (istream &s)
  read responseRep::functionValues in tabular format from an istream.

- **void** write\_tabular (ostream &s) const
  write responseRep::functionValues in tabular format to an ostream.

- **void** read (DakotaBiStream &s)
  read a response object from the binary restart stream.

- **void** write (DakotaBoStream &s) const
  write a response object to the binary restart stream.

- **void** read (UnPackBuffer &s)
  read a response object from a packed MPI buffer.

- **void** write (PackBuffer &s) const
  write a response object to a packed MPI buffer.

- **DakotaResponse** copy () const
  a deep copy for use in history mechanisms.
• int data_size ()
  handle class forward to corresponding body class member function.

• void read_data (double *response_data)
  handle class forward to corresponding body class member function.

• void write_data (double *response_data)
  handle class forward to corresponding body class member function.

• void overlay (const DakotaResponse &response)
  handle class forward to corresponding body class member function.

• void copy_results (const DakotaResponse &response)
  handle class forward to corresponding body class member function.

• void purge_inactive ()
  handle class forward to corresponding body class member function.

• void reset ()
  handle class forward to corresponding body class member function.

Private Attributes

• DakotaResponseRep * responseRep
  pointer to the body (handle-body idiom).

Friends

• bool operator== (const DakotaResponse &resp1, const DakotaResponse &resp2)
  equality operator.

• bool operator!= (const DakotaResponse &resp1, const DakotaResponse &resp2)
  inequality operator.

6.35.1 Detailed Description

Container class for response functions and their derivatives. DakotaResponse provides the handle class.

The DakotaResponse class is a container class for an abstract set of functions (functionValues) and their first (functionGradients) and second (functionHessians) derivatives. The functions may involve objective and constraint functions (optimization data set), least squares terms (parameter estimation data set), or generic response functions (uncertainty quantification data set). It is not currently part of a class hierarchy, since the abstraction has been sufficiently general and has not required specialization. For memory efficiency, it employs the “handle-body idiom” approach to reference counting and representation sharing (see Coplien "Advanced C++", p. 58), for which DakotaResponse serves as the handle and DakotaResponseRep serves as the body.
6.35 DakotaResponse Class Reference

6.35.2 Constructor & Destructor Documentation

6.35.2.1 DakotaResponse::DakotaResponse ()

default constructor.

Need a populated problem description database to build a meaningful DakotaResponse object, so set the
responseRep=NULL in default constructor for efficiency. This then requires a check on NULL in the copy
constructor, assignment operator, and destructor.

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

- DakotaResponse.H
- DakotaResponse.C
6.36 DakotaResponseRep Class Reference

Container class for response functions and their derivatives. DakotaResponseRep provides the body class.

Private Methods

- **DakotaResponseRep ()**
  default constructor.

- **DakotaResponseRep (int num_params, const ProblemDescDB &problem_db)**
  standard constructor built from problem description database.

- **DakotaResponseRep (int num_params, const DakotaIntArray &asv)**
  alternate constructor using limited data.

- **~DakotaResponseRep ()**
  destructor.

- **void read (istream &s)**
  read a responseRep object from an istream.

- **void write (ostream &s) const**
  write a responseRep object to an ostream.

- **void read_annotated (istream &s)**
  read a responseRep object from an istream (annotated format).

- **void write_annotated (ostream &s) const**
  write a responseRep object to an ostream (annotated format).

- **void read_tabular (istream &s)**
  read functionValues from an istream (tabular format).

- **void write_tabular (ostream &s) const**
  write functionValues to an ostream (tabular format).

- **void read (DakotaBiStream &s)**
  read a responseRep object from a binary stream.

- **void write (DakotaBoStream &s) const**
  write a responseRep object to a binary stream.

- **void read (UnPackBuffer &s)**
  read a responseRep object from a packed MPI buffer.

- **void write (PackBuffer &s) const**
write a responseRep object to a packed MPI buffer.

- **int data_size ()**
  return the number of doubles active in response. Used for sizing double* response data arrays passed into read_data and write_data.

- **void read_data (double *response_data)**
  read from an incoming double* array.

- **void write_data (double *response_data)**
  write to an incoming double* array.

- **void overlay (const DakotaResponse &response)**
  add incoming response to functionValues/Gradients/Hessians.

- **void copy_results (const DakotaResponse &response)**
  copy functionValues, functionGradients, & functionHessians data only. Do not copy ASV, tags, id’s, etc. Used in place of assignment operator for retrieving results data from the data_pairs list without corrupting other data.

- **void purge_inactive ()**
  Purge extraneous data from the response object (used when a response object is returned from the database pair with more data than needed by the search pair ASV (see ApplicationInterface::map and DakotaModel::fd_gradients).

- **void reset ()**
  resets functionValues, functionGradients, and functionHessians to zero.

**Private Attributes**

- **int referenceCount**
  number of handle objects sharing responseRep.

- **DakotaRealVector functionValues**
  abstract set of functions.

- **DakotaRealMatrix functionGradients**
  first derivatives.

- **DakotaRealMatrixArray functionHessians**
  second derivatives.

- **DakotaIntArray responseASV**
  Copy of DakotaIterator’s activeSetVector needed for operator overloaded I/O.

- **DakotaStringArray fnTags**
  function identifiers used to improve output readability.

- **DakotaString interfaceId**
  the interface used to generate this response object. Used in PRPair::vars_asv_compare.
Friends

- class DakotaResponse
  
  the handle class can access attributes of the body class directly.

- bool operator==(const DakotaResponseRep &rep1, const DakotaResponseRep &rep2)
  
  equality operator.

6.36.1 Detailed Description

Container class for response functions and their derivatives. DakotaResponseRep provides the body class. The DakotaResponseRep class is the "representation" of the response container class. It is the "body" portion of the "handle-body idiom" (see Coplien "Advanced C++", p. 58). The handle class (DakotaResponse) provides for memory efficiency in management of multiple response objects through reference counting and representation sharing. The body class (DakotaResponseRep) actually contains the response data (functionValues, functionGradients, functionHessians, etc.). The representation is hidden in that an instance of DakotaResponseRep may only be created by DakotaResponse. Therefore, programmers create instances of the DakotaResponse handle class, and only need to be aware of the handle/body mechanisms when it comes to managing shallow copies (shared representation) versus deep copies (separate representation used for history mechanisms).

6.36.2 Constructor & Destructor Documentation

6.36.2.1 DakotaResponseRep::DakotaResponseRep (int num_params, const ProblemDescDB & problem_db) [private]

standard constructor built from problem description database.

The standard constructor used by DakotaModelRep. An interfaceId identifies a set of results with the interface used in generating them, which allows vars_asv_compare to prevent duplicate detection on results from different interfaces.

6.36.2.2 DakotaResponseRep::DakotaResponseRep (int num_params, const DakotaIntArray & asv) [private]

alternate constructor using limited data.

Used for building a response object of the correct size on the fly (e.g., by slave analysis servers performing execute() on a local_response). fnTags and interfaceId are not needed for this purpose since they’re not passed in the MPI send/recv buffers (NOTE: if interfaceId becomes needed, it could be set from an AppInt attribute passed from AppInt::serve()). However, NPSOLOptimizer’s user-defined functions option uses this constructor to build bestResponses and bestResponses needs fnTags for I/O, so construction of fnTags has been added.

6.36.3 Member Function Documentation
6.36.3.1 void DakotaResponseRep::read (istream & s) [private]

read a responseRep object from an istream.
ASCII version of read needs capabilities for capturing data omissions or formatting errors (resulting from user error or async race condition) and analysis failures (resulting from nonconvergence, instability, etc.).

6.36.3.2 void DakotaResponseRep::write (ostream & s) const [private]

write a responseRep object to an ostream.
ASCII version of write.

6.36.3.3 void DakotaResponseRep::read_annotated (istream & s) [private]

read a responseRep object from an istream (annotated format).
read_annotated version is used for neutral file translation of restart files. Since objects are built solely from this data, annotations are used. This version is currently identical to the DakotaBiStream version.

6.36.3.4 void DakotaResponseRep::write_annotated (ostream & s) const [private]

write a responseRep object to an ostream (annotated format).
write_annotated version is used for neutral file translation of restart files. Since objects need to be build solely from this data, annotations are used. This version differs from the DakotaBoStream version only in the use of white space between fields.

6.36.3.5 void DakotaResponseRep::read_tabular (istream & s) [private]

read functionValues from an istream (tabular format).
read_tabular is used to read functionValues in tabular format. It is currently only used by Approximation-Interfaces in reading samples from a file. There is insufficient data in a tabular file to build complete response objects; rather, the response object must be constructed a priori and then its functionValues can be set.

6.36.3.6 void DakotaResponseRep::write_tabular (ostream & s) const [private]

write functionValues to an ostream (tabular format).
write_tabular is used for output of functionValues in a tabular format for convenience in post-processing/plotting of DAKOTA results.

6.36.3.7 void DakotaResponseRep::read (DakotaBiStream & s) [private]

read a responseRep object from a binary stream.
Binary version differs from ASCII version in 2 primary ways: (1) it lacks formatting. (2) the DakotaResponse has not been sized a priori. In reading data from the binary restart file, a ParamResponsePair was constructed with its default constructor which called the DakotaResponse default constructor. Therefore, we must first read sizing data and resize the arrays.
6.36.3.8  **void DakotaResponseRep::write(DakotaBoStream & s) const**  [private]

write a responseRep object to a binary stream.

Binary version differs from ASCII version in 2 primary ways: (1) It lacks formatting. (2) In reading data from the binary restart file, ParamResponsePairs are constructed with their default constructor which calls the DakotaResponse default constructor. Therefore, we must first write sizing data so that DakotaResponseRep::read(DakotaBoStream& s) can resize the arrays.

6.36.3.9  **void DakotaResponseRep::read(UnPackBuffer & s)**  [private]

read a responseRep object from a packed MPI buffer.

UnpackBuffer version differs from DakotaBiStream version only in omission of interfaceId and default fnTags. Master processor retains tags and ids and communicates asv and response data only with slaves.

6.36.3.10  **void DakotaResponseRep::write(PackBuffer & s) const**  [private]

write a responseRep object to a packed MPI buffer.

PackBuffer version differs from DakotaBoStream version only in omissions of interfaceId and flush. The master processor retains tags and ids and communicates asv and response data only with slaves.

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

- DakotaResponse.H
- DakotaResponse.C
6.37 DakotaStrategy Class Reference

Base class for the strategy class hierarchy.

Inheritance diagram for DakotaStrategy:

```
DakotaStrategy
     |     |
```

Public Methods

- **DakotaStrategy** ()
  
  *default constructor.*

- **DakotaStrategy** (ProblemDescDB &problem_db)
  
  *constructor.*

- **DakotaStrategy** (const DakotaStrategy &strat)
  
  *copy constructor.*

- virtual **~DakotaStrategy** ()
  
  *destructor.*

- DakotaStrategy operator= (const DakotaStrategy &strat)
  
  *assignment operator.*

- virtual void **run_strategy** ()
  
  *the run function for the strategy: invoke the iterator(s) on the model(s). Called from main.C.*

- virtual const DakotaVariables & strategy_variable_results () const
  
  *return the final strategy solution (variables).*

- virtual const DakotaResponse & strategy_response_results () const
  
  *return the final strategy solution (response).*

- void **run_iterator** (DakotaIterator &the_iterator, DakotaModel &the_model)
  
  *Convenience function for invoking an iterator and managing parallelism. This version omits communicator repartitioning. Function must be public due to use by MINLPNode.*

- int **world_rank** () const
  
  *return worldRank (used only by MINLPNode).*

- MPI_Comm **iterator_communicator** () const
  
  *return iteratorComm (used only by MINLPNode).*
- int iterator_communicator_size () const
  return iteratorCommSize (used only by MINLPNode).

Protected Methods

- DakotaStrategy (BaseConstructor, ProblemDescDB &problem_db)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

- void run_iterator_repartition (DakotaIterator &the_iterator, DakotaModel &the_model)
  Convenience function for invoking an iterator and managing parallelism. This version repartitions communicators.

- void init_communicators (DakotaIterator &the_iterator, DakotaModel &the_model)
  convenience function for allocating comms prior to running an iterator.

- void free_communicators (DakotaModel &the_model)
  convenience function for deallocating comms after running an iterator.

- void initialize_graphics (const DakotaModel &model)
  convenience function for initialization of 2D graphics and data tabulation.

Protected Attributes

- ProblemDescDB & probDescDB
  class member reference to the problem description database.

- ParallelLibrary & parallelLib
  class member reference to the parallel library.

- DakotaString strategyName
  type of strategy: single_method, multi_level, surrogate_based_opt, opt_under_uncertainty, branch_and_bound, multi_start, or pareto_set.

- int worldRank
  processor rank in MPI_COMM_WORLD.

- int worldSize
  size of MPI_COMM_WORLD.

- MPI_Comm iteratorComm
  the communicator defining the group of processors on which an iterator executes. Results from init_iterator_commoms.

- int iteratorCommRank
  processor rank in iteratorComm.

- int iteratorCommSize
number of processors in iteratorComm.

- bool mpirunFlag
  flag for parallel MPI launch of DAKOTA.

- bool graphicsFlag
  flag for using graphics in a graphics executable.

- bool tabularDataFlag
  flag for file tabulation of graphics data.

- DakotaString tabularDataFile
  filename for tabulation of graphics data.

Private Methods

- DakotaStrategy * get_strategy (ProblemDescDB &problem_db)
  Used by the envelope to instantiate the correct letter class.

- ProblemDescDB & prob_desc_db () const
  returns the problem description database (probDescDB).

Private Attributes

- DakotaStrategy * strategyRep
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  number of objects sharing strategyRep.

6.37.1 Detailed Description

Base class for the strategy class hierarchy.

The DakotaStrategy class is the base class for the class hierarchy providing the top level control in DAKOTA. The strategy is responsible for creating and managing iterators and models. For memory efficiency and enhanced polymorphism, the strategy hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaStrategy) serves as the envelope and one of the derived classes (selected in DakotaStrategy::get_strategy()) serves as the letter.

6.37.2 Constructor & Destructor Documentation
6.37.2.1 DakotaStrategy::DakotaStrategy ()

default constructor.

The default constructor is used in SIERRA procedure classes. strategyRep is NULL in this case (a populated problem.db is needed to build a meaningful DakotaStrategy object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

6.37.2.2 DakotaStrategy::DakotaStrategy (ProblemDescDB & problem_db)

constructor.

Used in main.C instantiation to build the envelope. This constructor only needs to extract enough data to properly execute get_strategy, since DakotaStrategy::DakotaStrategy(BaseConstructor, problem_db) builds the actual base class data inherited by the derived strategies.

6.37.2.3 DakotaStrategy::DakotaStrategy (const DakotaStrategy & strat)

copy constructor.

Copy constructor manages sharing of strategyRep and incrementing of referenceCount.

6.37.2.4 DakotaStrategy::~DakotaStrategy () [virtual]

destructor.

Destructor decrements referenceCount and only deletes strategyRep when referenceCount reaches zero.

6.37.2.5 DakotaStrategy::DakotaStrategy (BaseConstructor, ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

This constructor is the one which must build the base class data for all inherited strategies. get_strategy() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_strategy() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaStrategy).

6.37.3 Member Function Documentation

6.37.3.1 DakotaStrategy DakotaStrategy::operator= (const DakotaStrategy & strat)

assignment operator.

6.37.3.2  void DakotaStrategy::run_iterator (DakotaIterator & the_iterator, DakotaModel & the_model)

Convenience function for invoking an iterator and managing parallelism. This version omits communicator repartitioning. Function must be public due to use by MINLPNode.

This is a convenience function for encapsulating the parallel features (run/serve) of running an iterator. This function omits allocation/deallocation of communicators to provide greater efficiency in those strategies which involve multiple iterator executions but only require communicator allocation/deallocation to be performed once.

It does not require a strategyRep forward since it is only used by letter objects. While it is currently a public function due to its use in MINLPNode, this usage still involves a strategy letter object.

6.37.3.3  void DakotaStrategy::run_iterator_repartition (DakotaIterator & the_iterator, DakotaModel & the_model) [protected]

Convenience function for invoking an iterator and managing parallelism. This version repartitions communicators.

This is a convenience function for encapsulating the parallel features (init/run/serve/free) of running an iterator. This function includes allocation/deallocation of communicators as part of each iterator invocation. Reallocating comms for each run_iterator_repartition() call can be wasteful if little is changing (e.g., BranchBndStrategy, ConcurrentStrategy). In these cases, use run_iterator() instead. This function does not require a strategyRep forward since it is only used by letter objects.

6.37.3.4  void DakotaStrategy::init_communicators (DakotaIterator & the_iterator, DakotaModel & the_model) [protected]

close convenience function for allocating comms prior to running an iterator.

This is a convenience function for encapsulating the allocation of communicators prior to running an iterator. It does not require a strategyRep forward since it is only used by letter objects.

6.37.3.5  void DakotaStrategy::free_communicators (DakotaModel & the_model) [protected]

convenience function for deallocating comms after running an iterator.

This is a convenience function for encapsulating the deallocation of communicators after running an iterator. It does not require a strategyRep forward since it is only used by letter objects.

6.37.3.6  void DakotaStrategy::initialize_graphics (const DakotaModel & model) [protected]

close convenience function for initialization of 2D graphics and data tabulation.

This is a convenience function for encapsulating graphics initialization operations. It does not require a strategyRep forward since it is only used by letter objects.

6.37.3.7  DakotaStrategy * DakotaStrategy::get_strategy (ProblemDescDB & problem_db) [private]

Used by the envelope to instantiate the correct letter class.
Used only by the envelope constructor to initialize strategyRep to the appropriate derived type, as given by the strategyName attribute.

6.37.3.8 `ProblemDescDB & DakotaStrategy::prob_desc_db () const [inline, private]`

returns the problem description database (probDescDB).
Used only by the copy constructor (otherwise strategyRep forward needed).
The documentation for this class was generated from the following files:

- DakotaStrategy.H
- DakotaStrategy.C
### DakotaString Class Reference

DakotaString class, used as main string class for Dakota.

#### Public Methods

- **DakotaString ()**
  - Default constructor.

- **DakotaString (const DakotaString &a)**
  - Default copy constructor.

- **DakotaString (const char *initial_val)**
  - Copy constructor from standard C char array.

- **~DakotaString ()**
  - Destructor.

- **DakotaString & operator= (const DakotaString &)**
  - Normal assignment operator.

- **DakotaString & operator= (const DAKOTA_BASE_STRING &)**
  - Assignment operator for base string.

- **DakotaString & operator= (const char *)**
  - Assignment operator, standard C char*.

- **operator const char * () const**
  - The operator() returns pointer to standard C char array.

- **DakotaString & toUpper ()**
  - Convert to upper case string.

- **void upper ()**
- **DakotaString & toLower ()**
  - Convert to lower case string.

- **void lower ()**
- **bool contains (const char *subString) const**
  - Returns true if DakotaString contains char* substring.

- **bool isNull () const**
  - Returns true of DakotaString is empty.

- **char * data () const**
  - Returns pointer to standard C char array.
• void testClass ()
  
  Class unit test method.

6.38.1 Detailed Description

DakotaString class, used as main string class for Dakota.
The DakotaString class is the common string class for Dakota. It provides a common interface for string operations whether inheriting from the STL basic_string or the Rogue Wave RWCString class.

6.38.2 Member Function Documentation

6.38.2.1 DakotaString::operator const char * () const [inline]

The operator() returns pointer to standard C char array.
The operator() returns a pointer to a char string. Uses the STL c_str() method. This allows for the Dakota-String to be used in method calls without having to call the data() or c_str() methods.

6.38.2.2 void DakotaString::upper ()

Private method which converts DakotaString to upper. Utilizes a STL iterator to step through the string and then calls the STL toupper() method. Needs to be done this way because STL only provides a single char toupper method.

6.38.2.3 void DakotaString::lower ()

Private method which converts DakotaString to lower. Utilizes a STL iterator to step through the string and then calls the STL tolower() method. Needs to be done this way because STL only provides a single char tolower method.

6.38.2.4 bool DakotaString::contains (const char * subString) const [inline]

Returns true if DakotaString contains char* substring.
Returns true of the DakotaString contains the char* substring. Calls the STL rfind() method, then checks if substring was found within the DakotaString.

6.38.2.5 char * DakotaString::data () const [inline]

Returns pointer to standard C char array.
Returns a pointer to c style char array. Needed to mimic the Rogue Wave string class. USE WITH CARE.
6.38 DakotaString Class Reference

6.38.2.6 void DakotaString::testClass()

Class unit test method.

Unit test method for the DakotaString class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will abort if an unexpected answer is received.

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

- DakotaString.H
- DakotaString.C
6.39 DakotaVarConstraints Class Reference

Base class for the variable constraints class hierarchy.

Inheritance diagram for DakotaVarConstraints:

```
DakotaVarConstraints
  └── AllMergedVarConstraints
      └── AllVarConstraints
          └── FundamentalVarConstraints
              └── MergedVarConstraints
```

Public Methods

- **DakotaVarConstraints ()**
  
  *default constructor.*

- **DakotaVarConstraints (const ProblemDescDB &problem_db, const DakotaString &vars_type)**
  
  *standard constructor.*

- **DakotaVarConstraints (const DakotaVarConstraints &vc)**
  
  *copy constructor.*

- virtual **~DakotaVarConstraints ()**
  
  *destructor.*

- **DakotaVarConstraints operator= (const DakotaVarConstraints &vc)**
  
  *assignment operator.*

- virtual const DakotaRealVector & **continuous_lower_bounds ()** const
  
  *return the active continuous variable lower bounds.*

- virtual void **continuous_lower_bounds (const DakotaRealVector &c_lbnds)**
  
  *set the active continuous variable lower bounds.*

- virtual const DakotaRealVector & **continuous_upper_bounds ()** const
  
  *return the active continuous variable upper bounds.*

- virtual void **continuous_upper_bounds (const DakotaRealVector &c_ubnds)**
  
  *set the active continuous variable upper bounds.*

- virtual const DakotaIntVector & **discrete_lower_bounds ()** const
  
  *return the active discrete variable lower bounds.*

- virtual void **discrete_lower_bounds (const DakotaIntVector &d_lbnds)**
  
  *set the active discrete variable lower bounds.*

- virtual void **discrete_upper_bounds (const DakotaIntVector &d_ubnds)**
  
  *set the active discrete variable upper bounds.*
virtual const DakotaIntVector & discrete_upper_bounds () const
return the active discrete variable upper bounds.

virtual void discrete_upper_bounds (const DakotaIntVector &d_u_bnds)
set the active discrete variable upper bounds.

virtual const DakotaRealVector & inactive_continuous_lower_bounds () const
return the inactive continuous lower bounds.

virtual void inactive_continuous_lower_bounds (const DakotaRealVector &i_c_l_bnds)
set the inactive continuous lower bounds.

virtual const DakotaRealVector & inactive_continuous_upper_bounds () const
return the inactive continuous upper bounds.

virtual void inactive_continuous_upper_bounds (const DakotaRealVector &i_c_u_bnds)
set the inactive continuous upper bounds.

virtual const DakotaIntVector & inactive_discrete_lower_bounds () const
return the inactive discrete lower bounds.

virtual void inactive_discrete_lower_bounds (const DakotaIntVector &i_d_l_bnds)
set the inactive discrete lower bounds.

virtual const DakotaIntVector & inactive_discrete_upper_bounds () const
return the inactive discrete upper bounds.

virtual void inactive_discrete_upper_bounds (const DakotaIntVector &i_d_u_bnds)
set the inactive discrete upper bounds.

virtual DakotaRealVector all_continuous_lower_bounds () const
returns a single array with all continuous lower bounds.

virtual DakotaRealVector all_continuous_upper_bounds () const
returns a single array with all continuous upper bounds.

virtual DakotaIntVector all_discrete_lower_bounds () const
returns a single array with all discrete lower bounds.

virtual DakotaIntVector all_discrete_upper_bounds () const
returns a single array with all discrete upper bounds.

virtual void write (ostream &s) const
write a variable constraints object to an ostream.

virtual void read (istream &s)
read a variable constraints object from an istream.

size_t num_linear_ineq_constraints () const
return the number of linear inequality constraints.
• size_t num_linear_eq_constraints () const
  return the number of linear equality constraints.

• const DakotaRealMatrix & linear_ineq_constraint_coeffs () const
  return the linear inequality constraint coefficients.

• const DakotaRealVector & linear_ineq_constraint_lower_bounds () const
  return the linear inequality constraint lower bounds.

• const DakotaRealVector & linear_ineq_constraint_upper_bounds () const
  return the linear inequality constraint upper bounds.

• const DakotaRealMatrix & linear_eq_constraint_coeffs () const
  return the linear equality constraint coefficients.

• const DakotaRealVector & linear_eq_constraint_targets () const
  return the linear equality constraint targets.

Protected Methods

• DakotaVarConstraints (BaseConstructor, const ProblemDescDB &problem_db)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

• void manage_linear_constraints (const ProblemDescDB &problem_db, const size_t &num_vars)
  perform checks on user input, convert linear constraint coefficient input to matrices, and assign defaults.

• size_t num_active_variables () const
  return number of active variables.

Protected Attributes

• DakotaString variablesType
  All, Merged, AllMerged, or Fundamental.

• bool discreteFlag
  flags discrete variable mode.

• size_t numLinearIneqConstraints
  number of linear inequality constraints.

• size_t numLinearEqConstraints
  number of linear equality constraints.

• DakotaRealMatrix linearIneqConstraintCoeffs
  linear inequality constraint coefficients.
- DakotaRealMatrix *linearEqConstraintCoeffs
  "linear equality constraint coefficients."

- DakotaRealVector *linearIneqConstraintLowerBnds
  "linear inequality constraint lower bounds."

- DakotaRealVector *linearIneqConstraintUpperBnds
  "linear inequality constraint upper bounds."

- DakotaRealVector *linearEqConstraintTargets
  "linear equality constraint targets."

- DakotaRealVector *emptyRealVector
  "an empty real vector returned in get functions when there are no variable constraints corresponding to the request."

- DakotaIntVector *emptyIntVector
  "an empty int vector returned in get functions when there are no variable constraints corresponding to the request."

**Private Methods**

- DakotaVarConstraints *get_var_constraints (const ProblemDescDB &problem_db)
  "Used only by the constructor to initialize varConstraintsRep to the appropriate derived type."

**Private Attributes**

- DakotaVarConstraints * varConstraintsRep
  "pointer to the letter (initialized only for the envelope)."

- int referenceCount
  "number of objects sharing varConstraintsRep."

### 6.39.1 Detailed Description

Base class for the variable constraints class hierarchy.

The DakotaVarConstraints class is the base class for the class hierarchy managing linear and bound constraints on the variables. Using the variable lower and upper bounds arrays and linear constraint coefficients and bounds from the input specification, different derived classes define different views of this data. For memory efficiency and enhanced polymorphism, the variable constraints hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaVarConstraints) serves as the envelope and one of the derived classes (selected in DakotaVarConstraints::get_var_constraints()) serves as the letter.
6.39.2 Constructor & Destructor Documentation

6.39.2.1 DakotaVarConstraints::DakotaVarConstraints()

default constructor.
The default constructor: varConstraintsRep is NULL in this case (a populated problem_db is needed to
build a meaningful DakotaVarConstraints object). This makes it necessary to check for NULL in the copy
constructor, assignment operator, and destructor.

6.39.2.2 DakotaVarConstraints::DakotaVarConstraints(const ProblemDescDB & problem_db,
const DakotaString & vars_type)

standard constructor.
The envelope constructor only needs to extract enough data to properly execute get_var_constraints, since
the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived
classes.

6.39.2.3 DakotaVarConstraints::DakotaVarConstraints(const DakotaVarConstraints & vc)

copy constructor.
Copy constructor manages sharing of varConstraintsRep and incrementing of referenceCount.

6.39.2.4 DakotaVarConstraints::~DakotaVarConstraints() [virtual]

destructor.
Destructor decrements referenceCount and only deletes varConstraintsRep when referenceCount reaches
zero.

6.39.2.5 DakotaVarConstraints::DakotaVarConstraints(BaseConstructor, const ProblemDescDB
& problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite
recursion in the derived class constructors - Coplien, p. 139).
This constructor is the one which must build the base class data for all derived classes. get_var_constraints()
instantiates a derived class letter and the derived constructor selects this base class constructor in its ini-
tialization list (to avoid recursion in the base class constructor calling get_var_constraints() again). Since
the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in
~DakotaVarConstraints).

6.39.3 Member Function Documentation
6.39 DakotaVarConstraints Class Reference

6.39.3.1 DakotaVarConstraints DakotaVarConstraints::operator= (const DakotaVarConstraints & vc)

assignment operator.

Assignment operator decrements referenceCount for old varConstraintsRep, assigns new varConstraints-

6.39.3.2 void DakotaVarConstraints::manage_linear_constraints (const ProblemDescDB & problem_db, const size_t & num_vars) [protected]

perform checks on user input, convert linear constraint coefficient input to matrices, and assign defaults.

Convenience function called from derived class constructors. The number of variables active for applying
linear constraints is passed up from the particular derived class.

6.39.3.3 DakotaVarConstraints * DakotaVarConstraints::get_var_constraints (const ProblemDescDB & problem_db) [private]

Used only by the constructor to initialize varConstraintsRep to the appropriate derived type.

Initializes varConstraintsRep to the appropriate derived type, as given by the variablesType attribute.

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

- DakotaVarConstraints.H
- DakotaVarConstraints.C
6.40 DakotaVariables Class Reference

Base class for the variables class hierarchy.

Inheritance diagram for DakotaVariables::

```
DakotaVariables
  AllMergedVariables
  AllVariables
  FundamentalVariables
  MergedVariables
```

Public Methods

- **DakotaVariables ()**
  
  *default constructor.*

- **DakotaVariables (const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **DakotaVariables (const DakotaString &vars_type)**
  
  *alternate constructor.*

- **DakotaVariables (const DakotaVariables &vars)**
  
  *copy constructor.*

- virtual **~DakotaVariables ()**

  *destructor.*

- **DakotaVariables operator= (const DakotaVariables &vars)**

  *assignment operator.*

- virtual **size_t tv () const**

  *Returns total number of vars.*

- virtual **size_t cv () const**

  *Returns number of active continuous vars.*

- virtual **size_t dv () const**

  *Returns number of active discrete vars.*

- virtual **const DakotaRealVector & continuous_variables () const**

  *return the active continuous variables.*

- virtual **void continuous_variables (const DakotaRealVector &c_vars)**

  *set the active continuous variables.*
virtual const DakotaIntVector & discrete_variables() const  
    return the active discrete variables.

virtual void discrete_variables(const DakotaIntVector &d_vars)  
    set the active discrete variables.

virtual const DakotaStringArray & continuous_variable_labels() const  
    return the active continuous variable labels.

virtual void continuous_variable_labels(const DakotaStringArray &cv_labels)  
    set the active continuous variable labels.

virtual const DakotaStringArray & discrete_variable_labels() const  
    return the active discrete variable labels.

virtual void discrete_variable_labels(const DakotaStringArray &dv_labels)  
    set the active discrete variable labels.

virtual const DakotaRealVector & inactive_continuous_variables() const  
    return the inactive continuous variables.

virtual void inactive_continuous_variables(const DakotaRealVector &i_c_vars)  
    set the inactive continuous variables.

virtual const DakotaIntVector & inactive_discrete_variables() const  
    return the inactive discrete variables.

virtual void inactive_discrete_variables(const DakotaIntVector &i_d_vars)  
    set the inactive discrete variables.

virtual size_t acv() const  
    returns total number of continuous vars.

virtual size_t adv() const  
    returns total number of discrete vars.

virtual DakotaRealVector all_continuous_variables() const  
    returns a single array with all continuous variables.

virtual DakotaIntVector all_discrete_variables() const  
    returns a single array with all discrete variables.

virtual DakotaStringArray all_continuous_variable_labels() const  
    returns a single array with all continuous variable labels.

virtual DakotaStringArray all_discrete_variable_labels() const  
    returns a single array with all discrete variable labels.

virtual void read(istream &s)  
    read a variables object from an istream.
- virtual void write (ostream &s) const
  write a variables object to an ostream.

- virtual void read_annotated (istream &s)
  read a variables object in annotated format from an istream.

- virtual void write_annotated (ostream &s) const
  write a variables object in annotated format to an ostream.

- virtual void read (DakotaBiStream &s)
  read a variables object from the binary restart stream.

- virtual void write (DakotaBoStream &s) const
  write a variables object to the binary restart stream.

- virtual void read (UnPackBuffer &s)
  read a variables object from a packed MPI buffer.

- virtual void write (PackBuffer &s) const
  write a variables object to a packed MPI buffer.

- void write_tabular (ostream &s) const
  write a variables object in tabular format to an ostream.

- DakotaVariables copy () const
  for use when a true copy is needed (the representation is not shared).

- const DakotaIntList & merged_integer_list () const
  returns the list of discrete variables merged into a continuous array.

- const DakotaString & variables_type () const
  returns the variables type: All, Merged, AllMerged, or Fundamental.

Protected Methods

- DakotaVariables (BaseConstructor, const ProblemDescDB &problem_db)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

Protected Attributes

- DakotaIntList mergedIntegerList
  the list of discrete variables for which integrality is relaxed by merging them into a continuous array.

- DakotaString variablesType
  All, Merged, AllMerged, or Fundamental.
• bool a_preproFlag
  used to trigger special behavior in write(ostream&).

• DakotaRealVector emptyRealVector
  an empty real vector returned in get functions when there are no variables corresponding to the request.

• DakotaIntVector emptyIntVector
  an empty int vector returned in get functions when there are no variables corresponding to the request.

• DakotaStringArray emptyStringArray
  an empty label array returned in get functions when there are no variables corresponding to the request.

Private Methods

• virtual void copy_rep (const DakotaVariables *vars_rep)
  Used by copy() to copy the contents of a letter class.

• DakotaVariables * get_variables (const ProblemDescDB &problem_db)
  Used by the standard envelope constructor to instantiate the correct letter class.

• DakotaVariables * get_variables (const DakotaString &vars_type) const
  Used by the alternate envelope constructor, by read functions, and by copy() to instantiate a new letter class.

Private Attributes

• DakotaVariables * variablesRep
  pointer to the letter (initialized only for the envelope).

• int referenceCount
  number of objects sharing variablesRep.

Friends

• bool operator== (const DakotaVariables &vars1, const DakotaVariables &vars2)
  equality operator.

• bool operator!= (const DakotaVariables &vars1, const DakotaVariables &vars2)
  inequality operator.
6.40.1 Detailed Description

Base class for the variables class hierarchy.

The DakotaVariables class is the base class for the class hierarchy providing design, uncertain, and state variables for continuous and discrete domains within a DakotaModel. Using the fundamental arrays from the input specification, different derived classes define different views of the data. For memory efficiency and enhanced polymorphism, the variables hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaVariables) serves as the envelope and one of the derived classes (selected in DakotaVariables::get_variables()) serves as the letter.

6.40.2 Constructor & Destructor Documentation

6.40.2.1 DakotaVariables::DakotaVariables ()

default constructor.

The default constructor: variablesRep is NULL in this case (a populated problem_db is needed to build a meaningful DakotaVariables object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

6.40.2.2 DakotaVariables::DakotaVariables (const ProblemDescDB & problem_db)

standard constructor.

This is the primary envelope constructor which uses problem_db to build a fully populated variables object. It only needs to extract enough data to properly execute get_variables(problem_db), since the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived classes.

6.40.2.3 DakotaVariables::DakotaVariables (const DakotaString & vars_type)

alternate constructor.

This is the alternate envelope constructor for instantiations on the fly. Since it does not have access to problem_db, the letter class is not fully populated. This constructor executes get_variables(vars_type), which invokes the default constructor of the derived letter class, which in turn invokes the default constructor of the base class.

6.40.2.4 DakotaVariables::DakotaVariables (const DakotaVariables & vars)

copy constructor.

Copy constructor manages sharing of variablesRep and incrementing of referenceCount.

6.40.2.5 DakotaVariables::~DakotaVariables () [virtual]

destructor.

Destructor decrements referenceCount and only deletes variablesRep when referenceCount reaches zero.
6.40 DakotaVariables Class Reference

6.40.2.6 DakotaVariables::DakotaVariables (BaseConstructor, const ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

This constructor is the one which must build the base class data for all derived classes. get_variables() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_variables() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaVariables).

6.40.3 Member Function Documentation

6.40.3.1 DakotaVariables::operator= (const DakotaVariables & vars)

assignment operator.


6.40.3.2 DakotaVariables::copy () const

for use when a true copy is needed (the representation is not shared).

Deep copies are used for history mechanisms such as bestVariables and data_pairs since these must catalogue copies (and should not change as the representation within currentVariables changes).

6.40.3.3 DakotaVariables::get_variables (const ProblemDescDB & problem_db) [private]

Used by the standard envelope constructor to instantiate the correct letter class.

Initializes variablesRep to the appropriate derived type, as given by problem_db attributes. The standard derived class constructors are invoked.

6.40.3.4 DakotaVariables::get_variables (const DakotaString & vars_type) const [private]

Used by the alternate envelope constructor, by read functions, and by copy() to instantiate a new letter class.

Initializes variablesRep to the appropriate derived type, as given by the vars_type attribute. The default derived class constructors are invoked.

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

- DakotaVariables.H
- DakotaVariables.C
6.41 DakotaVector Class Template Reference

Template class for the Dakota numerical vector.

Inheritance diagram for DakotaVector::

```
DakotaBaseVector< T >

DakotaVector
```

Public Methods

- DakotaVector ()
  
  Default constructor.

- DakotaVector (size_t len)
  
  Constructor which takes an initial length.

- DakotaVector (size_t len, const T &initial_val)
  
  Constructor which takes an initial length and an initial value.

- DakotaVector (const DakotaVector< T > &a)
  
  Copy constructor.

- DakotaVector (const T *p, size_t len)
  
  Constructor, creates array of length len, with initial value <T> p.

- ~DakotaVector ()
  
  Destructor.

- DakotaVector< T > & operator= (const DakotaVector< T > &a)
  
  Normal const assignment operator.

- DakotaVector< T > & operator= (const T &ival)
  
  Sets all elements in self to the value ival.

- operator T * () const
  
  Converts the DakotaVector to a standard C-style array. Use with care!

- void read (istream &s)
  
  Reads a DakotaVector from an input stream.

- void read (istream &s, DakotaArray< DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array from an input stream.
- void `read_partial` (istream &s, size_t start_index, size_t num_items)
  
  Reads part of a DakotaVector from an input stream.

- void `read_partial` (istream &s, size_t start_index, size_t num_items, DakotaArray< DakotaString > &label_array)
  
  Reads part of a DakotaVector and the corresponding labels from an input stream.

- void `read_tabular` (istream &s)
  
  Reads a DakotaVector from a tabular text input file.

- void `read_annotated` (istream &s, DakotaArray< DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array in annotated from an input stream.

- void `print` (ostream &s) const
  
  Prints a DakotaVector to an output stream.

- void `print` (ostream &s, const DakotaArray< DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array to an output stream.

- void `print_partial` (ostream &s, size_t start_index, size_t num_items) const
  
  Prints part of a DakotaVector to an output stream.

- void `print_partial` (ostream &s, size_t start_index, size_t num_items, const DakotaArray< DakotaString > &label_array) const
  
  Prints part of a DakotaVector and the corresponding labels to an output stream.

- void `print_aprepro` (ostream &s, const DakotaArray< DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array to an output stream in aprepro format.

- void `print_partial_aprepro` (ostream &s, size_t start_index, size_t num_items, const DakotaArray< DakotaString > &label_array) const
  
  Prints part of a DakotaVector and the corresponding labels to an output stream in aprepro format.

- void `print_annotated` (ostream &s, const DakotaArray< DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array in annotated form to an output stream.

- void `read` (DakotaBiStream &s, DakotaArray< DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array from a binary input stream.

- void `print` (DakotaBoStream &s, const DakotaArray< DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array from a binary output stream.

- void `read` (UnPackBuffer &s)
  
  Reads a DakotaVector from a buffer after an MPI receive.

- void `read` (UnPackBuffer &s, DakotaArray< DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array from a buffer after an MPI receive.

- void `print` (PackBuffer &s) const
6.41.1 Detailed Description

template<class T> class DakotaVector<T>

Template class for the Dakota numerical vector.

The DakotaVector class is the numeric vector class. It inherits from the common vector class DakotaBaseVector which provides the same interface for both the STL and RW vector classes. If the STL version of DakotaBaseVector is based on the valarray class then some basic vector operations such as +, * are available. This class adds functionality to read/print vectors in a variety of ways.

6.41.2 Constructor & Destructor Documentation

6.41.2.1 template<class T> DakotaVector<T>::DakotaVector (const T * p, size_t len) [inline]

Constructor, creates array of length len, with initial value <T> p.
Assigns size values from p into array.

6.41.3 Member Function Documentation

6.41.3.1 template<class T> DakotaVector<T> & DakotaVector<T>::operator= (const T & ival) [inline]

Sets all elements in self to the value ival.
Assigns all values of array to ival. If STL, uses the vector assign method because there is no operator=(ival).
Reimplemented from DakotaBaseVector.

6.41.3.2 template<class T> void DakotaVector<T>::testClass ()

Class unit test method.

Unit test method for the DakotaVector class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an unexpected answer is received.

The documentation for this class was generated from the following file:
- DakotaVector.H
6.42 DataInterface Class Reference

Container class for interface specification data.

Public Methods

- **DataInterface ()**
  constructor.

- **DataInterface (const DataInterface &)**
  copy constructor.

- **~DataInterface ()**
  destructor.

- **DataInterface & operator= (const DataInterface &)**
  assignment operator.

- **bool operator== (const DataInterface &)**
  equality operator.

- **void write (ostream &s) const**
  write a DataInterface object to an ostream.

- **void read (UnPackBuffer &s)**
  read a DataInterface object from a packed MPI buffer.

- **void write (PackBuffer &s) const**
  write a DataInterface object to a packed MPI buffer.

Public Attributes

- **DakotaString interfaceType**
  the interface selection: application, system, fork, direct, grid or approximation, ann, rsm, mars, hermite, ksm, mpa, taylor, hierarchical.

- **DakotaString idInterface**
  string identifier for an interface specification data set (from the id_interface specification in InterfSetId).

- **DakotaString inputFilter**
  the input filter for a simulation-based interface (from the input_filter specification in InterfApplic).

- **DakotaString outputFilter**
  the output filter for a simulation-based interface (from the output_filter specification in InterfApplic).
- DakotaStringList analysisDrivers
  the set of analysis drivers for a simulation-based interface (from the analysis_drivers specification in InterfApplic).

- DakotaString parametersFile
  the parameters file for system call and fork interfaces (from the parameters_file specification in InterfApplic).

- DakotaString resultsFile
  the results file for system call and fork interfaces (from the results_file specification in InterfApplic).

- DakotaString analysisUsage
  the analysis command usage string for a system call interface (from the analysis_usage specification in InterfApplic).

- bool apreproFormatFlag
  the flag for aprepro format usage in the parameters file for system call and fork interfaces (from the aprepro specification in InterfApplic).

- bool fileTagFlag
  the flag for file tagging of parameters and results files for system call and fork interfaces (from the file_tag specification in InterfApplic).

- bool fileSaveFlag
  the flag for saving of parameters and results files for system call and fork interfaces (from the file_save specification in InterfApplic).

- int procsPerAnalysis
  processors per parallel analysis for a direct interface (from the processors_per_analysis specification in InterfApplic).

- DakotaString modelCenterFile
  configuration file for defining the simulation model accessed via the direct interface to the ModelCenter framework from Phoenix Integration (from the modelcenter_file specification in InterfApplic).

- DakotaStringList gridHostNames
  names of host machines for a grid interface (from the hostnames specification in InterfApplic).

- DakotaIntArray gridProcsPerHost
  processors per host machine for a grid interface (from the processors_per_host specification in InterfApplic).

- DakotaString interfaceSynchronization
  parallel mode for a simulation-based interface: synchronous or asynchronous (from the asynchronous specification in InterfApplic).

- int asynchLocalEvalConcurrency
  evaluation concurrency for asynchronous simulation-based interfaces (from the evaluation_concurrency specification in InterfApplic).

- int asynchLocalAnalysisConcurrency
  analysis concurrency for asynchronous simulation-based interfaces (from the analysis_concurrency specification in InterfApplic).
analysis concurrency for asynchronous simulation-based interfaces (from the analysis_concurrency
specification in InterfApplic).

- **int evalServers**
  number of evaluation servers to be used in the parallel configuration (from the evaluation_servers
specification in InterfApplic).

- **DakotaString evalScheduling**
  the scheduling approach to be used for concurrent evaluations within an iterator (from the evaluation_self_scheduling and evaluation_static_scheduling specifications in InterfApplic).

- **int analysisServers**
  number of analysis servers to be used in the parallel configuration (from the analysis_servers specification in InterfApplic).

- **DakotaString analysisScheduling**
  the scheduling approach to be used for concurrent analyses within a function evaluation (from the analysis_self_scheduling and analysis_static_scheduling specifications in InterfApplic).

- **DakotaString failAction**
  the selected action upon capture of a simulation failure: abort, retry, recover, or continuation (from the failure_capture specification in InterfApplic).

- **int retryLimit**
  the limit on retries for captured simulation failures (from the retry specification in InterfApplic).

- **DakotaRealVector recoveryFnVals**
  the function values to be returned in a recovery operation for captured simulation failures (from the recover specification in InterfApplic).

- **bool activeSetVectorFlag**
  active set vector: 1=active (ASV control on), 0=inactive (ASV control off) (from the deactivate active_set_vector specification in InterfApplic).

- **bool evalCacheFlag**
  function evaluation cache: 1=active (all new evaluations checked against existing cache and then added to cache), 0=inactive (cache neither queried nor augmented) (from the deactivate evaluation_cache specification in InterfApplic).

- **bool restartFileFlag**
  function evaluation cache: 1=active (all new evaluations written to restart), 0=inactive (no records written to restart) (from the deactivate restart_file specification in InterfApplic).

- **DakotaString approxType**
  the selected approximation type: global, multipoint, local, or hierarchical.

- **DakotaString actualInterfacePtr**
  pointer to the interface specification for constructing the truth model used in building local and multipoint approximations (from the actual_interface_pointer specification in InterfApprox).

- **DakotaString actualInterfaceResponsesPtr**
pointer to the responses specification for constructing the truth model used in building local approximations (from the actual_interface_responses_pointer specification in InterfApprox). This allows differences in gradient specifications between the responses used to build the approximation and the responses computed from the approximation.

- **DakotaString lowFidelityInterfacePtr**
  pointer to the low fidelity interface specification used in hierarchical approximations (from the low_fidelity_interface_pointer specification in InterfApprox).

- **DakotaString highFidelityInterfacePtr**
  pointer to the high fidelity interface specification used in hierarchical approximations (from the high_fidelity_interface_pointer specification in InterfApprox).

- **DakotaString approxDaceMethodPtr**
  pointer to the design of experiments method used in building global approximations (from the dace_method_pointer specification in InterfApprox).

- **DakotaString approxSampleReuse**
  sample reuse selection for building global approximations: none, all, region, or file (from the reuse_samples specification in InterfApprox).

- **DakotaString approxSampleReuseFile**
  the file name for the "file" setting for the reuse_samples specification in InterfApprox.

- **DakotaString approxCorrectionType**
  correction type for global and hierarchical approximations: additive or multiplicative (from the correction specification in InterfApprox).

- **DakotaString approxCorrectionOrder**
  correction order for global and hierarchical approximations: zeroth or first (from the correction specification in InterfApprox).

- **bool approxGradUsageFlag**
  flags the use of gradients in building global approximations (from the use_gradients specification in InterfApprox).

- **DakotaRealVector krigingCorrelations**
  vector of correlations used in building a kriging approximation (from the correlations specification in InterfApprox).

- **int polynomialOrder**
  scalar integer indicating the order of the polynomial approximation (1=linear, 2=quadratic, 3=cubic).

**Private Methods**

- **void assign (const DataInterface &data_interface)**
  convenience function for setting this objects attributes equal to the attributes of the incoming data_interface object (used by copy constructor and assignment operator).
6.42.1 Detailed Description

Container class for interface specification data.

The DataInterface class is used to contain the data from a interface keyword specification. It is populated by `ProblemDescDB::interface_kwhandler()` and is queried by the `ProblemDescDB::get_<datatype>()` functions. A list of DataInterface objects is maintained in `ProblemDescDB::interfaceList`, one for each interface specification in an input file. Default values are managed in the DataInterface constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within `ProblemDescDB` since `ProblemDescDB::interfaceList` is private (a similar model is used with `SurrogateDataPoint` objects contained in `DakotaApproximation` and with `ParallelismLevel` objects contained in `ParallelLibrary`).

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

- DataInterface.H
- DataInterface.C
6.43 DataMethod Class Reference

Container class for method specification data.

Public Methods

- **DataMethod ()**
  constructor.

- **DataMethod (const DataMethod &)**
  copy constructor.

- **~DataMethod ()**
  destructor.

- **DataMethod & operator= (const DataMethod &)**
  assignment operator.

- **bool operator== (const DataMethod &)**
  equality operator.

- **void write (ostream &s) const**
  write a DataMethod object to an ostream.

- **void read (UnPackBuffer &s)**
  read a DataMethod object from a packed MPI buffer.

- **void write (PackBuffer &s) const**
  write a DataMethod object to a packed MPI buffer.

Public Attributes

- **DakotaString methodName**
  the method selection: one of the dot, npsol, opt++, apps, sgopt, nond, dace, or parameter study methods.

- **DakotaString idMethod**
  string identifier for the method specification data set (from the id method specification in MethodIndControl).

- **DakotaString variablesPointer**
  string pointer to the variables specification to be used by this method (from the variables pointer specification in MethodIndControl).

- **DakotaString interfacePointer**
string pointer to the interface specification to be used by this method (from the `interface_pointer` specification in `MethodIndControl`).

- **DakotaString responsesPointer**
  string pointer to the responses specification to be used by this method (from the `responses_pointer` specification in `MethodIndControl`).

- **DakotaString modelType**
  model type selection: single, nested, or layered (from the `model_type` specification in `MethodIndControl`).

- **DakotaString subMethodPointer**
  string pointer to the sub-iterator used by nested models (from the `sub_method_pointer` specification in `MethodIndControl`).

- **DakotaString optionalInterfaceResponsesPointer**
  string pointer to the responses specification used by the optional interface in nested models (from the `interface_responses_pointer` specification in `MethodIndControl`).

- **DakotaRealVector primaryCoeffs**
  the primary mapping matrix used in nested models for weighting contributions from the sub-iterator responses in the top level (objective) functions (from the `primary_mapping_matrix` specification in `MethodIndControl`).

- **DakotaRealVector secondaryCoeffs**
  the secondary mapping matrix used in nested models for weighting contributions from the sub-iterator responses in the top level (constraint) functions (from the `secondary_mapping_matrix` specification in `MethodIndControl`).

- **DakotaString methodOutput**
  method verbosity control: quiet, verbose, debug, or normal (default) (from the `output` specification in `MethodIndControl`).

- **Real convergenceTolerance**
  iteration convergence tolerance for the method (from the `convergence_tolerance` specification in `MethodIndControl`).

- **Real constraintTolerance**
  tolerance for controlling the amount of infeasibility that is allowed before an active constraint is considered to be violated (from the `constraint_tolerance` specification in `MethodIndControl`).

- **int maxIterations**
  maximum number of iterations allowed for the method (from the `max_iterations` specification in `MethodIndControl`).

- **int maxFunctionEvaluations**
  maximum number of function evaluations allowed for the method (from the `max_function_evaluations` specification in `MethodIndControl`).

- **bool speculativeFlag**
  flag for use of speculative gradient approaches for maintaining parallel load balance during the line search portion of optimization algorithms (from the `speculative` specification in `MethodIndControl`).
- DakotaRealVector linearIneqConstraintCoeffs
  coefficient matrix for the linear inequality constraints (from the linear_inequality_constraint_-matrix specification in MethodIndControl).

- DakotaRealVector linearIneqLowerBnds
  lower bounds for the linear inequality constraints (from the linear_inequality_lower_bounds specification in MethodIndControl).

- DakotaRealVector linearIneqUpperBnds
  upper bounds for the linear inequality constraints (from the linear_inequality_upper_bounds specification in MethodIndControl).

- DakotaRealVector linearEqConstraintCoeffs
  coefficient matrix for the linear equality constraints (from the linear_equation_constraint_-matrix specification in MethodIndControl).

- DakotaRealVector linearEqTargets
  targets for the linear equality constraints (from the linear_equation_targets specification in MethodIndControl).

- DakotaString minMaxType
  the optimization_type specification in MethodDOTDC.

- int verifyLevel
  the verify_level specification in MethodNPSOLDC.

- Real functionPrecision
  the function_precision specification in MethodNPSOLDC.

- Real lineSearchTolerance
  the linesearch_tolerance specification in MethodNPSOLDC.

- DakotaString searchMethod
  the search_method specification for Newton and nonlinear interior-point methods in MethodOPTPPDC.

- Real gradientTolerance
  the gradient_tolerance specification in MethodOPTPPDC.

- Real maxStep
  the max_step specification in MethodOPTPPDC.

- DakotaString meritFn
  the merit_function specification for nonlinear interior-point methods in MethodOPTPPDC.

- DakotaString centralPath
  the central_path specification for nonlinear interior-point methods in MethodOPTPPDC.

- Real stepLenToBoundary
the steplength_to_boundary specification for nonlinear interior-point methods in MethodOPTPPDC.

- Real centeringParam
the centering parameter specification for nonlinear interior-point methods in MethodOPTPPDC.

- int searchSchemeSize
the search_scheme_size specification for PDS methods in MethodOPTPPDC.

- Real solnAccuracy
the solution accuracy specification in MethodSGOPTDC.

- Real maxCPUPtime
the max_cpu_time specification in MethodSGOPTDC.

- Real crossoverRate
the crossover_rate specification for GA/EPSA methods in MethodSGOPTEA.

- Real mutationDimRate
the dimension_rate specification for mutation in GA/EPSA methods in MethodSGOPTEA.

- Real mutationPopRate
the population_rate specification for mutation in GA/EPSA methods in MethodSGOPTEA.

- Real mutationScale
the mutation_scale specification for GA/EPSA methods in MethodSGOPTEA.

- Real mutationMinScale
the min_scale specification for mutation in EPSA methods in MethodSGOPTEA.

- Real initDelta
the initial_delta specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.

- Real threshDelta
the threshold_delta specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.

- Real contractFactor
the contraction_factor specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.

- int populationSize
the population_size specification for GA/EPSA methods in MethodSGOPTEA.

- int newSolnsGenerated
the new_solutions_generated specification for GA/EPSA methods in MethodSGOPTEA.

- int numberRetained
the integer assignment to random, chc, or elitist in the replacement_type specification for GA/EPSA methods in MethodSGOPTEA.
- **int expandAfterSuccess**
  
  The `expand_after_success` specification for PS/SW methods in MethodSGOPTPS and MethodSGOPTSW.

- **int contractAfterFail**
  
  The `contract_after_failure` specification for the SW method in MethodSGOPTSW.

- **int mutationRange**
  
  The `mutation_range` specification for the pga_int method in MethodSGOPTEA.

- **int numPartitions**
  
  The `num_partitions` specification for EPSA methods in MethodSGOPTEA.

- **int totalPatternSize**
  
  The `total_pattern_size` specification for APPS/PS methods in MethodAPPSDC and MethodSGOPTPS.

- **int batchSize**
  
  The `batch_size` specification for the sMC method in MethodSGOPTSMC.

- **bool nonAdaptiveFlag**
  
  The `non_adaptive` specification for the pga_real method in MethodSGOPTEA.

- **bool randomizeOrderFlag**
  
  The stochastic specification for the PS method in MethodSGOPTPS.

- **bool expansionFlag**
  
  The no_expansion specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.

- **DakotaString selectionPressure**
  
  The `selection_pressure` specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString replacementType**
  
  The `replacement_type` specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString crossoverType**
  
  The `crossover_type` specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString mutationType**
  
  The `mutation_type` specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString exploratoryMoves**
  
  The `exploratory_moves` specification for the PS method in MethodSGOPTPS.

- **DakotaString patternBasis**
  
  The `pattern_basis` specification for APPS/PS methods in MethodAPPSDC and MethodSGOPTPS.

- **DakotaIntArray varPartitions**
the partitions specification for sMC/PStudy methods in MethodSGOPTSMC and MethodPSMPS.

- **DakotaString daceMethod**
  
  the dace method selection: grid, random, oas, lhs, qlhs, oaslhs, box, behken, or central_composite (from the dace specification in MethodDACE).

- **int numSymbols**
  
  the symbols specification for DACE methods.

- **int randomSeed**
  
  the seed specification for SGOPT, NonD, & DACE methods.

- **int numSamples**
  
  the samples specification for NonD & DACE methods.

- **bool fixedSeedFlag**
  
  flag for fixing the value of the seed among different NonD/DACE sample sets. This results in the use of the same sampling stencil/pattern throughout a strategy with repeated sampling.

- **int expansionTerms**
  
  the expansion_terms specification in MethodNonDPCE.

- **int expansionOrder**
  
  the expansion_order specification in MethodNonDPCE.

- **DakotaString sampleType**
  
  the sample_type specification in MethodNonDMC and MethodNonDPCE.

- **DakotaString reliabilityMethod**
  
  the amv/c iterated_amv/form/c sorm selection in MethodNonDAMV.

- **DakotaRealArray responseThresholds**
  
  the response_thresholds specification in MethodNonDMC and MethodNonDPCE.

- **DakotaRealArray responseLevels**
  
  the response_levels specification in MethodNonDAMV.

- **DakotaRealArray probabilityLevels**
  
  the probability_levels specification in MethodNonDAMV.

- **bool allVarsFlag**
  
  the all.variables specification in MethodNonDMC.

- **int paramStudyType**
  
  the type of parameter study: list(-1), vector(1, 2, or 3), centered(4), or multidim(5).

- **DakotaRealVector finalPoint**
  
  the final_point specification in MethodPSVPS.

- **DakotaRealVector stepVector**
the step vector specification in MethodPSVPS.

- Real stepLength
  the step length specification in MethodPSVPS.

- int numSteps
  the num steps specification in MethodPSVPS.

- DakotaRealVector listOfPoints
  the list of points specification in MethodPSLPS.

- Real percentDelta
  the percent delta specification in MethodPSCPS.

- int deltasPerVariable
  the deltas per variable specification in MethodPSCPS.

Private Methods

- void assign (const DataMethod &data_method)
  convenience function for setting this objects attributes equal to the attributes of the incoming data_method
  object (used by copy constructor and assignment operator).

6.43.1 Detailed Description

Container class for method specification data.

The DataMethod class is used to contain the data from a method keyword specification. It is populated by ProblemDescDB::method_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataMethod objects is maintained in ProblemDescDB::methodList, one for each method specification in an input file. Default values are managed in the DataMethod constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::methodList is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

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

- DataMethod.H
- DataMethod.C
6.44 DataResponses Class Reference

Container class for responses specification data.

Public Methods

- **DataResponses ()**
  constructor.

- **DataResponses (const DataResponses &)**
  copy constructor.

- **~DataResponses ()**
  destructor.

- DataResponses & operator= (const DataResponses &)
  assignment operator.

- bool operator== (const DataResponses &)
  equality operator.

- void write (ostream &s) const
  write a DataResponses object to an ostream.

- void read (UnPackBuffer &s)
  read a DataResponses object from a packed MPI buffer.

- void write (PackBuffer &s) const
  write a DataResponses object to a packed MPI buffer.

Public Attributes

- size_t numObjectiveFunctions
  number of objective functions (from the num_objective_functions specification in RespFnOpt).

- size_t numNonlinearIneqConstraints
  number of nonlinear inequality constraints (from the num_nonlinear_inequality_constraints specification in RespFnOpt).

- size_t numNonlinearEqConstraints
  number of nonlinear equality constraints (from the num_nonlinear_equality_constraints specification in RespFnOpt).

- size_t numLeastSqTerms
  number of least squares terms (from the num_least_squares_terms specification in RespFnLS).
• `size_t numResponseFunctions`
  number of generic response functions (from the `num_response_functions` specification in `RespFnGen`).

• `DakotaRealVector multiObjectiveWeights`
  vector of multiobjective weightings (from the `multi_objective_weights` specification in `RespFnOpt`).

• `DakotaRealVector nonlinearIneqLowerBnds`
  vector of nonlinear inequality constraint lower bounds (from the `nonlinear_inequality_lower_bounds` specification in `RespFnOpt`).

• `DakotaRealVector nonlinearIneqUpperBnds`
  vector of nonlinear inequality constraint upper bounds (from the `nonlinear_inequality_upper_bounds` specification in `RespFnOpt`).

• `DakotaRealVector nonlinearEqTargets`
  vector of nonlinear equality constraint targets (from the `nonlinear_equality_targets` specification in `RespFnOpt`).

• `DakotaString gradientType`
  gradient type: none, numerical, analytic, or mixed (from the `no_gradients`, `numerical_gradients`, `analytic_gradients`, and `mixed_gradients` specifications in `RespGrad`).

• `DakotaString hessianType`
  Hessian type: none or analytic (from the `no_hessians` and `analytic_hessians` specifications in `RespHess`).

• `DakotaString methodSource`
  numerical gradient method source: dakota or vendor (from the `method_source` specification in `RespGradNum` and `RespGradMixed`).

• `DakotaString intervalType`
  numerical gradient interval type: forward or central (from the `interval_type` specification in `RespGradNum` and `RespGradMixed`).

• `Real fdStepSize`
  numerical gradient finite difference step size (from the `fd_step_size` specification in `RespGradNum` and `RespGradMixed`).

• `DakotaIntList idNumerical`
  mixed gradient numerical identifiers (from the `id_numerical` specification in `RespGradMixed`).

• `DakotaIntList idAnalytic`
  mixed gradient analytic identifiers (from the `id_analytic` specification in `RespGradMixed`).

• `DakotaString idResponses`
  string identifier for the responses specification data set (from the `id_responses` specification in `RespSetId`).
- DakotaStringArray responseLabels

  the response labels array (from the response_descriptors specification in RespLabels).

### Private Methods

- void assign (const DataResponses &data_responses)

  convenience function for setting this objects attributes equal to the attributes of the incoming data_responses object (used by copy constructor and assignment operator).

### 6.44.1 Detailed Description

Container class for responses specification data.

The DataResponses class is used to contain the data from a responses keyword specification. It is populated by ProblemDescDB::responses_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataResponses objects is maintained in ProblemDescDB::responsesList, one for each responses specification in an input file. Default values are managed in the DataResponses constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::responsesList is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

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

- DataResponses.H
- DataResponses.C
6.45  DataStrategy Class Reference

Container class for strategy specification data.

Public Methods

- **DataStrategy ()**
  *constructor.*

- **DataStrategy (const DataStrategy &)**
  *copy constructor.*

- **~DataStrategy ()**
  *destructor.*

- **DataStrategy & operator= (const DataStrategy &)**
  *assignment operator.*

- **void write (ostream &s) const**
  *write a DataStrategy object to an ostream.*

- **void read (UnPackBuffer &s)**
  *read a DataStrategy object from a packed MPI buffer.*

- **void write (PackBuffer &s) const**
  *write a DataStrategy object to a packed MPI buffer.*

Public Attributes

- **DakotaString strategyType**
  *the strategy selection: multi_level, surrogate_based_opt, opt_under_uncertainty, branch_and_bound, multi_start, pareto_set, or single_method.*

- **bool graphicsFlag**
  *flags use of graphics by the strategy (from the graphics specification in StratIndControl).*

- **bool tabularDataFlag**
  *flags tabular data collection by the strategy (from the tabular_graphics_data specification in StratIndControl).*

- **DakotaString tabularDataFile**
  *the filename used for tabular data collection by the strategy (from the tabular_graphics_file specification in StratIndControl).*

- **int iteratorServers**
number of servers for concurrent iterator parallelism (from the iterator_servers specification in StratIndControl).

- **DakotaString iteratorScheduling**
  type of scheduling (self or static) used in concurrent iterator parallelism (from the iterator_self_scheduling and iterator_static_scheduling specifications in StratIndControl).

- **DakotaString methodPointer**
  method identifier for the strategy (from the opt_method_pointer specifications in StratSBO, StratOUU, StratBandB, and StratParetoSet and method_pointer specifications in StratSingle and StratMultiStart).

- **int branchBndNumSamplesRoot**
  number of samples at the root for the branch and bound strategy (from the num_samples_at_root specification in StratBandB).

- **int branchBndNumSamplesNode**
  number of samples at each node for the branch and bound strategy (from the num_samples_at_node specification in StratBandB).

- **DakotaStringList multilevelMethodList**
  list of methods for the multilevel hybrid optimization strategy (from the method_list specification in StratML).

- **DakotaString multilevelType**
  the type of multilevel hybrid optimization strategy: uncoupled, uncoupled adaptive, or coupled (from the uncoupled, adaptive, and coupled specifications in StratML).

- **Real multilevelProgThresh**
  progress threshold for uncoupled adaptive multilevel hybrids (from the progress_threshold specification in StratML).

- **DakotaString multilevelGlobalMethodPointer**
  global method pointer for coupled multilevel hybrids (from the global_method_pointer specification in StratML).

- **DakotaString multilevelLocalMethodPointer**
  local method pointer for coupled multilevel hybrids (from the local_method_pointer specification in StratML).

- **Real multilevelLSProb**
  local search probability for coupled multilevel hybrids (from the local_search_probability specification in StratML).

- **int surrBasedOptMaxIterations**
  maximum number of iterations in the surrogate-based optimization strategy (from the max_iterations specification in StratSBO).

- **Real surrBasedOptConvTol**
  convergence tolerance in the surrogate-based optimization strategy (from the convergence_tolerance specification in StratSBO).

- **int surrBasedOptSoftConvLimit**
number of consecutive iterations with change less than \texttt{surrBasedOptConToI} required to trigger convergence within the surrogate-based optimization strategy (from the soft-convergence-limit specification in \texttt{StratSBO}).

- **Real \texttt{surrBasedOptTRInitSize}**
  
  initial trust region size in the surrogate-based optimization strategy (from the initial-size specification in \texttt{StratSBO}) note: this is a relative value, e.g., 0.1 = 10\% of global bounds distance (upper bound - lower bound) for each variable.

- **Real \texttt{surrBasedOptTRMinSize}**
  
  minimum trust region size in the surrogate-based optimization strategy (from the minimum-size specification in \texttt{StratSBO}), if the trust region size falls below this threshold the SBO iterations are terminated (note: if kriging is used with SBO, the min trust region size is set to 1.0e-3 in attempt to avoid ill-conditioned matrices that arise in kriging over small trust ).

- **Real \texttt{surrBasedOptTRContractTrigger}**
  
  trust region minimum improvement level (ratio of actual to predicted decrease in objective fcn) in the surrogate-based optimization strategy (from the contract-region-threshold specification in \texttt{StratSBO}), the trust region shrinks or is rejected if the ratio is below this value (“eta_1” in the Conn-Gould-Toint trust region book).

- **Real \texttt{surrBasedOptTRExpandTrigger}**
  
  trust region sufficient improvement level (ratio of actual to predicted decrease in objective fcn) in the surrogate-based optimization strategy (from the expand-region-threshold specification in \texttt{StratSBO}), the trust region expands if the ratio is above this value (“eta_2” in the Conn-Gould-Toint trust region book).

- **Real \texttt{surrBasedOptTRContract}**
  
  trust region contraction factor in the surrogate-based optimization strategy (from the contraction-factor specification in \texttt{StratSBO}).

- **Real \texttt{surrBasedOptTRExpand}**
  
  trust region expansion factor in the surrogate-based optimization strategy (from the expansion-factor specification in \texttt{StratSBO}).

- **int \texttt{concurrentRandomJobs}**
  
  number of random jobs to perform in the concurrent strategy (from the random-starts and random-weight-sets specifications in \texttt{StratMultiStart} and \texttt{StratParetoSet}).

- **int \texttt{concurrentSeed}**
  
  seed for the selected random jobs within the concurrent strategy (from the seed specification in \texttt{StratMulti-Start} and \texttt{StratParetoSet}).

- **DakotaRealVector \texttt{concurrentParameterSets}**
  
  user-specified (i.e., nonrandom) parameter sets to evaluate in the concurrent strategy (from the starting-points and multi-objective-weight-sets specifications in \texttt{StratMultiStart} and \texttt{StratParetoSet}).

**Private Methods**

- **void \texttt{assign} (const DataStrategy &data\_strategy)**
  
  convenience function for setting this objects attributes equal to the attributes of the incoming data\_strategy object (used by copy constructor and assignment operator).
6.45.1 Detailed Description

Container class for strategy specification data.

The DataStrategy class is used to contain the data from a strategy keyword specification. It is populated by `ProblemDescDB::strategy_kwhandler()` and is queried by the `ProblemDescDB::get_<datatype>()` functions. Default values are managed in the DataStrategy constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within `ProblemDescDB` since `ProblemDescDB::strategySpec` is private (a similar model is used with `SurrogateDataPoint` objects contained in `DakotaApproximation` and with `ParallelismLevel` objects contained in `ParallelILibrary`).

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

- DataStrategy.H
- DataStrategy.C
6.46 DataVariables Class Reference

Container class for variables specification data.

Public Methods

- `DataVariables()`
  constructor.

- `DataVariables(const DataVariables&)`
  copy constructor.

- `~DataVariables()`
  destructor.

- `DataVariables & operator=(const DataVariables&)`
  assignment operator.

- `bool operator==(const DataVariables&)`
  equality operator.

- `void write(ostream&) const`
  write a DataVariables object to an ostream.

- `void read(UnPackBuffer&)`
  read a DataVariables object from a packed MPI buffer.

- `void write(PackBuffer&) const`
  write a DataVariables object to a packed MPI buffer.

- `size_t design()`
  return total number of design variables.

- `size_t uncertain()`
  return total number of uncertain variables.

- `size_t state()`
  return total number of state variables.

- `size_t num_continuous_variables()`
  return total number of continuous variables.

- `size_t num_discrete_variables()`
  return total number of discrete variables.

- `size_t num_variables()`
  return total number of variables.
Public Attributes

- **DakotaString idVariables**
  
  string identifier for the variables specification data set (from the id_variables specification in VarSet-Id).

- `size_t numContinuousDesVars`
  
  number of continuous design variables (from the continuous_design specification in VarDV).

- `size_t numDiscreteDesVars`
  
  number of discrete design variables (from the discrete_design specification in VarDV).

- `size_t numNormalUncVars`
  
  number of normal uncertain variables (from the normal_uncertain specification in VarUV).

- `size_t numLognormalUncVars`
  
  number of lognormal uncertain variables (from the lognormal_uncertain specification in VarUV).

- `size_t numUniformUncVars`
  
  number of uniform uncertain variables (from the uniform_uncertain specification in VarUV).

- `size_t numLoguniformUncVars`
  
  number of loguniform uncertain variables (from the loguniform_uncertain specification in VarUV).

- `size_t numWeibullUncVars`
  
  number of weibull uncertain variables (from the weibull_uncertain specification in VarUV).

- `size_t numHistogramUncVars`
  
  number of histogram uncertain variables (from the histogram_uncertain specification in VarUV).

- `size_t numContinuousStateVars`
  
  number of continuous state variables (from the continuous_state specification in VarSV).

- `size_t numDiscreteStateVars`
  
  number of discrete state variables (from the discrete_state specification in VarSV).

- **DakotaRealVector continuousDesignVars**
  
  initial values for the continuous design variables array (from the cdv_initial_point specification in VarDV).

- **DakotaRealVector continuousDesignLowerBnds**
  
  the continuous design lower bounds array (from the cdv_lower_bounds specification in VarDV).

- **DakotaRealVector continuousDesignUpperBnds**
  
  the continuous design upper bounds array (from the cdv_upper_bounds specification in VarDV).

- **DakotaIntVector discreteDesignVars**
  
  initial values for the discrete design variables array (from the ddv_initial_point specification in VarDV).

- **DakotaIntVector discreteDesignLowerBnds**
the discrete design lower bounds array (from the \texttt{ddv_lower_bounds} specification in \texttt{VarDV}).

- DakotaIntVector \texttt{discreteDesignUpperBnds}
  the discrete design upper bounds array (from the \texttt{ddv_upper_bounds} specification in \texttt{VarDV}).

- DakotaStringArray \texttt{continuousDesignLabels}
  the continuous design labels array (from the \texttt{cdv_descriptors} specification in \texttt{VarDV}).

- DakotaStringArray \texttt{discreteDesignLabels}
  the discrete design labels array (from the \texttt{ddv_descriptors} specification in \texttt{VarDV}).

- DakotaRealVector \texttt{normalUncMeans}
  means of the normal uncertain variables (from the \texttt{nuv_means} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{normalUncStdDevs}
  standard deviations of the normal uncertain variables (from the \texttt{nuv_std_deviations} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{normalUncDistLowerBnds}
  distribution lower bounds for the normal uncertain variables (from the \texttt{nuv_dist_lower_bounds} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{normalUncDistUpperBnds}
  distribution upper bounds for the normal uncertain variables (from the \texttt{nuv_dist_upper_bounds} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{lognormalUncMeans}
  means of the lognormal uncertain variables (from the \texttt{lnuv_means} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{lognormalUncStdDevs}
  standard deviations of the lognormal uncertain variables (from the \texttt{lnuv_std_deviations} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{lognormalUncErrFacts}
  error factors for the lognormal uncertain variables (from the \texttt{lnuv_error_factors} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{lognormalUncDistLowerBnds}
  distribution lower bounds for the lognormal uncertain variables (from the \texttt{lnuv_dist_lower_bounds} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{lognormalUncDistUpperBnds}
  distribution upper bounds for the lognormal uncertain variables (from the \texttt{lnuv_dist_upper_bounds} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{uniformUncDistLowerBnds}
  distribution lower bounds for the uniform uncertain variables (from the \texttt{uuv_dist_lower_bounds} specification in \texttt{VarUV}).

- DakotaRealVector \texttt{uniformUncDistUpperBnds}
  distribution upper bounds for the uniform uncertain variables (from the \texttt{uuv_dist_upper_bounds} specification in \texttt{VarUV}).
- DakotaRealVector `loguniformUncDistLowerBnds`
  distribution lower bounds for the loguniform uncertain variables (from the `luu_v_dist_lower_bounds` specification in `VarUV`).

- DakotaRealVector `loguniformUncDistUpperBnds`
  distribution upper bounds for the loguniform uncertain variables (from the `luu_v_dist_upper_bounds` specification in `VarUV`).

- DakotaRealVector `weibullUncAlphas`
  alpha factors for the weibull uncertain variables (from the `wuv_alphas` specification in `VarUV`).

- DakotaRealVector `weibullUncBetas`
  beta factors for the weibull uncertain variables (from the `wuv_betas` specification in `VarUV`).

- DakotaRealVectorArray `histogramUncBinPairs`
  an array containing a vector of (x,y) pairs for each bin-based histogram uncertain variable (see continuous linear histogram in LHS manual; from the `huv_num_bin_pairs` and `huv_bin_pairs` specifications in `VarUV`).

- DakotaRealVectorArray `histogramUncPointPairs`
  an array containing a vector of (x,y) pairs for each point-based histogram uncertain variable (see discrete histogram in LHS manual; from the `huv_num_point_pairs` and `huv_point_pairs` specifications in `VarUV`).

- DakotaRealMatrix `uncertainCorrelations`
  correlation matrix for all uncertain variables (from the `uncertain_correlation_matrix` specification in `VarUV`). This matrix specifies rank correlations for sampling methods (i.e., LHS) and correlation coefficients ($\rho_{ij} = \text{normalized covariance matrix}$) for analytic reliability methods.

- DakotaRealVector `uncertainVars`
  array of values for all uncertain variables (built and initialized in `ProblemDescDB::variables&whandler()`).

- DakotaRealVector `uncertainDistLowerBnds`
  distribution lower bounds for all uncertain variables (collected from `nuv_dist_lower_bounds`, `lnuv_dist_lower_bounds`, `uuv_dist_lower_bounds`, `luu_v_dist_lower_bounds`, and `huv_dist_lower_bounds` specifications in `VarUV`).

- DakotaRealVector `uncertainDistUpperBnds`
  distribution upper bounds for all uncertain variables (collected from `nuv_dist_upper_bounds`, `lnuv_dist_upper_bounds`, `uuv_dist_upper_bounds`, `luu_v_dist_upper_bounds`, and `huv_dist_upper_bounds` specifications in `VarUV`).

- DakotaStringArray `uncertainLabels`
  labels for all uncertain variables (collected from `nuv_descriptors`, `lnuv_descriptors`, `uuv_descriptors`, `luu_v_descriptors`, and `huv_descriptors` specifications in `VarUV`).

- DakotaRealVector `continuousStateVars`
  initial values for the continuous state variables array (from the `csv_initial_state` specification in `VarSV`).
• DakotaRealVector **continuousStateLowerBnds**
  the continuous state lower bounds array (from the csv.lower.bounds specification in VarSV).

• DakotaRealVector **continuousStateUpperBnds**
  the continuous state upper bounds array (from the csv.upper.bounds specification in VarSV).

• DakotaIntVector **discreteStateVars**
  initial values for the discrete state variables array (from the dsv.initial.state specification in VarSV).

• DakotaIntVector **discreteStateLowerBnds**
  the discrete state lower bounds array (from the dsv.lower.bounds specification in VarSV).

• DakotaIntVector **discreteStateUpperBnds**
  the discrete state upper bounds array (from the dsv.upper.bounds specification in VarSV).

• DakotaStringArray **continuousStateLabels**
  the continuous state labels array (from the csv.descriptors specification in VarSV).

• DakotaStringArray **discreteStateLabels**
  the discrete state labels array (from the dsv.descriptors specification in VarSV).

### Private Methods

• void **assign** (const DataVariables &data_variables)
  convenience function for setting this objects attributes equal to the attributes of the incoming data_variables object (used by copy constructor and assignment operator).

### 6.46.1 Detailed Description

Container class for variables specification data.

The DataVariables class is used to contain the data from a variables keyword specification. It is populated by ProblemDescDB::variables_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataVariables objects is maintained in ProblemDescDB::variablesList, one for each variables specification in an input file. Default values are managed in the DataVariables constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::variablesList is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

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

• DataVariables.H
• DataVariables.C
6.47 DirectFnApplicInterface Class Reference

Derived application interface class which spawns simulation codes and testers using direct procedure calls.

Inheritance diagram for DirectFnApplicInterface::

![Inheritance Diagram](image)

**Public Methods**

- **DirectFnApplicInterface** (const ProblemDescDB &problem_db, const size_t &num_fns)
  
  *constructor.*

- **~DirectFnApplicInterface ()**
  
  *destructor.*

- void **derived_map** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int fn_eval_id)
  
  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- void **derived_map_asynch** (const ParamResponsePair &pair)
  
  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- void **derived_synch** (DakotaPRPList &prp_list)
  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- void **derived_synch_nowait** (DakotaPRPList &prp_list)
  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- int **derived_synchronous_local_analysis** (const int &analysis_id)
  
  Execute a particular analysis (identified by analysis_id) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve Analyses Synch().
Protected Methods

- int derived_map_if (const DakotaString &if_name)
  
  execute the input filter portion of a direct evaluation invocation.

- int derived_map_ac (const DakotaString &ac_name)
  
  execute an analysis code portion of a direct evaluation invocation.

- int derived_map_of (const DakotaString &of_name)
  
  execute the output filter portion of a direct evaluation invocation.

- void set_local_data ()
  
  convenience function for local test simulators which sets variable attributes and zeros response data.

- void overlay_response (DakotaResponse &response)
  
  convenience function for local test simulators which overlays response contributions from multiple analyses using MPI_Reduce.

Protected Attributes

- DakotaString iFilterName
  
  name of the direct function input filter.

- DakotaString oFilterName
  
  name of the direct function output filter.

- DakotaString pxcFile
  
  name of the ModelCenter simulation config file.

- bool gradFlag
  
  signals use of fnGrads in direct simulator functions.

- bool hessFlag
  
  signals use of fnHessians in direct simulator functions.

- size_t numFns
  
  number of functions in fnVals.

- size_t numVars
  
  total number of continuous and discrete variables.

- size_t numGradVars
  
  number of continuous variables.

- DakotaRealVector xVect
  
  continuous and discrete variable set used within direct simulator functions.

- DakotaRealVector fnVals
  
  response function values set within direct simulator functions.
- DakotaRealMatrix fnGrads
  response function gradients set within direct simulator functions.

- DakotaRealMatrixArray fnHessians
  response function Hessians set within direct simulator functions.

- DakotaVariables directFnVars
  class scope variables object.

- DakotaIntArray directFnASV
  class scope active set vector object.

- DakotaResponse directFnResponse
  class scope response object.

### Private Methods

- int cantilever (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  the cantilever optimization under uncertainty test function.

- int cyl_head (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  the cylinder head constrained optimization test function.

- int rosenbrock (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  the rosenbrock optimization and least squares test function.

- int text_book (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  the text_book constrained optimization test function.

- int text_book1 (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  portion of text_book() evaluating the objective function and its derivatives.

- int text_book2 (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  portion of text_book() evaluating constraint 1 and its derivatives.

- int text_book3 (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  portion of text_book() evaluating constraint 2 and its derivatives.

- int text_book_ouu (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)
  the text_book_ouu optimization under uncertainty test function.
6.47 DirectFnApplicInterface Class Reference

- int salinas (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)

  direct interface to the SALINAS structural dynamics simulation code.

- int mc_api_run (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)

  Call ModelCenter via API, HKIM 4/3/03.

6.47.1 Detailed Description

Derived application interface class which spawns simulation codes and testers using direct procedure calls. DerivedFnApplicInterface uses a few linkable simulation codes and several internal member functions to perform parameter to response mappings.

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

- DirectFnApplicInterface.H
- DirectFnApplicInterface.C
6.48  DOTOptimizer Class Reference

Wrapper class for the DOT optimization library.

Inheritance diagram for DOTOptimizer::

```
DakotaOptimizer
    |__________________________|
    DakotaOptLeastSq
    |__________________________|
    DakotaOptimizer
    |__________________________|
    DOTOptimizer
```

**Public Methods**

- **DOTOptimizer (DakotaModel &model)**
  
  *constructor.*

- **~DOTOptimizer ()**
  
  *destructor.*

- **void find_optimum ()**
  
  *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

**Private Methods**

- **void allocate_workspace ()**
  
  *Allocates workspace for the optimizer.*

**Private Attributes**

- **int dotInfo**
  
  *INFO from DOT manual.*

- **int dotFDSinfo**
  
  *internal DOT parameter NGOTOZ.*

- **int dotMethod**
  
  *METHOD from DOT manual.*
6.48 DOTOptimizer Class Reference

- int `printControl`
  
  `IPRINT` from DOT manual (controls output verbosity).

- int `optimizationType`
  
  `MINMAX` from DOT manual (minimize or maximize).

- DakotaRealArray `realCntlParmArray`
  
  `RPRM` from DOT manual.

- DakotaIntArray `intCntlParmArray`
  
  `IPRM` from DOT manual.

- DakotaRealVector `localConstraintValues`
  
  Array of nonlinear constraint values passed to DOT.

- int `realWorkSpaceSize`
  
  Size of `realWorkSpace`.

- int `intWorkSpaceSize`
  
  Size of `intWorkSpace`.

- DakotaRealArray `realWorkSpace`
  
  Real work space for DOT.

- DakotaIntArray `intWorkSpace`
  
  Int work space for DOT.

- DakotaSizetList `constraintMappingIndices`
  
  A list of indices for referencing the corresponding DakotaResponse constraints used in computing the DOT constraints.

- DakotaRealList `constraintMappingMultipliers`
  
  A list of multipliers for mapping the DakotaResponse constraints to the DOT constraints.

- DakotaRealList `constraintMappingOffsets`
  
  A list of offsets for mapping the DakotaResponse constraints to the DOT constraints.

6.48.1 Detailed Description

Wrapper class for the DOT optimization library.

The DOTOptimizer class provides a wrapper for DOT, a commercial Fortran 77 optimization library from Vanderplaats Research and Development. It uses a reverse communication mode, which avoids the static function and static attribute issues that arise with function pointer designs (see NPSOLOptimizer and SNLLOptimizer).

The user input mappings are as follows: max_iterations is mapped into DOT’s ITMAX parameter within its IPRM array, max_function_evaluations is implemented directly in the find_optimum() loop since there is no DOT parameter equivalent, convergence_tolerance is mapped into DOT’s
DELOBJ parameter (the relative convergence tolerance) within its RPRM array, output verbosity is mapped into DOT's IPRINT parameter within its function call parameter list (verbose: IPRINT = 7; quiet: IPRINT = 3), and optimization type is mapped into DOT's MINMAX parameter within its function call parameter list. Refer to [Vanderplaats Research and Development, 1995] for information on IPRM, RPRM, and the DOT function call parameter list.

6.48.2 Member Data Documentation

6.48.2.1 int DOTOptimizer::dotInfo [private]
INFO from DOT manual.
Information requested by DOT: 0=optimization complete, 1=get values, 2=get gradients

6.48.2.2 int DOTOptimizer::dotFDSinfo [private]
internal DOT parameter NGOTOZ.
the DOT parameter list has been modified to pass NGOTOZ, which signals whether DOT is finite-differencing (nonzero value) or performing the line search (zero value).

6.48.2.3 int DOTOptimizer::dotMethod [private]
METHOD from DOT manual.
For nonlinear constraints: 0/1 = dot_mmfd, 2 = dot_slp, 3 = dot_sqp. For unconstrained: 0/1 = dot_bfgs, 2 = dot_frcg.

6.48.2.4 int DOTOptimizer::printControl [private]
IPRINT from DOT manual (controls output verbosity).
Values range from 0 (least output) to 7 (most output).

6.48.2.5 int DOTOptimizer::optimizationType [private]
MINMAX from DOT manual (minimize or maximize).
Values of 0 or -1 (minimize) or 1 (maximize).

6.48.2.6 DakotaRealArray DOTOptimizer::realCntlParmArray [private]
RPRM from DOT manual.
Array of real control parameters.

6.48.2.7 DakotaIntArray DOTOptimizer::intCntlParmArray [private]
IPRM from DOT manual.
Array of integer control parameters.

6.48.2.8 DakotaRealVector DOTOptimizer::localConstraintValues  [private]

array of nonlinear constraint values passed to DOT.
This array must be of nonzero length (sized with localConstraintArraySize) and must contain only one-
sided inequality constraints which are $\leq 0$ (which requires a transformation from 2-sided inequalities and
equalities).

6.48.2.9 DakotaSizetList DOTOptimizer::constraintMappingIndices  [private]

a list of indices for referencing the corresponding DakotaResponse constraints used in computing the DOT
constraints.
The length of the list corresponds to the number of DOT constraints, and each entry in the list points to the
corresponding DAKOTA constraint.

6.48.2.10 DakotaRealList DOTOptimizer::constraintMappingMultipliers  [private]

a list of multipliers for mapping the DakotaResponse constraints to the DOT constraints.
The length of the list corresponds to the number of DOT constraints, and each entry in the list contains
a multiplier for the DAKOTA constraint identified with constraintMappingIndices. These multipliers are
currently +1 or -1.

6.48.2.11 DakotaRealList DOTOptimizer::constraintMappingOffsets  [private]

a list of offsets for mapping the DakotaResponse constraints to the DOT constraints.
The length of the list corresponds to the number of DOT constraints, and each entry in the list contains
an offset for the DAKOTA constraint identified with constraintMappingIndices. These offsets involve
inequality bounds or equality targets, since DOT assumes constraint allowables = 0.
The documentation for this class was generated from the following files:

- DOTOptimizer.H
- DOTOptimizer.C
6.49 ErrorTable Struct Reference

Data structure to hold errors.

Public Attributes

- CtelRegexp::RStatus rc
  Enumerated type to hold status codes.

- const char * msg
  Holds character string error message.

6.49.1 Detailed Description

Data structure to hold errors.

This module implements a C++ wrapper for Regular Expressions based on the public domain engine for regular expressions released by: Copyright (c) 1986 by University of Toronto. Written by Henry Spencer. Not derived from licensed software.

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

- CtelRegExp.C
6.50 ForkAnalysisCode Class Reference

Derived class in the AnalysisCode class hierarchy which spawns simulations using forks.

Inheritance diagram for ForkAnalysisCode::

```
  AnalysisCode
    ForkAnalysisCode
```

Public Methods

- **ForkAnalysisCode** (const ProblemDescDB &problem_db)
  - constructor.

- **~ForkAnalysisCode** ()
  - destructor.

- **pid_t fork_program** (const bool block_flag)
  - spawn a child process using fork()/vfork()/execvp() and wait for completion using waitpid() if block_flag is true.

- **void check_status** (const int status)
  - check the exit status of a forked process and abort if an error code was returned.

- **void argument_list** (const int index, const DakotaString &arg)
  - set argList[index] to arg.

- **void tag_argument_list** (const int index, const int tag)
  - append an additional tag to argList[index] (beyond that already present in the modified file names) for managing concurrent analyses within a function evaluation.

Private Attributes

- **const char * argList [4]**
  - an array of strings for use with execvp(char * const arg, char * const *) (an argList entry can be passed as the first argument, and the entire argList can be cast as the second argument).

6.50.1 Detailed Description

Derived class in the AnalysisCode class hierarchy which spawns simulations using forks.
ForkAnalysisCode creates a copy of the parent DAKOTA process using fork()/vfork() and then replaces the copy with a simulation process using execvp(). The parent process can then use waitpid() to wait on completion of the simulation process.

6.50.2 Member Function Documentation

6.50.2.1 void ForkAnalysisCode::check_status (const int status)

check the exit status of a forked process and abort if an error code was returned.

Check to see if the 3-piece interface terminated abnormally (WIFEXITED(status)==0) or if either execvp or the application returned a status code of -1 (WIFEXITED(status)!=0 && (signed char)WEXITSTATUS(status)==-1). If one of these conditions is detected, output a failure message and abort. Note: the application code should not return a status code of -1 unless an immediate abort of dakota is wanted. If for instance, failure capturing is to be used, the application code should write the word “FAIL” to the appropriate results file and return a status code of 0 through exit().

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

- ForkAnalysisCode.H
- ForkAnalysisCode.C
6.51 ForkApplicInterface Class Reference

Derived application interface class which spawns simulation codes using forks.

Inheritance diagram for ForkApplicInterface:

```
DakotaInterface
  |
  |
ApplicationInterface
  |
  |
ForkApplicInterface
```

Public Methods

- **ForkApplicInterface** (const ProblemDescDB &problem_db, const size_t &num_fns)
  
  *constructor.*

- ~**ForkApplicInterface** ()
  
  *destructor.*

- void **derived_map** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int fn_eval_id)
  
  *Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.*

- void **derived_map_asynch** (const ParamResponsePair &pair)
  
  *Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.*

- void **derived_synth** (DakotaPRPList &prp_list)
  
  *For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.*

- void **derived_synth_nowait** (DakotaPRPList &prp_list)
  
  *For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.*

- int **derived_synchronous_local_analysis** (const int &analysis_id)
  
  *Execute a particular analysis (identified by analysis_id) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve_analyses_synch().*
Private Methods

- void derived\_synch\_kernel (DakotaPRPList &prp\_list, const pid\_t pid)
  
  Convenience function for common code between derived\_synch() & derived\_synch\_nowait().

- pid\_t fork\_application (const bool block\_flag)
  
  perform the complete function evaluation by managing the input filter, analysis programs, and output filter.

- void asynchronous\_local\_analyses (const int &start, const int &end, const int &step)
  
  execute analyses asynchronously on the local processor.

- void synchronous\_local\_analyses (const int &start, const int &end, const int &step)
  
  execute analyses synchronously on the local processor.

- void serve\_analyses\_asynch ()
  
  serve the analysis scheduler and execute analysis assignments asynchronously.

Private Attributes

- ForkAnalysisCode forkSimulator
  
  ForkAnalysisCode provides convenience functions for forking individual programs and checking fork exit status.

- DakotaList< pid\_t > processIdList
  
  list of process id’s for asynchronous evaluations; correspondence to evalIdList used for mapping captured fork process id’s to function evaluation id’s.

- DakotalntList evalIdList
  
  list of function evaluation id’s for asynchronous evaluations; correspondence to processIdList used for mapping captured fork process id’s to function evaluation id’s.

6.51.1 Detailed Description

Derived application interface class which spawns simulation codes using forks.

ForkApplicInterface uses a ForkAnalysisCode object for performing simulation invocations.

6.51.2 Member Function Documentation

6.51.2.1 pid\_t ForkApplicInterface::fork\_application (const bool block\_flag) [private]

perform the complete function evaluation by managing the input filter, analysis programs, and output filter.

Manage the input filter, 1 or more analysis programs, and the output filter in blocking or nonblocking mode as governed by block\_flag. In the case of a single analysis and no filters, a single fork is performed, while in other cases, an initial fork is reforked multiple times. Called from derived\_map()
with block_flag == BLOCK and from derived_map_asynch() with block_flag == FALL_THROUGH. Uses ForkAnalysisCode::fork_program() to spawn individual program components within the function evaluation.

6.51.2.2 void ForkApplicInterface::asynchronous_local_analyses (const int & start, const int & end, const int & step) [private]

execute analyses asynchronously on the local processor.
Schedule analyses asynchronously on the local processor using a self-scheduling approach (start to end in step increments). Concurrency is limited by asyncLocalAnalysisConcurrency. Modeled after ApplicationInterface::asynchronous_local_evaluations(). NOTE: This function should be elevated to ApplicationInterface if and when another derived interface class supports asynchronous local analyses.

6.51.2.3 void ForkApplicInterface::synchronous_local_analyses (const int & start, const int & end, const int & step) [private]

execute analyses synchronously on the local processor.
Execute analyses synchronously in succession on the local processor (start to end in step increments). Modeled after ApplicationInterface::synchronous_local_evaluations().

6.51.2.4 void ForkApplicInterface::serve_analyses_asynch () [private]

serve the analysis scheduler and execute analysis assignments asynchronously.
This code runs multiple asynch analyses on each server. It is modeled after ApplicationInterface::serve_evaluations_asynch(). NOTE: This fn should be elevated to ApplicationInterface if and when another derived interface class supports hybrid analysis parallelism.

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

- ForkApplicInterface.H
- ForkApplicInterface.C
6.52 FunctionCompare Class Template Reference

Public Methods

- FunctionCompare (bool (func)(const T &, void *), void *)
  Constructor that defines the pointer to function and search value.

- bool operator() (T t) const
  The operator() must be defined. Calls the function testFunction.

Private Attributes

- bool (* testFunction)(const T &, void *)
  Pointer to test function.

- void * search_val
  Holds the value to search for.

6.52.1 Detailed Description

template<class T> class FunctionCompare< T >

Internal functor to mimic the RW find and index functions using the STL find_if() method. The class holds a pointer to the test function and the search value.

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

- DakotaList.H
Derived class within the DakotaVarConstraints hierarchy which employs the default data view (no variable or domain type array merging).

Inheritance diagram for FundamentalVarConstraints:

```
FundamentalVarConstraints
  |__________________________|
  |                            |
  | DakotaVarConstraints       |
  |                            |
  | VariablesUtil              |
```

Public Methods

- **FundamentalVarConstraints** (const ProblemDescDB &problem_db)
  
  * constructor.

- ~**FundamentalVarConstraints**()
  
  * destructor.

- const DakotaRealVector & **continuous_lower_bounds** () const
  
  * return the active continuous variable lower bounds.

- void **continuous_lower_bounds** (const DakotaRealVector &c_l_bnds)
  
  * set the active continuous variable lower bounds.

- const DakotaRealVector & **continuous_upper_bounds** () const
  
  * return the active continuous variable upper bounds.

- void **continuous_upper_bounds** (const DakotaRealVector &c_u_bnds)
  
  * set the active continuous variable upper bounds.

- const DakotaIntVector & **discrete_lower_bounds** () const
  
  * return the active discrete variable lower bounds.

- void **discrete_lower_bounds** (const DakotaIntVector &d_l_bnds)
  
  * set the active discrete variable lower bounds.

- const DakotaIntVector & **discrete_upper_bounds** () const
  
  * return the active discrete variable upper bounds.

- void **discrete_upper_bounds** (const DakotaIntVector &d_u_bnds)
  
  * set the active discrete variable upper bounds.

- const DakotaRealVector & **inactive_continuous_lower_bounds** () const
return the inactive continuous lower bounds.

- void `inactive_continuous_lower_bounds` (const DakotaRealVector &iclbdns)  
  set the inactive continuous lower bounds.

- const DakotaRealVector & `inactive_continuous_upper_bounds` () const  
  return the inactive continuous upper bounds.

- void `inactive_continuous_upper_bounds` (const DakotaRealVector &icubnds)  
  set the inactive continuous upper bounds.

- const DakotaIntVector & `inactive_discrete_lower_bounds` () const  
  return the inactive discrete lower bounds.

- void `inactive_discrete_lower_bounds` (const DakotaIntVector &idlbnds)  
  set the inactive discrete lower bounds.

- const DakotaIntVector & `inactive_discrete_upper_bounds` () const  
  return the inactive discrete upper bounds.

- void `inactive_discrete_upper_bounds` (const DakotaIntVector &idubnds)  
  set the inactive discrete upper bounds.

- DakotaRealVector `all_continuous_lower_bounds` () const  
  returns a single array with all continuous lower bounds.

- DakotaRealVector `all_continuous_upper_bounds` () const  
  returns a single array with all continuous upper bounds.

- DakotaIntVector `all_discrete_lower_bounds` () const  
  returns a single array with all discrete lower bounds.

- DakotaIntVector `all_discrete_upper_bounds` () const  
  returns a single array with all discrete upper bounds.

- void `write` (ostream &s) const  
  write a variable constraints object to an ostream.

- void `read` (istream &s)  
  read a variable constraints object from an istream.

**Private Attributes**

- bool `nonDFlag`  
  this flag is set if uncertain variables are active (the default is design variables are active; see constructor for logic).

- DakotaRealVector `continuousDesignLowerBnds`  
  the continuous design lower bounds array.
- DakotaRealVector continuousDesignUpperBnds
  the continuous design upper bounds array.

- DakotaIntVector discreteDesignLowerBnds
  the discrete design lower bounds array.

- DakotaIntVector discreteDesignUpperBnds
  the discrete design upper bounds array.

- DakotaRealVector uncertainDistLowerBnds
  the uncertain distribution lower bounds array.

- DakotaRealVector uncertainDistUpperBnds
  the uncertain distribution upper bounds array.

- DakotaRealVector continuousStateLowerBnds
  the continuous state lower bounds array.

- DakotaRealVector continuousStateUpperBnds
  the continuous state upper bounds array.

- DakotaIntVector discreteStateLowerBnds
  the discrete state lower bounds array.

- DakotaIntVector discreteStateUpperBnds
  the discrete state upper bounds array.

6.53.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which employs the default data view (no variable or domain type array merging).

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The FundamentalVarConstraints derived class separates the design, uncertain, and state variable types as well as the continuous and discrete domain types. The result is separate lower and upper bounds arrays for continuous design, discrete design, uncertain, continuous state, and discrete state variables. This is the default approach, so all iterators and strategies not specifically utilizing the All, Merged, or AllMerged views use this approach (see DakotaVariables::get_variables(problem_db) for variables type selection; variables type is passed to the DakotaVarConstraints constructor in DakotaModel).

6.53.2 Constructor & Destructor Documentation
6.53.2.1 FundamentalVarConstraints::FundamentalVarConstraints (const ProblemDescDB & problem_db)

constructor.

Extract fundamental lower and upper bounds (VariablesUtil is not used).

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

- FundamentalVarConstraints.H
- FundamentalVarConstraints.C
6.54 FundamentalVariables Class Reference

Derived class within the DakotaVariables hierarchy which employs the default data view (no variable or domain type array merging).

Inheritance diagram for FundamentalVariables::

```
DakotaVariables
    VariablesUtil
        FundamentalVariables
```

Public Methods

- **FundamentalVariables ()**
  
  `default constructor.`

- **FundamentalVariables (const ProblemDescDB &problem_db)**
  
  `standard constructor.`

- **~FundamentalVariables ()**
  
  `destructor.`

- **size tv () const**
  
  `Returns total number of vars.`

- **size cv () const**
  
  `Returns number of active continuous vars.`

- **size dv () const**
  
  `Returns number of active discrete vars.`

- **const DakotaRealVector & continuous variables () const**
  
  `return the active continuous variables.`

- **void continuous variables (const DakotaRealVector &c_vars)**
  
  `set the active continuous variables.`

- **const DakotaIntVector & discrete variables () const**
  
  `return the active discrete variables.`

- **void discrete variables (const DakotaIntVector &d_vars)**
  
  `set the active discrete variables.`

- **const DakotaStringArray & continuous_variable_labels () const**
return the active continuous variable labels.

- void continuous_variable_labels (const DakotaStringArray &cv_labels)
  set the active continuous variable labels.

- const DakotaStringArray & discrete_variable_labels ()
  return the active discrete variable labels.

- void discrete_variable_labels (const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

- const DakotaRealVector & inactive_continuous_variables ()
  return the inactive continuous variables.

- void inactive_continuous_variables (const DakotaRealVector &ic_vars)
  set the inactive continuous variables.

- const DakotaIntVector & inactive_discrete_variables ()
  return the inactive discrete variables.

- void inactive_discrete_variables (const DakotaIntVector &id_vars)
  set the inactive discrete variables.

- size_t acv ()
  returns total number of continuous vars.

- size_t adv ()
  returns total number of discrete vars.

- DakotaRealVector all_continuous_variables ()
  returns a single array with all continuous variables.

- DakotaIntVector all_discrete_variables ()
  returns a single array with all discrete variables.

- DakotaStringArray all_continuous_variable_labels ()
  returns a single array with all continuous variable labels.

- DakotaStringArray all_discrete_variable_labels ()
  returns a single array with all discrete variable labels.

- void read (istream &s)
  read a variables object from an istream.

- void write (ostream &s) const
  write a variables object to an ostream.

- void read_annotated (istream &s)
  read a variables object in annotated format from an istream.
void write_annotated (ostream &s) const
write a variables object in annotated format to an ostream.

void read (DakotaBiStream &s)
read a variables object from the binary restart stream.

void write (DakotaBoStream &s) const
write a variables object to the binary restart stream.

void read (UnPackBuffer &s)
read a variables object from a packed MPI buffer.

void write (PackBuffer &s) const
write a variables object to a packed MPI buffer.

Private Methods

void copy_rep (const DakotaVariables *vars_rep)
Used by copy() to copy the contents of a letter class.

Private Attributes

bool nonDFlag
this flag is set if uncertain variables are active (the default is design variables are active; see constructor for logic).

DakotaRealVector continuousDesignVars
the continuous design variables array.

DakotaIntVector discreteDesignVars
the discrete design variables array.

DakotaRealVector uncertainVars
the uncertain variables array.

DakotaRealVector continuousStateVars
the continuous state variables array.

DakotaIntVector discreteStateVars
the discrete state variables array.

DakotaStringArray continuousDesignLabels
the continuous design variables label array.

DakotaStringArray discreteDesignLabels
the discrete design variables label array.
• DakotaStringArray uncertainLabels
  the uncertain variables label array.

• DakotaStringArray continuousStateLabels
  the continuous state variables label array.

• DakotaStringArray discreteStateLabels
  the discrete state variables label array.

**Friends**

• bool operator==(const FundamentalVariables &vars1, const FundamentalVariables &vars2)
  equality operator.

### 6.54.1 Detailed Description

Derived class within the DakotaVariables hierarchy which employs the default data view (no variable or domain type array merging).

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The FundamentalVariables derived class separates the design, uncertain, and state variable types as well as the continuous and discrete domain types. The result is separate arrays for continuous design, discrete design, uncertain, continuous state, and discrete state variables. This is the default approach, so all iterators and strategies not specifically utilizing the All, Merged, or AllMerged views use this approach (see DakotaVariables::get_variables(problem_db)).

### 6.54.2 Constructor & Destructor Documentation

#### 6.54.2.1 FundamentalVariables::FundamentalVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels (VariablesUtil is not used).

### 6.54.3 Friends And Related Function Documentation

#### 6.54.3.1 bool operator==(const FundamentalVariables & vars1, const FundamentalVariables & vars2) [friend]

equality operator.

Check each fundamental array using operator== from data_types.C. Labels are ignored.

The documentation for this class was generated from the following files:
- FundamentalVariables.H
- FundamentalVariables.C
6.55 GetLongOpt Class Reference

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France).

Inheritance diagram for GetLongOpt::

```
GetLongOpt
    CommandLineHandler
```

Public Types

- enum OptType { Valueless, OptionalValue, MandatoryValue }
  
  enum for different types of values associated with command line options.

Public Methods

- GetLongOpt (const char optmark='\'-\')
  
  Constructor.

- ~GetLongOpt ()
  
  Destructor.

- int parse (int argc, char *const *argv)
  
  parse the command line args (argc, argv).

- int parse (char *const str, char *const p)
  
  parse a string of options (typically given from the environment).

- int enroll (const char *const opt, const OptType t, const char *const desc, const char *const val)
  
  Add an option to the list of valid command options.

- const char * retrieve (const char *const opt) const
  
  Retrieve value of option.

- void usage (ostream &outfile=cout) const
  
  Print usage information to outfile.

- void usage (const char *str)
  
  Change header of usage output to str.
Private Methods

- char * basename (char *const p) const
  extract the base name from a string as delimited by '/'.

- int setcell (Cell *c, char *valtoken, char *nexttoken, const char *p)
  internal convenience function for setting Cell::value.

Private Attributes

- Cell * table
  option table.

- const char * ustring
  usage message.

- char * pname
  program basename.

- char optmarker
  option marker.

- int enroll_done
  finished enrolling.

- Cell * last
  last entry in option table.

6.55.1 Detailed Description

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France).

GetLongOpt manages the definition and parsing of “long options.” Command line options can be abbreviated as long as there is no ambiguity. If an option requires a value, the value should be separated from the option either by whitespace or an "=".

6.55.2 Constructor & Destructor Documentation

6.55.2.1 GetLongOpt::GetLongOpt (const char optmark = '‐')

Constructor.

Constructor for GetLongOpt takes an optional argument: the option marker. If unspecified, this defaults to '‐', the standard (?) Unix option marker.
6.55.3  Member Function Documentation

6.55.3.1  int GetLongOpt::parse (int argc, char *const * argv)

parse the command line args (argc, argv).
A return value < 1 represents a parse error. Appropriate error messages are printed when errors are seen.
parse returns the the optind (see getopt(3)) if parsing is successful.

6.55.3.2  int GetLongOpt::parse (char *const str, char *const p)

parse a string of options (typically given from the environment).
A return value < 1 represents a parse error. Appropriate error messages are printed when errors are seen.
parse takes two strings: the first one is the string to be parsed and the second one is a string to be prefixed
to the parse errors.

6.55.3.3  int GetLongOpt::enroll (const char *const opt, const OptType t, const char *const desc, const char *const val)

Add an option to the list of valid command options.
enroll adds option specifications to its internal database. The first argument is the option string. The second
is an enum saying if the option is a flag (Valueless), if it requires a mandatory value (MandatoryValue) or
if it takes an optional value (OptionalValue). The third argument is a string giving a brief description of the
option. This description will be used by GetLongOpt::usage. GetLongOpt, for usage-printing, uses {${val}}
to represent values needed by the options. {<$val>} is a mandatory value and {[$val]} is an optional
value. The final argument to enroll is the default string to be returned if the option is not specified. For
flags (options with Valueless), use "" (empty string, or in fact any arbitrary string) for specifying TRUE
and 0 (null pointer) to specify FALSE.

6.55.3.4  const char * GetLongOpt::retrieve (const char *const opt) const

Retrieve value of option.
The values of the options that are enrolled in the database can be retrieved using retrieve. This returns a
string and this string should be converted to whatever type you want. See atoi, atof, atol, etc. If a "parse"
is not done before retrieving all you will get are the default values you gave while enrolling! Ambiguities
while retrieving (may happen when options are abbreviated) are resolved by taking the matching option
that was enrolled last. For example, -{v} will expand to {-verify}. If you try to retrieve something you
didn’t enroll, you will get a warning message.

6.55.3.5  void GetLongOpt::usage (const char * str) [inline]

Change header of usage output to str.
GetLongOpt::usage is overloaded. If passed a string "str", it sets the internal usage string to "str". Other-
wise it simply prints the command usage.
The documentation for this class was generated from the following files:
- CommandLineHandler.H
- CommandLineHandler.C
6.56 HermiteSurf Class Reference

Derived approximation class for Hermite polynomials (global approximation).

Inheritance diagram for HermiteSurf:

```
DakotaApproximation
   |       |
   |       |
   |       | HermiteSurf
```

Public Methods

- \texttt{HermiteSurf} (const \texttt{ProblemDescDB} &\texttt{problem\_db}, const size\_t \&\texttt{num\_acv})
  \textit{constructor}.

- \texttt{\sim{}HermiteSurf} ()
  \textit{destructor}.

Protected Methods

- int \texttt{required\_samples} ()
  \textit{return the minimum number of samples required to build the derived class approximation type in numVars dimensions.}

- const DakotaRealVector \& \texttt{approximation\_coefficients} ()
  \textit{return the coefficient array computed by \texttt{find\_coefficients}().}

- void \texttt{find\_coefficients} ()
  \textit{find the Polynomial Chaos coefficients for the response surface.}

- Real \texttt{get\_value} (const DakotaRealVector \&\texttt{x})
  \textit{retrieve the function value for a given parameter set \texttt{x}.}

Private Methods

- void \texttt{get\_num\_chaos} ()
  \textit{calculate number of Chaos according to the highest order of Chaos.}

- DakotaRealVector \texttt{get\_chaos} (const DakotaRealVector \&\texttt{x}, int order)
  \textit{calculate the Polynomial Chaos from variables.}
Private Attributes

- DakotaRealVector chaosCoeffs
  
  `numChaos` entries.

- DakotaRealVectorArray chaosSamples
  
  `numChaos*numCurrentPoints` entries.

- int numChaos
  
  `Number of terms in Polynomial Chaos Expansion.`

- int highestOrder
  
  `Highest order of Hermite Polynomials in Expansion`.

6.56.1 Detailed Description

Derived approximation class for Hermite polynomials (global approximation).

The HermiteSurf class provides a global approximation based on Hermite polynomials. It is used primarily for polynomial chaos expansions (for stochastic finite element approaches to uncertainty quantification).

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

- HermiteSurf.H
- HermiteSurf.C
6.57 HierLayeredModel Class Reference

Derived model class within the layered model branch for managing hierarchical surrogates (models of varying fidelity).

Inheritance diagram for HierLayeredModel::

```
HierLayeredModel
    |      |
    |      | DakotaModel
    |      | LayeredModel
    |      | HierLayeredModel
```

Public Methods

- **HierLayeredModel (ProblemDescDB &problem_db)**
  
  *constructor.*

- ~**HierLayeredModel ()**
  
  *destructor.*

Protected Methods

- void **derived::compute_response** (const DakotaIntArray &asv)
  
  *portion of compute_response() specific to HierLayeredModel.*

- void **derived::asynch_compute_response** (const DakotaIntArray &asv)
  
  *portion of asynch_compute_response() specific to HierLayeredModel.*

- const DakotaResponseArray & **derived::synchronize ()**
  
  *portion of synchronize() specific to HierLayeredModel.*

- const DakotaResponseList & **derived::synchronize_nowait ()**
  
  *portion of synchronize_nowait() specific to HierLayeredModel.*

- DakotaModel & **subordinate_model ()**
  
  *return highFidelityModel to SurrBasedOptStrategy.*

- void **build_approximation ()**
  
  *use highFidelityModel to compute the truth values needed for correction of lowFidelityInterface results.*

- DakotaString **local_eval_synchronization ()**
  
  *return lowFidelityInterface local evaluation synchronization setting.*
- const DakotaIntList & synchronize_nowait_completions ()
  
  return completion id’s matching response list from synchronize_nowait (request forwarded to lowFidelityInterface).

- bool derived_master_overload () const
  
  flag which prevents overloading the master with a multiprocessor evaluation (request forwarded to lowFidelityInterface).

- void derived_init_communicators (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  
  portion of init_communicators() specific to HierLayeredModel (request forwarded to lowFidelityInterface).

- void free_communicators ()
  
  deallocate communicator partitions for the HierLayeredModel (request forwarded to lowFidelityInterface).

- void serve ()
  
  Service job requests received from the master. Completes when a termination message is received from stop_servers() (request forwarded to lowFidelityInterface).

- void stop_servers ()
  
  executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (request forwarded to lowFidelityInterface).

- int total_eval_counter () const
  
  return the total evaluation count for the HierLayeredModel (request forwarded to lowFidelityInterface).

- int new_eval_counter () const
  
  return the new evaluation count for the HierLayeredModel (request forwarded to lowFidelityInterface).

**Private Attributes**

- DakotaInterface lowFidelityInterface
  
  manages the approximate low fidelity function evaluations.

- DakotaModel highFidelityModel
  
  provides truth evaluations for computing corrections to the low fidelity results.

- DakotaResponse highFidResponse
  
  the high fidelity response is computed in build_approximation() and needs class scope for use in automatic surrogate construction in derived compute_response functions.

- DakotaIntList evalIdList
  
  bookkeeps fnEvalId’s for correction of asynchronous low fidelity evaluations.
6.57.1 Detailed Description

Derived model class within the layered model branch for managing hierarchical surrogates (models of varying fidelity).

The HierLayeredModel class manages hierarchical models of varying fidelity. In particular, it uses a low fidelity model as a surrogate for a high fidelity model. The class contains a lowFidelityInterface which manages the approximate low fidelity function evaluations and a highFidelityModel which provides truth evaluations for computing corrections to the low fidelity results.

6.57.2 Member Function Documentation

6.57.2.1 void HierLayeredModel::derived_compute_response (const DakotaIntArray & asv) [protected, virtual]
portion of compute_response() specific to HierLayeredModel.
Evaluate the approximate response using lowFidelityInterface, compute the high fidelity response with build_approximation() (if not performed previously), and, if correction is active, correct the low fidelity results.
Reimplemented from DakotaModel.

6.57.2.2 void HierLayeredModel::derived_asynch_compute_response (const DakotaIntArray & asv) [protected, virtual]
portion of asynch_compute_response() specific to HierLayeredModel.
Evaluate the approximate response using an asynchronous lowFidelityInterface mapping and compute the high fidelity response with build_approximation() (for correcting the low fidelity results in derived_synchronize() and derived_synchronize_nowait() if not performed previously.
Reimplemented from DakotaModel.

6.57.2.3 const DakotaResponseArray & HierLayeredModel::derived_synchronize () [protected, virtual]
portion of synchronize() specific to HierLayeredModel.
Perform a blocking retrieval of all asynchronous evaluations from lowFidelityInterface and, if automatic correction is on, apply correction to each response in the array.
Reimplemented from DakotaModel.

6.57.2.4 const DakotaResponseList & HierLayeredModel::derived_synchronize_nowait () [protected, virtual]
portion of synchronize_nowait() specific to HierLayeredModel.
Perform a nonblocking retrieval of currently available asynchronous evaluations from lowFidelityInterface and, if automatic correction is on, apply correction to each response in the list.
Reimplemented from DakotaModel.
The documentation for this class was generated from the following files:

- HierLayeredModel.H
- HierLayeredModel.C
6.58 KrigApprox Class Reference

Utility class for kriging interpolation.

Public Methods

- **KrigApprox** (int, int, const DakotaRealVector &, const DakotaRealVector &, const DakotaRealVector &)
  
  *constructor.*

- **~KrigApprox** ()
  
  *destructor.*

- void **ModelBuild** (int, int, const DakotaRealVector &, const DakotaRealVector &, bool)
  
  *Function to compute vector and matrix terms in the kriging surface.*

- Real **ModelApply** (int, int, const DakotaRealVector &)
  
  *Function returns a response value using the kriging surface.*

Private Attributes

- int **N1**
  
  *Size variable for CONMIN arrays. See CONMIN manual.*

- int **N2**
  
  *Size variable for CONMIN arrays. See CONMIN manual.*

- int **N3**
  
  *Size variable for CONMIN arrays. See CONMIN manual.*

- int **N4**
  
  *Size variable for CONMIN arrays. See CONMIN manual.*

- int **N5**
  
  *Size variable for CONMIN arrays. See CONMIN manual.*

- int **conminSingleArray**
  
  *Array size parameter needed in interface to CONMIN.*

- int **numcon**
  
  *CONMIN variable: Number of constraints.*

- int **NFDG**
  
  *CONMIN variable: Finite difference flag.*
- **int IPRINT**
  *CONMIN* variable: Flag to control amount of output data.

- **int ITMAX**
  *CONMIN* variable: Flag to specify the maximum number of iterations.

- **Real FDCH**
  *CONMIN* variable: Relative finite difference step size.

- **Real FDCHM**
  *CONMIN* variable: Absolute finite difference step size.

- **Real CT**
  *CONMIN* variable: Constraint thickness parameter.

- **Real CTMIN**
  *CONMIN* variable: Minimum absolute value of CT used during optimization.

- **Real CTL**
  *CONMIN* variable: Constraint thickness parameter for linear and side constraints.

- **Real CTLMIN**
  *CONMIN* variable: Minimum value of CTL used during optimization.

- **Real DELFUN**
  *CONMIN* variable: Relative convergence criterion threshold.

- **Real DABFUN**
  *CONMIN* variable: Absolute convergence criterion threshold.

- **int conminInfo**
  *CONMIN* variable: status flag for optimization.

- **Real * S**
  Internal *CONMIN* array.

- **Real * G1**
  Internal *CONMIN* array.

- **Real * G2**
  Internal *CONMIN* array.

- **Real * B**
  Internal *CONMIN* array.

- **Real * C**
  Internal *CONMIN* array.

- **int * MS1**
  Internal *CONMIN* array.
- Real * SCAL
  Internal CONMIN array.

- Real * DF
  Internal CONMIN array.

- Real * A
  Internal CONMIN array.

- int * ISC
  Internal CONMIN array.

- int * IC
  Internal CONMIN array.

- Real * conminThetaVars
  Temporary array of design variables used by CONMIN (length NI = numdv+2).

- Real * conminThetaLowerBnds
  Temporary array of lower bounds used by CONMIN (length NI = numdv+2).

- Real * conminThetaUpperBnds
  Temporary array of upper bounds used by CONMIN (length NI = numdv+2).

- Real ALPHAX
  Internal CONMIN variable: 1-D search parameter.

- Real ABOBJ1
  Internal CONMIN variable: 1-D search parameter.

- Real THETA
  Internal CONMIN variable: mean value of push-off factor.

- Real PHI
  Internal CONMIN variable: "participation coefficient".

- int NSIDE
  Internal CONMIN variable: side constraints parameter.

- int NSCAL
  Internal CONMIN variable: scaling control parameter.

- int NACMX1
  Internal CONMIN variable: estimate of 1+(max # of active constraints).

- int LINOBJ
  Internal CONMIN variable: linear objective function identifier (unused).

- int ITRM
Internal CONMIN variable: diminishing return criterion iteration number.

- int ICNDIR
  Internal CONMIN variable: conjugate direction restart parameter.

- int IGOTO
  Internal CONMIN variable: internal optimization termination flag.

- int NAC
  Internal CONMIN variable: number of active and violated constraints.

- int INFOG
  Internal CONMIN variable: gradient information flag.

- int ITER
  Internal CONMIN variable: iteration count.

- int iFlag
  Fortran77 flag for kriging computations.

- Real betaHat
  Estimate of the beta term in the kriging model.

- Real maxLikelihoodEst
  Error term computed via Maximum Likelihood Estimation.

- int numNewPts
  Size variable for the arrays used in kriging computations.

- int numSampQuad
  Size variable for the arrays used in kriging computations.

- Real * thetaVector
  Array of correlation parameters for the kriging model.

- Real * xMatrix
  A 2-D array of design points used to build the kriging model.

- Real * yValueVector
  Array of response values corresponding to the array of design points.

- Real * xNewVector
  A 2-D array of design points where the kriging model will be evaluated.

- Real * yNewVector
  Array of response values corresponding to the design points specified in xNewVector.

- Real * thetaLoBndVector
  Array of lower bounds in optimizer-to-kriging interface.
• Real * thetaUpBndVector
  
  Array of upper bounds in optimizer-to-kriging interface.

• Real * constraintVector

  Array of constraint values (used with optimizer).

• Real * rhsTermsVector

  Internal array for kriging Fortran77 code: matrix algebra result.

• int * iPivotVector

  Internal array for kriging Fortran77 code: pivot vector for linear algebra.

• Real * correlationMatrix

  Internal array for kriging Fortran77 code: correlation matrix.

• Real * invcorrelMatrix

  Internal array for kriging Fortran77 code: inverse correlation matrix.

• Real * fValueVector

  Internal array for kriging Fortran77 code: response value vector.

• Real * fRinvVector

  Internal array for kriging Fortran77 code: vector matrix result.

• Real * yfbVector

  Internal array for kriging Fortran77 code: vector arithmetic result.

• Real * yfbRinvVector

  Internal array for kriging Fortran77 code: vector matrix result.

• Real * rXhatVector

  Internal array for kriging Fortran77 code: local correlation vector.

• Real * workVector

  Internal array for kriging Fortran77 code: temporary storage.

• Real * workVectorQuad

  Internal array for kriging Fortran77 code: temporary storage.

• int * iworkVector

  Internal array for kriging Fortran77 code: temporary storage.

### 6.58.1 Detailed Description

Utility class for kriging interpolation.

The KrigApprox class provides utilities for the KrigingSurf class. It is based on the Ph.D. thesis work of Tony Giunta.
6.58.2 Member Function Documentation

6.58.2.1 Real KrigApprox::ModelApply (int, int, const DakotaRealVector &)

Function returns a response value using the kriging surface. The response value is computed at the design point specified by the DakotaRealVector function argument.

6.58.3 Member Data Documentation

6.58.3.1 int KrigApprox::N1 [private]

Size variable for CONMIN arrays. See CONMIN manual. N1 = number of variables + 2

6.58.3.2 int KrigApprox::N2 [private]

Size variable for CONMIN arrays. See CONMIN manual. N2 = number of constraints + 2*(number of variables)

6.58.3.3 int KrigApprox::N3 [private]

Size variable for CONMIN arrays. See CONMIN manual. N3 = Maximum possible number of active constraints.

6.58.3.4 int KrigApprox::N4 [private]

Size variable for CONMIN arrays. See CONMIN manual. N4 = Maximum(N3,number of variables)

6.58.3.5 int KrigApprox::N5 [private]

Size variable for CONMIN arrays. See CONMIN manual. N5 = 2*(N4)

6.58.3.6 Real KrigApprox::CT [private]

CONMIN variable: Constraint thickness parameter. The value of CT decreases in magnitude during optimization.
6.58.3.7  **Real** KrigApprox::S  [private]

Internal CONMIN array.
Move direction in N-dimensional space.

6.58.3.8  **Real** KrigApprox::G1  [private]

Internal CONMIN array.
Temporary storage of constraint values.

6.58.3.9  **Real** KrigApprox::G2  [private]

Internal CONMIN array.
Temporary storage of constraint values.

6.58.3.10  **Real** KrigApprox::B  [private]

Internal CONMIN array.
Temporary storage for computations involving array S.

6.58.3.11  **Real** KrigApprox::C  [private]

Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.58.3.12  **int** KrigApprox::MS1  [private]

Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.58.3.13  **Real** KrigApprox::SCAL  [private]

Internal CONMIN array.
Vector of scaling parameters for design parameter values.

6.58.3.14  **Real** KrigApprox::DF  [private]

Internal CONMIN array.
Temporary storage for analytic gradient data.

6.58.3.15  **Real** KrigApprox::A  [private]

Internal CONMIN array.
Temporary 2-D array for storage of constraint gradients.
6.58.3.16 int KrigApprox::ISC [private]

Internal CONMIN array.
Array of flags to identify linear constraints. (not used in this implementation of CONMIN)

6.58.3.17 int KrigApprox::IC [private]

Internal CONMIN array.
Array of flags to identify active and violated constraints

6.58.3.18 int KrigApprox::iFlag [private]

Fortran77 flag for kriging computations.
iFlag=1 computes vector and matrix terms for the kriging surface, iFlag=2 computes the response value (using kriging) at the user-supplied design point.
The documentation for this class was generated from the following files:

- KSMSurf.H
- KSMSurf.C
6.59 KrigingSurf Class Reference

Derived approximation class for kriging interpolation.

Inheritance diagram for KrigingSurf:

```
KrigingSurf
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
DakotaApproximation |   KrigingSurf   |
```

Public Methods

- **KrigingSurf** (const ProblemDescDB &problem, const size_t &num_acv)
  
  *constructor.*

- **~KrigingSurf**()
  
  *destructor.*

Protected Methods

- void findCoefficients()
  
  *calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.*

- int requiredSamples()
  
  *return the minimum number of samples required to build the derived class approximation type in numVars dimensions.*

- Real getValue(const DakotaRealVector &x)
  
  *retrieve the approximate function value for a given parameter vector.*

Private Attributes

- **KrigApprox * krigObject**
  
  *Kriging Surface object declaration.*

- **DakotaRealVector x_matrix**
  
  *A 2-d array of all sample sites (design points) used to create the kriging surface.*

- **DakotaRealVector f_of_x_array**
  
  *An array of response values; one response value per sample site.*

- **DakotaRealVector correlationVector**
An array of correlation parameter values used to build the kriging surface.

- bool runConminFlag
  
  Flag to run CONMIN (value=1) or use user-supplied correlations (value=0).

### 6.59.1 Detailed Description

Derived approximation class for kriging interpolation.

The KrigingSurf class uses a the kriging approach to interpolate between data points. It is based on the Ph.D. thesis work of Tony Giunta.

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

- KSMSurf.H
- KSMSurf.C
6.60 LayeredModel Class Reference

Base class for the layered models (SurrLayeredModel and HierLayeredModel).

Inheritance diagram for LayeredModel::

```
  DakotaModel
    LayeredModel
      HierLayeredModel
        SurrLayeredModel
```

Protected Methods

- `LayeredModel(ProblemDescDB &problem_db)`
  
  *constructor.*

- `~LayeredModel()`
  
  *destructor.*

- `void compute_correction(const DakotaResponse &truth_response, const DakotaResponse &approx_response, const DakotaRealVector &c_vars)`
  
  *compute the correction required to bring approx_response into agreement with truth_response.*

- `void apply_correction(DakotaResponse &approx_response, const DakotaRealVector &c_vars, bool quiet_flag=0)`
  
  *apply the correction computed in compute_correction() to approx_response.*

- `void check_submodel_compatibility(const DakotaModel &sub_model)`
  
  *verify compatibility between LayeredModel attributes and attributes of the submodel (SurrLayeredModel::actualModel or HierLayeredModel::highFidelityModel).*

- `bool force_rebuild()`
  
  *evaluate whether a rebuild of the approximation should be forced based on changes in the inactive data.*

- `void auto_correction(bool correction_flag)`
  
  *sets autoCorrection to ON (1) or OFF (0).*

Protected Attributes

- `DakotaResponseArray correctedResponseArray`
  
  *array of corrected responses used in derived_synchronize() functions.*
- DakotaResponseList correctedResponseList
  list of corrected responses used in derived synchronize nowait() functions.

- DakotaRealVectorList rawCVarsList
  list of raw continuous variables used by apply_correction(). DakotaModel::varsList cannot be used for this purpose since it does not contain lower level variables sets from finite differencing.

- DakotaString correctionType
  approximation correction approach to be used: additive or multiplicative.

- DakotaString correctionOrder
  approximation correction order to be used: zeroth or first.

- int approxBuilds
  number of calls to build_approximation().

- bool autoCorrection
  a flag which controls the use of apply_correction() in SurrLayeredModel and HierLayeredModel approximate response computations.

- DakotaString approxType
  approximation type identifier string: global, local, or hierarchical.

- DakotaString refitInactive
  flag denoting a user setting for rebuilding the approximation when changes occur to the inactive variables data.

- DakotaRealVector fitInactiveCVars
  stores a copy of the inactive continuous variables when the approximation is built; used to detect when a rebuild is required.

- DakotaRealVector fitInactiveCLowerBnds
  stores a copy of the inactive continuous lower bounds when the approximation is built; used to detect when a rebuild is required.

- DakotaRealVector fitInactiveCUpperBnds
  stores a copy of the inactive continuous upper bounds when the approximation is built; used to detect when a rebuild is required.

- DakotaIntVector fitInactiveDVars
  stores a copy of the inactive discrete variables when the approximation is built; used to detect when a rebuild is required.

- DakotaIntVector fitInactiveDLowerBnds
  stores a copy of the inactive discrete lower bounds when the approximation is built; used to detect when a rebuild is required.

- DakotaIntVector fitInactiveDUpperBnds
  stores a copy of the inactive discrete upper bounds when the approximation is built; used to detect when a rebuild is required.
Private Attributes

- DakotaRealVector offsetValues
  values used to offset the function values in an approximate response in order to apply a truth model correction.

- DakotaRealVector scaleFactors
  values used to scale the function values, gradients, and Hessians in an approximate response in order to apply a truth model correction.

- bool correctionComputed
  flag used to indicate whether or not a correction is available.

- bool badScalingFlag
  flag used to indicate function values near zero for multiplicative corrections; triggers an automatic switch to additive corrections.

- DakotaRealVector betaFns
  1st-order correction term: If multiplicative (beta), then equals the ratio of high fidelity to low fidelity model values at $x=x_{\text{center}}$. If additive, then equals the difference between high and low fidelity model values at $x=x_{\text{center}}$.

- DakotaRealMatrix betaGrads
  1st-order correction term: If multiplicative (beta), then equals the gradient of the high/low function ratio at $x=x_{\text{center}}$. If additive, then equals the gradient of the high/low function difference at $x=x_{\text{center}}$.

- DakotaRealVector betaCenterPt
  1st-order correction term: center point of the trust region.

- DakotaRealVector betaApproxCenterVals
  Function values at the center of the trust region which are needed as a fall back if the current function values are unavailable when applying the beta-correction.

- DakotaRealMatrix betaApproxCenterGrads
  Gradient values at the center of the trust region which are needed as a fall back if the current function gradients are unavailable when applying the beta-correction.

6.60.1 Detailed Description

Base class for the layered models (SurrLayeredModel and HierLayeredModel).
The LayeredModel class provides common functions to derived classes for computing and applying corrections to approximations.

6.60.2 Member Function Documentation
compute the correction required to bring approx_response into agreement with truth_response.

Compute a correction for approximate responses based on an offset, scaled, or beta correction approach. The offset and scaled approaches will correct the approximate function values to match the truth function values at a single point in the parameter space (e.g., the center of a trust region). In the "beta" correction approach, the function value and the function gradient are matched at a single point. The beta-correction is similar to the scaled-correction method, but the scaled-correction uses a scalar value for each response function, whereas the beta-correction uses a scaling function for each response function that varies w.r.t. position in the parameter space.

Reimplemented from DakotaModel.

evaluate whether a rebuild of the approximation should be forced based on changes in the inactive data.

This function forces a rebuild of the approximation according to the approximation type, the refitInactive setting, and whether any inactive data has changed since the last build.

### Member Data Documentation

#### int LayeredModel::approxBuilds [protected]

number of calls to build_approximation().

used as a flag to automatically build the approximation if one of the derived compute_response functions is called prior to build_approximation().

#### bool LayeredModel::autoCorrection [protected]

a flag which controls the use of apply_correction() in SurrLayeredModel and HierLayeredModel approximate response computations.

the default is ON once compute_correction() has been called. However this should be overridden when a new correction is desired, since compute_correction() no longer automatically backs out an old correction.

#### DakotaString LayeredModel::refitInactive [protected]

flag denoting a user setting for rebuilding the approximation when changes occur to the inactive variables data.

A setting of "all" denotes that the approximation should be rebuilt every time the inactive variables change (e.g., for each instance of \(d\) in OUU). A setting of "region" denotes that the approximation should be rebuilt every time the bounded region for the inactive variables changes (e.g., for each new trust region on \(d\) in OUU).

The documentation for this class was generated from the following files:
- LayeredModel.H
- LayeredModel.C
6.61 MARSSurf Class Reference

Derived approximation class for multivariate adaptive regression splines.

Inheritance diagram for MARSSurf::

```
DakotaApproximation
    ---->
       |
   MARSSurf
```

**Public Methods**

- **MARSSurf** (const ProblemDescDB &problem_db, const size_t &num_acv)
  constructor.

- **~MARSSurf** ()
  destructor.

**Protected Methods**

- int **required_samples** ()
  return the minimum number of samples required to build the derived class approximation type in numVars dimensions.

- void **find_coefficients** ()
  calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

- Real **get_value** (const DakotaRealVector &x)
  retrieve the approximate function value for a given parameter vector.

**Private Attributes**

- int * **flags**
  variable type declarations (ordinal, excluded, categorical).

- Mars * **marsObject**
  pointer to the Mars object (MARS wrapper provided as part of DDACE).
6.61.1 Detailed Description

Derived approximation class for multivariate adaptive regression splines.

The MARSSurf class provides a global approximation based on regression splines. It employs the C++ wrapper developed by the DDACE team for the Multivariate Adaptive Regression Splines (MARS) package from Prof. Jerome Friedman of Stanford University Dept. of Statistics.

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

- MARSSurf.H
- MARSSurf.C
6.62  MergedVarConstraints Class Reference

Derived class within the DakotaVarConstraints hierarchy which employs the merged data view.
Inheritance diagram for MergedVarConstraints::

```
DakotaVarConstraints
    VariablesUtil
        MergedVarConstraints
```

Public Methods

- **MergedVarConstraints** (const ProblemDescDB &problem_db)
  
  *constructor.*

- **~MergedVarConstraints** ()
  
  *destructor.*

- const DakotaRealVector & **continuous_lower_bounds** () const
  
  *return the active continuous variable lower bounds.*

- void **continuous_lower_bounds** (const DakotaRealVector &c_lbounds)
  
  *set the active continuous variable lower bounds.*

- const DakotaRealVector & **continuous_upper_bounds** () const
  
  *return the active continuous variable upper bounds.*

- void **continuous_upper_bounds** (const DakotaRealVector &c_ubounds)
  
  *set the active continuous variable upper bounds.*

- const DakotaIntVector & **discrete_lower_bounds** () const
  
  *return the active discrete variable lower bounds.*

- void **discrete_lower_bounds** (const DakotaIntVector &d_lbounds)
  
  *set the active discrete variable lower bounds.*

- const DakotaIntVector & **discrete_upper_bounds** () const
  
  *return the active discrete variable upper bounds.*

- void **discrete_upper_bounds** (const DakotaIntVector &d_ubounds)
  
  *set the active discrete variable upper bounds.*

- const DakotaRealVector & **inactive_continuous_lower_bounds** () const
  
  *return the inactive continuous lower bounds.*
• void inactive_continuous_lower_bounds (const DakotaRealVector &i_c_l_bnds)  
  set the inactive continuous lower bounds.

• const DakotaRealVector & inactive_continuous_upper_bounds () const  
  return the inactive continuous upper bounds.

• void inactive_continuous_upper_bounds (const DakotaRealVector &i_c_u_bnds)  
  set the inactive continuous upper bounds.

• const DakotaIntVector & inactive_discrete_lower_bounds () const  
  return the inactive discrete lower bounds.

• void inactive_discrete_lower_bounds (const DakotaIntVector &i_d_l_bnds)  
  set the inactive discrete lower bounds.

• const DakotaIntVector & inactive_discrete_upper_bounds () const  
  return the inactive discrete upper bounds.

• void inactive_discrete_upper_bounds (const DakotaIntVector &i_d_u_bnds)  
  set the inactive discrete upper bounds.

• DakotaRealVector all_continuous_lower_bounds () const  
  returns a single array with all continuous lower bounds.

• DakotaRealVector all_continuous_upper_bounds () const  
  returns a single array with all continuous upper bounds.

• DakotaIntVector all_discrete_lower_bounds () const  
  returns a single array with all discrete lower bounds.

• DakotaIntVector all_discrete_upper_bounds () const  
  returns a single array with all discrete upper bounds.

• void write (ostream &s) const  
  write a variable constraints object to an ostream.

• void read (istream &s)  
  read a variable constraints object from an istream.

**Private Attributes**

• DakotaRealVector mergedDesignLowerBnds  
  a design lower bounds array merging continuous and discrete domains (integer values promoted to reals).

• DakotaRealVector mergedDesignUpperBnds  
  a design upper bounds array merging continuous and discrete domains (integer values promoted to reals).
6.62 MergedVarConstraints Class Reference

- DakotaRealVector uncertainDistLowerBnds
  
  the uncertain distribution lower bounds array (no discrete uncertain to merge).

- DakotaRealVector uncertainDistUpperBnds
  
  the uncertain distribution upper bounds array (no discrete uncertain to merge).

- DakotaRealVector mergedStateLowerBnds
  
  a state lower bounds array merging continuous and discrete domains (integer values promoted to reals).

- DakotaRealVector mergedStateUpperBnds
  
  a state upper bounds array merging continuous and discrete domains (integer values promoted to reals).

6.62.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which employs the merged data view.

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MergedVarConstraints derived class combines continuous and discrete domain types but separates design, uncertain, and state variable types. The result is merged design bounds arrays (mergedDesignLowerBnds, mergedDesignUpperBnds), uncertain distribution bounds arrays (uncertainDistLowerBnds, uncertainDistUpperBnds), and merged state bounds arrays (mergedStateLowerBnds, mergedStateUpperBnds). The branch and bound strategy uses this approach (see DakotaVariables::get_variables(problem_db) for variables type selection; variables type is passed to the DakotaVarConstraints constructor in DakotaModel).

6.62.2 Constructor & Destructor Documentation

6.62.2.1 MergedVarConstraints::MergedVarConstraints (const ProblemDescDB & problem_db)

constructor.

Extract fundamental lower and upper bounds and merge continuous and discrete domains to create mergedDesignLowerBnds, mergedDesignUpperBnds, mergedStateLowerBnds, and mergedStateUpperBnds using utilities from VariablesUtil (uncertain distribution bounds do not require any merging).

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

- MergedVarConstraints.H
- MergedVarConstraints.C
6.63 MergedVariables Class Reference

Derived class within the DakotaVariables hierarchy which employs the merged data view.

Inheritance diagram for MergedVariables::

```
  DakotaVariables
     |        |
     v        v
  MergedVariables
     ^        |
     |        V
  VariablesUtil
```

Public Methods

- **MergedVariables ()**
  
  *default constructor.*

- **MergedVariables (const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **~MergedVariables ()**
  
  *destructor.*

- **size_t tv () const**
  
  *Returns total number of vars.*

- **size_t cv () const**
  
  *Returns number of active continuous vars.*

- **size_t dv () const**
  
  *Returns number of active discrete vars.*

- **const DakotaRealVector & continuous_variables () const**
  
  *return the active continuous variables.*

- **void continuous_variables (const DakotaRealVector &c_vars)**
  
  *set the active continuous variables.*

- **const DakotaIntVector & discrete_variables () const**
  
  *return the active discrete variables.*

- **void discrete_variables (const DakotaIntVector &d_vars)**
  
  *set the active discrete variables.*

- **const DakotaStringArray & continuous_variable_labels () const**
  
  *return the active continuous variable labels.*
- void `continuous_variable_labels` (const DakotaStringArray &cv_labels)
  set the active continuous variable labels.

- const DakotaStringArray & `discrete_variable_labels` () const
  return the active discrete variable labels.

- void `discrete_variable_labels` (const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

- const DakotaRealVector & `inactive_continuous_variables` () const
  return the inactive continuous variables.

- void `inactive_continuous_variables` (const DakotaRealVector &i_c_vars)
  set the inactive continuous variables.

- const DakotaIntVector & `inactive_discrete_variables` () const
  return the inactive discrete variables.

- void `inactive_discrete_variables` (const DakotaIntVector &i_d_vars)
  set the inactive discrete variables.

- size_t `acv` () const
  returns total number of continuous vars.

- size_t `adv` () const
  returns total number of discrete vars.

- DakotaRealVector `all_continuous_variables` () const
  returns a single array with all continuous variables.

- DakotaIntVector `all_discrete_variables` () const
  returns a single array with all discrete variables.

- DakotaStringArray `all_continuous_variable_labels` () const
  returns a single array with all continuous variable labels.

- DakotaStringArray `all_discrete_variable_labels` () const
  returns a single array with all discrete variable labels.

- void `read` (istream &s)
  read a variables object from an istream.

- void `write` (ostream &s) const
  write a variables object to an ostream.

- void `read.annotated` (istream &s)
  read a variables object in annotated format from an istream.

- void `write.annotated` (ostream &s) const
write a variables object in annotated format to an ostream.

- void read (DakotaBiStream &s)
  read a variables object from the binary restart stream.

- void write (DakotaBoStream &s) const
  write a variables object to the binary restart stream.

- void read (UnPackBuffer &s)
  read a variables object from a packed MPI buffer.

- void write (PackBuffer &s) const
  write a variables object to a packed MPI buffer.

Private Methods

- void copy_rep (const DakotaVariables *vars_rep)
  Used by copy() to copy the contents of a letter class.

Private Attributes

- DakotaRealVector mergedDesignVars
  a design variables array merging continuous and discrete domains (integer values promoted to reals).

- DakotaRealVector uncertainVars
  the uncertain variables array (no discrete uncertain to merge).

- DakotaRealVector mergedStateVars
  a state variables array merging continuous and discrete domains (integer values promoted to reals).

- DakotaStringArray mergedDesignLabels
  a label array combining continuous design and discrete design labels.

- DakotaStringArray uncertainLabels
  the uncertain variables label array (no discrete uncertain to combine).

- DakotaStringArray mergedStateLabels
  a label array combining continuous state and discrete state labels.

Friends

- bool operator==(const MergedVariables &vars1, const MergedVariables &vars2)
  equality operator.
6.63 MergedVariables Class Reference

6.63.1 Detailed Description

Derived class within the DakotaVariables hierarchy which employs the merged data view.

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MergedVariables derived class combines continuous and discrete domain types but separates design, uncertain, and state variable types. The result is a single continuous array of design variables (mergedDesignVars), a single continuous array of uncertain variables (uncertainVars), and a single continuous array of state variables (mergedStateVars). The branch and bound strategy uses this approach (see DakotaVariables::getVariables(problem_db)).

6.63.2 Constructor & Destructor Documentation

6.63.2.1 MergedVariables::MergedVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels and merge continuous and discrete domains to create mergedDesignVars, mergedStateVars, mergedDesignLabels, and mergedStateLabels using utilities from VariablesUtil (uncertain variables and labels do not require any merging).

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

- MergedVariables.H
- MergedVariables.C
6.64 MultilevelOptStrategy Class Reference

Strategy for hybrid optimization using multiple optimizers on multiple models of varying fidelity.

Inheritance diagram for MultilevelOptStrategy::

```
DakotaStrategy

 MultilevelOptStrategy
```

Public Methods

- **MultilevelOptStrategy** (ProblemDescDB &problem_db)
  
  constructor.

- **~MultilevelOptStrategy** ()
  
  destructor.

- **void run_strategy()**
  
  Performs the hybrid optimization strategy by executing multiple iterators on different models of varying fidelity.

- **const DakotaVariables & strategy_variable_results()** const
  
  return the final solution from selectedIterators (variables).

- **const DakotaResponse & strategy_response_results()** const
  
  return the final solution from selectedIterators (response).

Private Methods

- **void run_coupled()**
  
  run a tightly coupled hybrid.

- **void run_uncoupled()**
  
  run an uncoupled hybrid.

- **void run_uncoupled_adaptive()**
  
  run an uncoupled adaptive hybrid.
Private Attributes

- **DakotaString multiLevelType**
  
  coupled, uncoupled, or uncoupled adaptive.

- **DakotaStringList methodList**
  
  the list of method identifiers.

- **int numIterators**
  
  number of methods in methodList.

- **Real localSearchProb**
  
  the probability of running a local search refinement within phases of the global optimization for coupled hybrids.

- **Real progressMetric**
  
  the amount of progress made in a single iterator++ cycle within an uncoupled adaptive hybrid.

- **Real progressThreshold**
  
  when the progress metric falls below this threshold, the uncoupled adaptive hybrid switches to the next method.

- **DakotaArray DakotaIterator > selectedIterators**
  
  the set of iterators, one for each entry in methodList.

- **DakotaArray DakotaModel > userDefinedModels**
  
  the set of models, one for each iterator.

6.64.1 Detailed Description

Strategy for hybrid optimization using multiple optimizers on multiple models of varying fidelity.

This strategy has three approaches to hybrid optimization: (1) the uncoupled hybrid runs one method to completion, passes its best results as the starting point for a subsequent method, and continues this succession until all methods have been executed; (2) the uncoupled adaptive hybrid is similar to the uncoupled hybrid, except that the stopping rules for the optimizers are controlled adaptively by the strategy instead of internally by each optimizer; and (3) the coupled hybrid uses multiple methods in close coordination, generally using a local search optimizer repeatedly within a global optimizer (the local search optimizer refines candidate optima which are fed back to the global optimizer). The uncoupled strategies only pass information forward, whereas the coupled strategy allows both feedback and feedback. Note that while the strategy is targeted at optimizers, any iterator may be used so long as it defines the notion of a final solution which can be passed as the starting point for subsequent iterators.

6.64.2 Member Function Documentation
6.64.2.1  void MultilevelOptStrategy::run_coupled () [private]

run a tightly coupled hybrid.
In the coupled case, use is made of external hybridization capabilities, such as those available in the global/local hybrids from SGOPT. This function is responsible only for publishing the local optimizer selection to the global optimizer and then invoking the global optimizer; the logic of method switching is handled entirely within the global optimizer. Status: incomplete.

6.64.2.2  void MultilevelOptStrategy::run_uncoupled () [private]

run an uncoupled hybrid.
In the uncoupled nonadaptive case, there is no interference with the iterators. Each runs until its own convergence criteria is satisfied (using iterator.run_iterator()). Status: fully operational.

6.64.2.3  void MultilevelOptStrategy::run_uncoupled_adaptive () [private]

run an uncoupled adaptive hybrid.
In the uncoupled adaptive case, there is interference with the iterators through the use of the ++ overloaded operator. iterator++ runs the iterator for one cycle, after which a progress metric is computed. This progress metric is used to dictate method switching instead of each iterator’s internal convergence criteria. Status: incomplete.

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

- MultilevelOptStrategy.H
- MultilevelOptStrategy.C
6.65 **NestedModel Class Reference**

Derived model class which performs a complete sub-iterator execution within every evaluation of the model.

Inheritance diagram for NestedModel:

```
DakotaModel

NestedModel
```

**Public Methods**

- **NestedModel (ProblemDescDB &problem_db)**
  
  constructor.

- **~NestedModel ()**
  
  destructor.

**Protected Methods**

- void **derived_compute_response** (const DakotaIntArray &asv)
  
  portion of **compute_response()** specific to NestedModel.

- void **derived_asynch_compute_response** (const DakotaIntArray &asv)
  
  portion of **asynch_compute_response()** specific to NestedModel.

- const DakotaResponseArray & **derived_synchronize** ()
  
  portion of **synchronize()** specific to NestedModel.

- const DakotaResponseList & **derived_synchronize_nowait** ()
  
  portion of **synchronize_nowait()** specific to NestedModel.

- const DakotaIntList & **synchronize_nowait_completions** ()
  
  Return completion id’s matching response list from **synchronize_nowait**.

- **DakotaModel & subordinate_model** ()
  
  return a reference to the subModel.

- bool **derived_master_overload** () const
  
  flag which prevents overloading the master with a multiprocessor evaluation (forwarded to subModel so that UQ portion of OUU can execute in parallel).
void derived_init_communicators (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
    portion of init_communicators() specific to NestedModel.

void free_communicators ()
    deallocate communicator partitions for the NestedModel (forwarded to subModel so that UQ portion of OUU can execute in parallel).

void serve ()
    Service job requests received from the master. Completes when a termination message is received from stop_servers(). (forwarded to subModel so that UQ portion of OUU can execute in parallel).

void stop_servers ()
    Executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (forwarded to subModel so that UQ portion of OUU can execute in parallel).

int total_eval_counter () const
    Return the total evaluation count for the NestedModel; forwarded to optionalInterface if present (placeholder for now).

int new_eval_counter () const
    Return the new evaluation count for the NestedModel; forwarded to optionalInterface if present (placeholder for now).

**Private Methods**

void response_mapping (const DakotaResponse &interface_response, const DakotaResponse &sub_iterator_response, DakotaResponse &mapped_response)
    combine the response from the optional interface evaluation with the response from the sub-iteration using the objCoeffs/constrCoeffs mappings to create the total response for the model.

void asv_mapping (const DakotaIntArray &mapped_asv, DakotaIntArray &interface_asv)
    define the evaluation requirements for the optional interface (interface_asv) from the total model evaluation requirements (mapped_asv).

**Private Attributes**

- DakotaIterator subIterator
    the sub-iterator that is executed on every evaluation of this model.

- DakotaModel subModel
    the sub-model used in sub-iterator evaluations.

- size_t numSubIteratorIneqConstr
    number of top-level inequality constraints mapped from the sub-iteration results.

- size_t numSubIteratorEqConstr
    number of top-level equality constraints mapped from the sub-iteration results.
6.65 NestedModel Class Reference

- **DakotaInterface optionalInterface**
  - the optional interface contributes nonnested response data to the total model response.

- **DakotaString interfacePointer**
  - the optional interface pointer from the nested model specification.

- **DakotaResponse interfaceResponse**
  - the response object resulting from optional interface evaluations.

- **size_t numInterfObjFns**
  - number of objective functions resulting from optional interface evaluations.

- **size_t numInterfIneqConstr**
  - number of inequality constraints resulting from optional interface evaluations.

- **size_t numInterfEqConstr**
  - number of equality constraints resulting from the optional interface evaluations.

- **DakotaRealMatrix objCoeffs**
  - "primary" response mapping matrix applied to the sub-iterator response functions. For OUU, the matrix is applied to UQ statistics to create contributions to the top-level objective function(s).

- **DakotaRealMatrix constrCoeffs**
  - "secondary" response mapping matrix applied to the sub-iterator response functions. For OUU, the matrix is applied to UQ statistics to create contributions to the top-level inequality and equality constraints.

- **DakotaResponseArray responseArray**
  - dummy response array for derived_synchronize() prior to derived_asynch_compute_response() support.

- **DakotaResponseList responseList**
  - dummy response list for derived_synchronize_nowait() prior to derived_asynch_compute_response() support.

- **DakotaIntList completionList**
  - dummy completion list for synchronize_nowait_completions() prior to derived_asynch_compute_response() support.

### 6.65.1 Detailed Description

Derived model class which performs a complete sub-iterator execution within every evaluation of the model.

The NestedModel class nests a sub-iterator execution within every model evaluation. This capability is most commonly used for optimization under uncertainty, in which a nondeterministic iterator is executed on every optimization function evaluation. The NestedModel also contains an optional interface, for portions of the model evaluation which are independent from the sub-iterator, and a set of mappings for combining sub-iterator and optional interface data into a top level response for the model.
6.65.2 Member Function Documentation

6.65.2.1 void NestedModel::derived_compute_response (const DakotaIntArray & asv) [protected, virtual]

portion of compute_response() specific to NestedModel.

Update subModel’s inactive variables with active variables from currentVariables, compute the optional interface and sub-iterator responses, and map these to the total model response.

Reimplemented from DakotaModel.

6.65.2.2 void NestedModel::derived_asynch_compute_response (const DakotaIntArray & asv) [protected, virtual]

portion of asynch_compute_response() specific to NestedModel.

Not currently supported by NestedModels (need to add concurrent iterator support). As a result, derived_synchronize(), derived_synchronize_nowait(), and synchronize_nowait_completions() are inactive as well.

Reimplemented from DakotaModel.

6.65.2.3 const DakotaResponseArray & NestedModel::derived_synchronize () [protected, virtual]

portion of synchronize() specific to NestedModel.

Asynchronous response computations are not currently supported by NestedModels. Return a dummy responseArray to satisfy the compiler.

Reimplemented from DakotaModel.

6.65.2.4 const DakotaResponseList & NestedModel::derived_synchronize_nowait () [protected, virtual]

portion of synchronize_nowait() specific to NestedModel.

Asynchronous response computations are not currently supported by NestedModels. Return a dummy responseList to satisfy the compiler.

Reimplemented from DakotaModel.

6.65.2.5 const DakotaIntList & NestedModel::synchronize_nowait_completions () [inline, protected, virtual]

Return completion id’s matching response list from synchronize_nowait.

Asynchronous response computations are not currently supported by NestedModels. Return a dummy completionList to satisfy the compiler.

Reimplemented from DakotaModel.
6.65 NestedModel Class Reference

6.65.2.6 void NestedModel::derived_init_communicators (const DakotaIntArray &
message_lengths, const int & max_iter_concurrency) [inline, protected,
virtual]

portion of init_communicators() specific to NestedModel.

Asynchronous flags need to be initialized for the subModel. In addition, max_iter_concurrency is the
outer level iterator concurrency, not the subIterator concurrency that subModel will see, and recomputing
the message_lengths on the subModel is probably not a bad idea either. Therefore, recompute everything
on subModel using init_communicators().

Reimplemented from DakotaModel.

6.65.2.7 void NestedModel::response_mapping (const DakotaResponse & interface_response,
const DakotaResponse & sub_iterator_response, DakotaResponse & mapped_response)
[private]

combine the response from the optional interface evaluation with the response from the sub-iteration using
the objCoeffs/constrCoeffs mappings to create the total response for the model.

In the OUU case,

optionalInterface functions = \{f\}, \{g\} (deterministic objectives & constraints)
subIterator functions = \{S\} (UQ response statistics)

Problem formulation for mapped functions:
minimize \{f\} + [W]\{S\}
subject to \{g_\text{l}\} <= \{g\} <= \{g_\text{u}\}
\{a_\text{l}\} <= [A]\{S\} <= \{a_\text{u}\}
\{g\} = \{g_\text{t}\}
\{A\}\{S\} = \{a_\text{t}\}

where [W] is the primary_mapping_matrix user input (objCoeffs class attribute), [A] is the secondary_mapping_matrix user input (constrCoeffs class attribute), \{\{g_\text{l}\},\{a_\text{l}\}\} are the top level inequality constraint lower bounds, \{\{g_\text{u}\},\{a_\text{u}\}\} are the top level inequality constraint upper bounds, and \{\{g_\text{t}\},\{a_\text{t}\}\} are the top level equality constraint targets.

NOTE: optionalInterface/subIterator objectives overlap but optionalInterface/subIterator constraints do not.
The [W] matrix can be specified so as to allow

- some purely deterministic objective functions and some combined: [W] filled and [W].num_rows() < \{f\}.length() [combined first] or [W].num_rows() == \{f\}.length() and [W] contains rows of zeros [combined last]
- some combined and some purely stochastic objective functions: [W] filled and [W].num_rows() > \{f\}.length()
- separate deterministic and stochastic objective functions: [W].num_rows() > \{f\}.length() and [W] contains \{f\}.length() rows of zeros.

If the need arises, could change constraint definition to allow overlap as well: \{g_\text{l}\} <= \{g\} + [A]\{S\} <= \{g_\text{u}\} with [A] usage the same as for [W] above.

In the UOO case, things are simpler, just compute statistics of each optimization response function: [W] = [I], \{f\}/\{g\}/[A] are empty.

6.65.3 Member Data Documentation
6.65.3.1 DakotaModel NestedModel::subModel [private]

the sub-model used in sub-iterator evaluations.

There are no restrictions on subModel, so arbitrary nestings are possible. This is commonly used to support surrogate-based optimization under uncertainty by having NestedModels contain LayeredModels and vice versa.

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

- NestedModel.H
- NestedModel.C
6.66 NLSSOLLeastSq Class Reference

Wrapper class for the NLSSOL nonlinear least squares library.

Inheritance diagram for NLSSOLLeastSq:

```
NLSSOLLeastSq
   |___DakotaLeastSq
       |___SOLBase
          |___NLSSOLLeastSq
```

Public Methods

- **NLSSOLLeastSq (DakotaModel &model)**
  
  *standard constructor.*

- **~NLSSOLLeastSq ()**
  
  *destructor.*

- **void minimize_residuals ()**
  
  *Used within the least squares branch for minimizing the sum of squares residuals. Redefines the run_iterator virtual function for the least squares branch.*

Static Private Methods

- **void least_sq_eval (int &mode, int &m, int &n, int &nrowfj, Real *x, Real *f, Real *gradf, int &nstate)**
  
  *Evaluator for NLSSOL: computes the values and first derivatives of the least squares terms (passed by function pointer to NLSSOL).*

6.66.1 Detailed Description

Wrapper class for the NLSSOL nonlinear least squares library.

The NLSSOLLeastSq class provides a wrapper for NLSSOL, a Fortran 77 sequential quadratic programming library from Stanford University marketed by Stanford Business Associates. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or static as well. To isolate the effect of these static requirements from the rest of the iterator hierarchy, static copies are made of many non-static attributes inherited from above.
The user input mappings are as follows: \texttt{max\_function\_evaluations} is implemented directly in \texttt{NLSSOLLeastSq}'s evaluator functions since there is no \texttt{NLSSOL} parameter equivalent, and \texttt{max\_iterations}, \texttt{convergence\_tolerance}, \texttt{output\_verbosity}, \texttt{verify\_level}, \texttt{function\_precision}, and \texttt{linesearch\_tolerance} are mapped into \texttt{NLSSOL}'s "Major Iteration Limit", "Optimality Tolerance", "Major Print Level" (\texttt{verbose}: Major Print Level $= 20$; \texttt{quiet}: Major Print Level $= 10$), "Verify Level", "Function Precision", and "Linesearch Tolerance" parameters, respectively, using \texttt{NLSSOL}'s \texttt{npoptn}() subroutine (as wrapped by \texttt{npoptn2()} from the \texttt{npoptn\_wrapper.f} file). Refer to [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] for information on \texttt{NLSSOL}'s optional input parameters and the \texttt{npoptn()} subroutine.

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

- \texttt{NLSSOLLeastSq.H}
- \texttt{NLSSOLLeastSq.C}
6.67 NoDBBaseConstructor Struct Reference

Dummy struct for overloading constructors used in on-the-fly instantiations.

Public Methods

- **NoDBBaseConstructor** (int=0)

  *C++ structs can have constructors.*

6.67.1 Detailed Description

Dummy struct for overloading constructors used in on-the-fly instantiations.

NoDBBaseConstructor is used to overload the constructor used for on-the-fly iterator instantiations in which ProblemDescDB queries cannot be used. Putting this struct here (rather than in a header of a class that uses it) avoids problems with circular dependencies.

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

- ProblemDescDB.H
6.68 NonDAdvMeanValue Class Reference

Class for the analytical reliability methods within DAKOTA/UQ.

Inheritance diagram for NonDAdvMeanValue::

```
NonDAdvMeanValue
   `-> DakotaIterator
       `-> DakotaNonD
           `-> NonDAdvMeanValue
```

Public Methods

- **NonDAdvMeanValue (DakotaModel &model)**
  constructor.

- **~NonDAdvMeanValue ()**
  destructor.

- **void quantify_uncertainty ()**
  performs an uncertainty propagation using analytical reliability methods which solve constrained optimization problems to obtain approximations of the cumulative distribution function of response.

- **void print_iterator_results (ostream &s) const**
  print the approximate mean, standard deviation, and importance factors when using the mean value method (MV) or the CDF information when using other reliability methods (AMV, AMV+, FORM).

Private Methods

- **void mean_value ()**
  convenience function for encapsulating the simple Mean Value computation of approximate statistics and importance factors.

- **void iterated_mean_value ()**
  convenience function for encapsulating the iterated reliability methods (AMV, AMV+, FORM, SORM).

- **void transUToX (const DakotaRealVector &uncorr_normal_vars, DakotaRealVector &random_vars)**
  Transformation Routine from u-space of random variables to x-space of random variables for DakotaRealVector data types.

- **void transXToU (const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseVector &uncorr_normal_vars)**
Transformation Routine from x-space of random variables to z-space of random variables for Petra data types.

- **void** `transXToZ` (const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseVector &correlated_normal_vars)

  Transformation Routine from x-space of random variables to z-space of random variables for Petra data types.

- **void** `transUToZ` (const Epetra_SerialDenseVector &uncorr_normal_vars, Epetra_SerialDenseVector &correlated_normal_vars)

  Transformation Routine from u-space of random variables to z-space of random variables for Petra data types.

- **void** `transZToU` (Epetra_SerialDenseVector &correlated_normal_vars, Epetra_SerialDenseVector &uncorr_normal_vars)

  Transformation Routine from z-space of random variables to u-space of random variables for Petra data types.

- **void** `jacXToZ` (const Epetra_SerialDenseVector &random_vars, const Epetra_SerialDenseVector &correlated_normal_vars, Epetra_SerialDenseMatrix &jacobianXZ)

  Jacobian of mapping from x to z random variable space.

- **void** `jacZToX` (const Epetra_SerialDenseVector &correlated_normal_vars, const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseMatrix &jacobianZX)

  Jacobian of mapping from z to x random variable space.

- **void** `jacXToU` (const Epetra_SerialDenseVector &random_vars, const Epetra_SerialDenseVector &uncorr_normal_vars, Epetra_SerialDenseMatrix &jacobianXU)

  Jacobian of mapping from x to u random variable space.

- **void** `jacUToX` (const Epetra_SerialDenseVector &uncorr_normal_vars, const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseMatrix &jacobianUX)

  Jacobian of mapping from u to x random variable space.

- **void** `transNataf` (Epetra_SerialSymDenseMatrix &mod_corr_matrix)

  This procedure modifies the correlation matrix input by the user to be used in the Nataf distribution model.

- **void** `erfInverse` (const double &p, double &z)

  Inverse of error function used to invert cdf of normal random variables.

### Static Private Methods

- **void** `lin_approx_objective_eval` (int &mode, int &n, Real *u, Real &f, Real *gradf, int &)

  static function used by NPSOL as the objective function in the constrained optimization problems solved in the analytical reliability methods.

- **void** `lin_approx_constraint_eval` (int &mode, int &ncnln, int &n, int &nrowj, int *needc, Real *u, Real *cjac, Real *c, int &nstate)

  static function used by NPSOL as the constraint function in the constrained optimization problems solved in the analytical reliability methods.
void transUToX (const Epetra_SerialDenseVector &uncorr_normal_vars, Epetra_SerialDenseVector &random_vars)

Transformation Routine from u-space of random variables to x-space of random variables for Petra data types.

**Private Attributes**

- Epetra_SerialSymDenseMatrix petraCorrMatrix
  petra copy of uncertainCorrelations.

- DakotaRealVector medianFnVals
  vector of median values of functions used to determine which side of probability equal 0.5 the response level is.

- DakotaRealVector probLevels
  computed probability values.

- DakotaString reliabilityMethod
  reliability method identifier specified by user specifies amvFlag.

- DakotaRealMatrix impFactor
  importance factors predicted by MV.

- int numRelEqConstr
  number of equality constraints applied during the MPP search.

- size_t numLevels
  number of response/probaility levels to loop over.

**Static Private Attributes**

- Epetra_SerialDenseVector staticFnVals
  static copy of DakotaResponseRep::functionValues.

- Epetra_SerialDenseMatrix staticFnGrads
  static copy of DakotaResponseRep::functionGradients.

- Epetra_SerialDenseMatrix staticGlobalGradsX
  Gradient of Response function in x-space for each response level.

- Epetra_SerialDenseMatrix staticGlobalGradsU
  Gradient of Response function in u-space for each response level.

- Epetra_SerialDenseMatrix cholCorrMatrix
  cholesky factor of petraCorrMatrix.
6.68 NonDAdvMeanValue Class Reference

- Epetra_SerialDenseMatrix mostProbPointX
  Location of MPP in x space.

- Epetra_SerialDenseMatrix mostProbPointU
  Location of MPP in u space.

- Epetra_SerialDenseVector ranVarMeans
  Mean Vector of all uncertain random variables.

- Epetra_SerialDenseVector ranVarSigmas
  Standard Deviation Vector of all uncertain random variables.

- Epetra_SerialDenseVector petraRespLevels
  user specified targets for response levels.

- Epetra_SerialDenseVector petraProbLevels
  user specified targets for probability levels.

- Epetra_SerialDenseVector correctedRespLevel
  output response levels calculated.

- int respLevelCount
  counter for which response level is being analyzed.

- short amvFlag
  flag to represent which reliability method is being used.

- size_t staticNumUncVars
  static copy of numUncertainVars.

- size_t staticNumFuncs
  static copy of numFunctions.

- DakotaIntVector ranVarType
  vector of indices indicating which type of uncertain variable.

### 6.68.1 Detailed Description

Class for the analytical reliability methods within DAKOTA/UQ.

The NonDAdvMeanValue class implements the following analytic reliability methods: advanced mean value method (AMV), iterated advanced mean value method (AMV+), first order reliability method (FORM), and second order reliability method (SORM). Each of these employ an optimizer (currently NPSOL) to perform a search for the most probable point (MPP).

### 6.68.2 Member Function Documentation
6.68.2.1 `void NonDAdvMeanValue::lin_approx_objective_eval (int & mode, int & n, Real * u, Real & f, Real * gradf, int &)` [static, private]

static function used by NPSOL as the objective function in the constrained optimization problems solved in the analytical reliability methods.

Need to be static so that they can be passed in function pointers without having to restrict the recipient to functions from the NonDAdvMeanValue class (see Stroustrup, p.166 - pointers to member functions must use class scope operators which would restrict the generality of the NPSOLOptimizer "user functions" interface).

6.68.2.2 `void NonDAdvMeanValue::lin_approx_constraint_eval (int & mode, int & ncnln, int & n, int & nrowj, int * needc, Real * u, Real * c, Real * jac, int & nstate)` [static, private]

static function used by NPSOL as the constraint function in the constrained optimization problems solved in the analytical reliability methods.

Need to be static so that they can be passed in function pointers without having to restrict the recipient to functions from the NonDAdvMeanValue class (see Stroustrup, p.166 - pointers to member functions must use class scope operators which would restrict the generality of the NPSOLOptimizer "user functions" interface).

6.68.2.3 `void NonDAdvMeanValue::transUToX (const Epetra_SerialDenseVector & uncorr_normal_vars, Epetra_SerialDenseVector & random_vars)` [static, private]

Transformation Routine from u-space of random variables to x-space of random variables for Petra data types.

This procedure performs the transformation from u to x space

uncorr_normal_vars is the vector of random variables in standard normal space (u-space).

random_vars is the vector of the random variables in the user-defined x-space

6.68.2.4 `void NonDAdvMeanValue::transXToU (const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseVector & uncorr_normal_vars)` [private]

Transformation Routine from x-space of random variables to u-space of random variables for Petra data types.

This procedure performs the transformation from x to u space

uncorr_normal_vars is the vector of random variables in standard normal space (u-space).

random_vars is the vector of the random variables in the user-defined x-space.

6.68.2.5 `void NonDAdvMeanValue::transXToZ (const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseVector & correlated_normal_vars)` [private]

Transformation Routine from x-space of random variables to z-space of random variables for Petra data types.

This procedure performs the transformation from x to z space:

correlated_normal_vars is the vector of random variables in normal space with proper correlations(z-space).
random_vars is the vector of the random variables in the user-defined x-space.

6.68.2.6 void NonDAdvMeanValue::transUToZ (const Epetra_SerialDenseVector & uncorr_normal_vars, Epetra_SerialDenseVector & correlated_normal_vars) [private]

Transformation Routine from u-space of random variables to z-space of random variables for Petra data types.
This procedure computes the transformation from u to z space.
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).

6.68.2.7 void NonDAdvMeanValue::transZToU (Epetra_SerialDenseVector & correlated_normal_vars, Epetra_SerialDenseVector & uncorr_normal_vars) [private]

Transformation Routine from z-space of random variables to u-space of random variables for Petra data types.
This procedure computes the transformation from z to u space.
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).

6.68.2.8 void NonDAdvMeanValue::jacXToZ (const Epetra_SerialDenseVector & random_vars, const Epetra_SerialDenseVector & correlated_normal_vars, Epetra_SerialDenseMatrix & jacobianXZ) [private]

Jacobian of mapping from x to z random variable space.
This procedure computes the jacobian of the transformation from x to z space.
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).
random_vars is the vector of the random variables in the user-defined x-space.

6.68.2.9 void NonDAdvMeanValue::jacZToX (const Epetra_SerialDenseVector & correlated_normal_vars, const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseMatrix & jacobianZX) [private]

Jacobian of mapping from z to x random variable space.
This procedure computes the jacobian of the transformation from z to x space.
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).
random_vars is the vector of the random variables in the user-defined x-space.

6.68.2.10 void NonDAdvMeanValue::jacXToU (const Epetra_SerialDenseVector & random_vars, const Epetra_SerialDenseVector & uncorr_normal_vars, Epetra_SerialDenseMatrix & jacobianXU) [private]

Jacobian of mapping from x to u random variable space.
This procedure computes the jacobian of the transformation from x to u space.

uncorr_normal_vars is the vector of random variables in standard normal space (u-space).
random_vars is the vector of the random variables in the user-defined x-space.

6.68.2.11 void NonDAdvMeanValue::jacUToX (const Epetra_SerialDenseVector & uncorr_normal_vars, const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseMatrix & jacobianUX) [private]

Jacobian of mapping from u to x random variable space.

This procedure computes the jacobian of the transformation from u to x space.
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).
random_vars is the vector of the random variables in the user-defined x-space.

6.68.2.12 void NonDAdvMeanValue::transNataf (Epetra_SerialSymDenseMatrix & mod_corr_matrix) [private]

This procedure modifies the correlation matrix input by the user to be used in the Nataf distribution model.

R: the correlation coefficient matrix of the random variables
mod_corr_matrix: modified correlation matrix

Note: The modification is exact for log-log, normal-log, normal-normal, normal-uniform transformations (numerical precision). The uniform-uniform and uniform-log case are approximations obtained in the above reference.

6.68.3 Member Data Documentation

6.68.3.1 Epetra_SerialDenseVector NonDAdvMeanValue::correctedRespLevel [static, private]

output response levels calculated.
identical to petraRespLevels for AMV+/FORM, but will differ for AMV

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

- NonDAdvMeanValue.H
- NonDAdvMeanValue.C
6.69 NonDLHSSampling Class Reference

Performs LHS and Monte Carlo sampling for uncertainty quantification.

Inheritance diagram for NonDLHSSampling::

```
DakotaIterator
  |
  DakoNonD
  |
  NonDSampling
  |
NonDLHSSampling
```

Public Methods

- **NonDLHSSampling** (DakotaModel &model)
  
  constructor.

- **NonDLHSSampling** (DakotaModel &model, int samples, int seed, int num_vars, const DakotaRealVector &lower_bnds, const DakotaRealVector &upper_bnds)

- ~**NonDLHSSampling**()
  
  destructor.

- void **quantify_uncertainty** ()
  
  performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results.

- void **print_iterator_results** (ostream &s) const
  
  print the final statistics.

Private Attributes

- bool **allVarsFlag**
  
  flags DACE mode using all variables.

6.69.1 Detailed Description

Performs LHS and Monte Carlo sampling for uncertainty quantification.

The Latin Hypercube Sampling (LHS) package from Sandia Albuquerque’s Risk and Reliability organization provides comprehensive capabilities for Monte Carlo and Latin Hypercube sampling within a broad
array of user-specified probabilistic parameter distributions. It enforces user-specified rank correlations through use of a mixing routine. The NonDLHSSampling class provides a C++ wrapper for the LHS library and is used for performing forward propagations of parameter uncertainties into response statistics.

6.69.2 Constructor & Destructor Documentation

6.69.2.1 NonDLHSSampling::NonDLHSSampling (DakotaModel & model)

This constructor is called for a standard letter-envelope iterator instantiation. In this case, setDBlistNodes has been called and probDescDB can be queried for settings from the method specification.

6.69.2.2 NonDLHSSampling::NonDLHSSampling (DakotaModel & model, int samples, int seed, int num-vars, const DakotaRealVector & lower_bnds, const DakotaRealVector & upper_bnds)

This alternate constructor is used by ConcurrentStrategy for generation of uniform, uncorrelated sample sets. It is not a letter-envelope instantiation and a setDBlistNodes has not been performed. It is called with all needed data passed through the constructor and is designed to allow more flexibility in variables set definition (i.e., relax connection to a variables specification and allow sampling over parameter sets such as multiobjective weights). Data attributes taken from the model in the NoDBBaseConstructor constructors for DakotaNonD and DakotaIterator are not used, and other data attributes are not initialized and should not be avoided.

6.69.3 Member Function Documentation

6.69.3.1 void NonDLHSSampling::quantify_uncertainty () [virtual]

performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results. Loop over the set of samples and compute responses. Compute statistics on the set of responses if statsFlag is set.

Implements DakotaNonD.

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

- NonDLHSSampling.H
- NonDLHSSampling.C
**6.70 NonDOptStrategy Class Reference**

Strategy for optimization under uncertainty (robust and reliability-based design).

Inheritance diagram for NonDOptStrategy:

```
DakotaStrategy
   NonDOptStrategy
```

**Public Methods**

- **NonDOptStrategy (ProblemDescDB &problem_db)**
  
  *constructor.*

- **~NonDOptStrategy ()**
  
  *destructor.*

- **void run_strategy ()**
  
  *Perform the strategy by executing optIterator (an optimizer) on designModel (a layered or nested model containing a nondeterministic iterator at a lower level).*

- **const DakotaVariables & strategy_variable_results () const**
  
  *return the final solution from optIterator (variables).*

- **const DakotaResponse & strategy_response_results () const**
  
  *return the final solution from optIterator (response).*

**Private Attributes**

- **DakotaModel designModel**
  
  *the nested or layered model interfaced with optIterator.*

- **DakotaIterator optIterator**
  
  *the top level optimizer.*

**6.70.1 Detailed Description**

Strategy for optimization under uncertainty (robust and reliability-based design).

This strategy uses a **NestedModel** to nest an uncertainty quantification iterator within an optimization iterator in order to perform optimization using nondeterministic data. For OUU based on surrogates, LayeredModels are also employed, and the general recursion facilities supported by nested and
layered models allow a broad array of OUU formulations. This class is very simple and is essentially identical to SingleMethodStrategy since all of the nested iteration mappings are contained within NestedModel::response_mapping().

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

- NonDOptStrategy.H
- NonDOptStrategy.C
6.71 NonDPCESampling Class Reference

Stochastic finite element approach to uncertainty quantification using polynomial chaos expansions.

Inheritance diagram for NonDPCESampling::

```
DakotaIterator
   ▼
  |    ▼
  |       ▼
  |        ▼
  |         ▼
DakotaNonD
    ▼
NonDSampling
      ▼
NonDPCESampling
```

Public Methods

- **NonDPCESampling (DakotaModel &model)**
  constructor.

- **~NonDPCESampling ()**
  destructor.

- **void quantify_uncertainty ()**
  perform a forward uncertainty propagation using SFEM/PCE methods.

- **void print_iterator_results (ostream &s) const**
  print the final statistics and PCE coefficient array.

Private Attributes

- **DakotaRealVectorArray coeffArray**
  Array containing Polynomial Chaos coefficients, one real vector per response function.

- **int highestOrder**
  Highest order of Hermite Polynomials in Expansion.

- **int numChaos**
  Number of terms in Polynomial Chaos Expansion.
6.71.1 Detailed Description

Stochastic finite element approach to uncertainty quantification using polynomial chaos expansions.

The NonDPCE class uses a polynomial chaos expansion (PCE) approach to approximate the effect of parameter uncertainties on response functions of interest. It utilizes the HermiteSurf and HermiteChaos classes to perform the PCE.

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

- NonDPCESampling.H
- NonDPCESampling.C
6.72 NonDSampling Class Reference

Base class for common code between NonDLHSSampling and NonDPCESampling.

Inheritance diagram for NonDSampling:

```
DakotaIterator
   
DakotaNonD
   
NonDSampling
   
NonDLHSSampling  NonDPCESampling
```

Public Methods

- **NonDSampling** (DakotaModel &model)
  
  Constructor.

- **NonDSampling** (NoDBaseConstructor, DakotaModel &model, int samples, int seed, int num_vars, const DakotaRealVector &lower_bnds, const DakotaRealVector &upper_bnds)

- **~NonDSampling** ()
  
  Destructor.

- void **sampling_reset** (int min_samples, bool all_data_flag, bool stats_flag)
  
  Resets number of samples and sampling flags.

- const DakotaString & **sampling_scheme** () const
  
  Return sample type: "lhs" or "random".

Protected Methods

- void **run_lhs** ()
  
  Generates the desired set of parameter samples from within user-specified probabilistic distributions. Supports both old and new LHS libraries. Used by NonDLHSSampling and NonDPCESampling.

- void **compute_statistics** (const DakotaRealVectorArray &samples)
  
  Computes mean, standard deviation, and probability of failure for the samples input.

- void **print_statistics** (ostream &s) const
  
  Prints the mean, standard deviation, and probability of failure statistics computed in compute_statistics().

---

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (c) 1997-2002
Protected Attributes

- int numObservations
  the number of samples to evaluate.

- DakotaString sampleType
  the sample type: "lhs" or "random".

- bool statsFlag
  flags computation/output of statistics.

- bool allDataFlag
  flags update of allVariables/allResponses.

- size_t numActiveVars
  total number of variables published to LHS.

- size_t numDesignVars
  number of design variables (treated as uniform distribution within design variable bounds for DACE usage of NonDSampling).

- size_t numStateVars
  number of state variables (treated as uniform distribution within state variable bounds for DACE usage of NonDSampling).

Private Methods

- void check_error (const int &err_code, const char *err_source) const
  checks the return codes from LHS routines and aborts if an error is returned.

Private Attributes

- const int originalSeed
  the user seed specification (default is 0).

- int randomSeed
  the current random number seed.

- size_t numLHSRuns
  counter for number of executions of runLhs() for this object.

- bool varyPattern
  flag for generating a sequence of seed values within multiple runLhs() calls so that the runLhs() executions (e.g., for surrogate-based optimization) are repeatable but not correlated.

- bool strategyFlag
  flag indicating a strategy other than "single_method". Used to deactivate some output when multiple sample sets may be generated.
6.72 NonDSampling Class Reference

6.72.1 Detailed Description

Base class for common code between NonDLHSSampling and NonDPCESampling.

This base class provides common code for sampling methods which employ the Latin Hypercube Sampling (LHS) package from Sandia Albuquerque’s Risk and Reliability organization. NonDSampling manages two LHS versions within a ifdef construct in run_lhs(): (1) the 1998 Fortran 90 LHS version as documented in SAND98-0210, which was converted to a UNIX link library in 2001, (2) the 1970’s vintage LHS that had been f2c’d and converted to (incomplete) classes.

6.72.2 Constructor & Destructor Documentation

6.72.2.1 NonDSampling::NonDSampling (DakotaModel & model)

constructor.

This constructor is called for a standard letter-envelope iterator instantiation. In this case, set_db_list_nodes has been called and probDescDB can be queried for settings from the method specification.

6.72.2.2 NonDSampling::NonDSampling (NoDBBaseConstructor, DakotaModel & model, int samples, int seed, int num_vars, const DakotaRealVector & lower_bnds, const DakotaRealVector & upper_bnds)

This alternate constructor is used by ConcurrentStrategy for generation of uniform, uncorrelated sample sets.

6.72.3 Member Function Documentation

6.72.3.1 void NonDSampling::sampling_reset (int min_samples, bool all_data_flag, bool stats_flag) [inline, virtual]

resets number of samples and sampling flags.

used by ApproximationInterface::build_global_approximation() to publish the minimum number of samples needed from the sampling routine (to build a particular global approximation) and to set allDataFlag and statsFlag. In this case, allDataFlag is set to true (vectors of variable and response sets must be returned to build the global approximation) and statsFlag is set to false (statistics computations are not needed).

Reimplemented from DakotaIterator.

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

- NonDSampling.H
- NonDSampling.C

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2002
6.73 NPSOLOptimizer Class Reference

Wrapper class for the NPSOL optimization library.

Inheritance diagram for NPSOLOptimizer:

```
DakotaIterator
  |        |
  |        | DakotaOptLeastSq
  |        | DakotaOptimizer
  |        | SOLBase
  |        | NPSOLOptimizer
```

**Public Methods**

- **NPSOLOptimizer (DakotaModel &model)**
  
  *standard constructor.*

- **NPSOLOptimizer (const DakotaRealVector &initial_point, const DakotaRealVector &var_lower_,
  bnds, const DakotaRealVector &var_upper_,
  bnds, int num_lin_ineq, int num_linear_eq, int num_nonlin_ineq,
  int num_nonlinear_eq, const DakotaRealMatrix &lin_ineq_coeffs,
  const DakotaRealVector &lin_ineq_lower_bnds,
  const DakotaRealVector &lin_ineq_upper_bnds,
  const DakotaRealMatrix &lin_eq_coeffs,
  const DakotaRealVector &lin_eq_targets,
  const DakotaRealVector &nonlin_ineq_lower_bnds,
  const DakotaRealVector &nonlin_ineq_upper_bnds,
  const DakotaRealVector &nonlin_eq_targets, void(*user_obj_eval)(int &, int &, Real ,
  Real *, Real &, Real *, int &), void(*user_con_eval)(int &,
  int &), int derivative_level)**

  *special constructor for instantiations "on the fly".*

- ~NPSOLOptimizer ()

  *destructor.*

- void find_optimum ()

  *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

**Private Methods**

- void find_optimum_on_model ()

  *called by find_optimum for setUpType == "model".*

- void find_optimum_on_user_functions ()

  *called by find_optimum for setUpType == "user functions".*
6.73 NPSOLOptimizer Class Reference

Static Private Methods

- void objective_eval (int &mode, int &n, Real *x, Real &f, Real *gradf, int &nstate)
  
  **OBJFUN in NPSOL manual:** computes the value and first derivatives of the objective function (passed by function pointer to NPSOL).

Private Attributes

- DakotaString setUpType
  
  controls iteration mode: "model" (normal usage) or "user functions" (user-supplied functions mode for "on the fly" instantiations). NonDAdvMeanValue currently uses the user functions mode.

- DakotaRealVector initialPoint
  
  holds initial point passed in for "user functions" mode.

- DakotaRealVector lowerBounds
  
  holds variable lower bounds passed in for "user functions" mode.

- DakotaRealVector upperBounds
  
  holds variable upper bounds passed in for "user functions" mode.

- void (*userObjectiveEval)(int &, int &, Real *, Real &, Real *, int &)
  
  holds function pointer for objective function evaluator passed in for "user functions" mode.

- void (*userConstraintEval)(int &, int &, int &, int &, int &, Real *, Real *, Real *, int &)
  
  holds function pointer for constraint function evaluator passed in for "user functions" mode.

6.73.1 Detailed Description

Wrapper class for the NPSOL optimization library.

The NPSOLOptimizer class provides a wrapper for NPSOL, a Fortran 77 sequential quadratic programming library from Stanford University marketed by Stanford Business Associates. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or static as well. To isolate the effect of these static requirements from the rest of the iterator hierarchy, static copies are made of many non-static attributes inherited from above.

The user input mappings are as follows: max_function_evaluations is implemented directly in NPSOLOptimizer’s evaluator functions since there is no NPSOL parameter equivalent, and max_iterations, convergence_tolerance, output verbosity, verify_level, function_precision, and linesearch_tolerance are mapped into NPSOL’s "Major Iteration Limit", "Optimality Tolerance", "Major Print Level" (verbose: Major Print Level = 20; quiet: Major Print Level = 10), "Verify Level", "Function Precision", and "Linesearch Tolerance" parameters, respectively, using NPSOL’s npoptn() subroutine (as wrapped by npoptn2() from the npoptn_wrapper.f file). Refer to [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] for information on NPSOL’s optional input parameters and the npoptn subroutine.

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

- NPSOLOptimizer.H
- NPSOLOptimizer.C
6.74 ParallelLibrary Class Reference

Class for managing partitioning of multiple levels of parallelism and message passing within the levels.

Public Methods

- **ParallelLibrary** (int &argc, char **&argv)
  constructor.

- **ParallelLibrary ()**
  default constructor.

- **ParallelLibrary (int dummy)**
  dummy constructor (used for dummy lib).

- **~ParallelLibrary ()**
  destructor.

- void **init_iterator_communicators** (const ProblemDescDB &problem_db)
  split MPI COMM WORLD into iterator communicators.

- void **init_evaluation_communicators** (int eval_servers, int procs_per_eval, int max_concurrency, int asynch_local_eval_concurrency, const DakotaString &eval_scheduling)
  split an iterator communicator into evaluation communicators.

- void **init_analysis_communicators** (int analysis_servers, int procs_per_analysis, int max_concurrency, int asynch_local_analysis_concurrency, const DakotaString &analysis_scheduling)
  split an evaluation communicator into analysis communicators.

- void **free_iterator_communicators** ()
  deallocate iterator communicators.

- void **free_evaluation_communicators** ()
  deallocate evaluation communicators.

- void **free_analysis_communicators** ()
  deallocate analysis communicators.

- void **print_configuration** ()
  print the parallel configuration for all parallelism levels.

- void **manage_outputs_restart** (CommandLineHandler &cmdline_handler)
  manage output streams and restart file(s) using command line inputs (normal mode).

- void **manage_outputs_restart** (const char *clh_std_output_filename, const char *clh_std_error_filename, const char *clh_read_restart_filename, const char *clh_write_restart_filename, int restart_evals)
manage output streams and restart file(s) using external inputs (library mode).

- void close_streams()
  close streams, files, and any other services.

- void send(si) (PackBuffer &send_buffer, int dest, int tag)
  blocking send at the strategy-iterator communication level.

- void isend(si) (PackBuffer &send_buffer, int dest, int tag, MPI_Request &send_request)
  nonblocking send at the strategy-iterator communication level.

- void recv(si) (UnPackBuffer &recv_buffer, int source, int tag, MPI_Status &status)
  blocking receive at the strategy-iterator communication level.

- void irecv(si) (UnPackBuffer &recv_buffer, int source, int tag, MPI_Request &recv_request)
  nonblocking receive at the strategy-iterator communication level.

- void send.ie (PackBuffer &send_buffer, int dest, int tag)
  blocking send at the iterator-evaluation communication level.

- void isend.ie (PackBuffer &send_buffer, int dest, int tag, MPI_Request &send_request)
  nonblocking send at the iterator-evaluation communication level.

- void recv.ie (UnPackBuffer &recv_buffer, int source, int tag, MPI_Status &status)
  blocking receive at the iterator-evaluation communication level.

- void irecv.ie (UnPackBuffer &recv_buffer, int source, int tag, MPI_Request &recv_request)
  nonblocking receive at the iterator-evaluation communication level.

- void send.ea (int &send_int, int dest, int tag)
  blocking send at the evaluation-analysis communication level.

- void isend.ea (int &send_int, int dest, int tag, MPI_Request &send_request)
  nonblocking send at the evaluation-analysis communication level.

- void recv.ea (int &recv_int, int source, int tag, MPI_Status &status)
  blocking receive at the evaluation-analysis communication level.

- void irecv.ea (int &recv_int, int source, int tag, MPI_Request &recv_request)
  nonblocking receive at the evaluation-analysis communication level.

- void bcast (int &data, MPI_Comm comm)
  broadcast an integer across a communicator.

- void bcast (PackBuffer &send_buffer, MPI_Comm comm)
  send a packed buffer across a communicator using a broadcast.

- void bcast (UnPackBuffer &recv_buffer, MPI_Comm comm)
  matching receive for a packed buffer broadcast.
void waitall (int num_recs, MPI_Request *recv_requests)
wait for all messages from a series of nonblocking receives.

int world_size () const
return worldSize.

int world_rank () const
return worldRank.

short parallelism_levels () const
return parallelismLevels.

bool mpirun_flag () const
return mpirunFlag.

Real parallel_time () const
returns current MPI wall clock time.

bool strategy_dedicated_master_flag () const
return strategyDedicatedMasterFlag.

bool strategy_iterator_split_flag () const
return straIteratorSplitFlag.

bool iterator_master_flag () const
return iteratorMasterFlag.

bool strategy_iterator_message_pass () const
return straIteratorMessagePass.

MPI_Comm iterator_intra_communicator () const
return iteratorIntraComm.

MPI_Comm strategy_iterator_intra_communicator () const
return straIteratorIntraComm.

MPI_Comm strategy_iterator_inter_communicator () const
return straIteratorInterComm.

MPI_Comm * strategy_iterator_inter_communicators () const
return straIteratorInterComms.

int iterator_servers () const
return numIteratorServers.

int iterator_communicator_rank () const
return iteratorCommRank.

int iterator_communicator_size () const
return iteratorCommSize.
• int \texttt{strategy\_iterator\_communicator\_rank} () const
  
  \hspace{1em} \texttt{return stratIteratorCommRank.}

• int \texttt{strategy\_iterator\_communicator\_size} () const
  
  \hspace{1em} \texttt{return stratIteratorCommSize.}

• int \texttt{iterator\_server\_id} () const
  
  \hspace{1em} \texttt{return iteratorServerId.}

• bool \texttt{iterator\_dedicated\_master\_flag} () const
  
  \hspace{1em} \texttt{return iteratorDedicatedMasterFlag.}

• bool \texttt{iterator\_eval\_split\_flag} () const
  
  \hspace{1em} \texttt{return iteratorEvalSplitFlag.}

• bool \texttt{evaluation\_master\_flag} () const
  
  \hspace{1em} \texttt{return evalMasterFlag.}

• bool \texttt{iterator\_eval\_message\_pass} () const
  
  \hspace{1em} \texttt{return iteratorEvalMessagePass.}

• \texttt{MPI\_Comm} \texttt{evaluation\_intra\_communicator} () const
  
  \hspace{1em} \texttt{return evalIntraComm.}

• \texttt{MPI\_Comm} \texttt{iterator\_eval\_intra\_communicator} () const
  
  \hspace{1em} \texttt{return iteratorEvalIntraComm.}

• \texttt{MPI\_Comm} \texttt{iterator\_eval\_inter\_communicator} () const
  
  \hspace{1em} \texttt{return iteratorEvalInterComm.}

• \texttt{MPI\_Comm} * \texttt{iterator\_eval\_inter\_communicators} () const
  
  \hspace{1em} \texttt{return iteratorEvalInterComms.}

• int \texttt{evaluation\_servers} () const
  
  \hspace{1em} \texttt{return numEvalServers.}

• int \texttt{evaluation\_communicator\_rank} () const
  
  \hspace{1em} \texttt{return evalCommRank.}

• int \texttt{evaluation\_communicator\_size} () const
  
  \hspace{1em} \texttt{return evalCommSize.}

• int \texttt{iterator\_eval\_communicator\_rank} () const
  
  \hspace{1em} \texttt{return iteratorEvalCommRank.}

• int \texttt{iterator\_eval\_communicator\_size} () const
  
  \hspace{1em} \texttt{return iteratorEvalCommSize.}

• int \texttt{evaluation\_server\_id} () const
6.74 ParallelLibrary Class Reference

```cpp
return evalServerId.

- bool evaluation_dedicated_master_flag () const
  return evalDedicatedMasterFlag.

- bool evaluation_analysis_split_flag () const
  return evalAnalysisSplitFlag.

- bool analysis_master_flag () const
  return analysisMasterFlag.

- bool evaluation_analysis_message_pass () const
  return evalAnalysisMessagePass.

- MPI_Comm analysis_intra_communicator () const
  return analysisIntraComm.

- MPI_Comm evaluation_analysis_intra_communicator () const
  return evalAnalysisIntraComm.

- MPI_Comm evaluation_analysis_inter_communicator () const
  return evalAnalysisInterComm.

- MPI_Comm * evaluation_analysis_inter_communicators () const
  return evalAnalysisInterComms.

- int analysis_servers () const
  return numAnalysisServers.

- int analysis_communicator_rank () const
  return analysisCommRank.

- int analysis_communicator_size () const
  return analysisCommSize.

- int evaluation_analysis_communicator_rank () const
  return evalAnalysisCommRank.

- int evaluation_analysis_communicator_size () const
  return evalAnalysisCommSize.

- int analysis_server_id () const
  return analysisServerId.
```
Private Methods


  split a parent communicator into a dedicated master processor and num_servers child communicators.


  split a parent communicator into num_servers child communicators (no dedicated master processor).

- bool resolve_inputs (int &num_servers, int &procs_per_server, const int &avail_procs, int &proc_remainder, const int &max_concurrency, const int &capacity_multiplier, const DakotaString &default_config, const DakotaString &scheduling_override)

  Resolve user inputs into a sensible partitioning scheme.

Private Attributes

- ofstream output ofstream
  tagged file redirection of stdout.

- ofstream error ofstream
  tagged file redirection of stderr.

- int worldRank
  rank in MPI_COMM_WORLD.

- int worldSize
  size of MPI_COMM_WORLD.

- short parallelismLevels
  number of parallelism levels.

- bool mpirunFlag
  flag for a parallel mpirun/yod launch.

- bool ownMPIFlag
  flag for ownership of MPI_Init/MPI_Finalize.

- bool dummyFlag
  prevents multiple MPI_Finalize calls due to dummy_lib.

- bool stdOutputFlag
flags redirection of DAKOTA std output to a file.

- bool stdErrorFlag
  flags redirection of DAKOTA std error to a file.

- Real startCPUTime
  start reference for UTILIB CPU timer.

- Real startWCTime
  start reference for UTILIB wall clock timer.

- Real startMPITime
  start reference for MPI wall clock timer.

- long startClock
  start reference for local clock() timer measuring parent+child CPU.

- bool strategyDedicatedMasterFlag
  signals ded. master partitioning.

- bool stratIteratorSplitFlag
  signals a communicator split was used.

- bool iteratorMasterFlag
  identifies master iterator processors.

- bool stratIteratorMessagePass
  flag for message passing at si level.

- MPI_Comm iteratorIntraComm
  intracomm for each iterator partition.

- MPI_Comm stratIteratorIntraComm
  intracomm for all iteratorCommRank==0 w/i MPI_COMM_World.

- MPI_Comm stratIteratorInterComm
  intercomm between an iterator & master strategy (on iterator partitions only).

- MPI_Comm * stratIteratorInterComms
  intercomm. array on master strategy.

- int numIteratorServers
  number of iterator servers.

- int procsPerIterator
  processors per iterator server.

- int iteratorCommRank
  rank in iteratorIntraComm.
- `int iteratorCommSize`
  
  size of `iteratorIntraComm`.

- `int stratIteratorCommRank`
  
  rank in `stratIteratorIntraComm`.

- `int stratIteratorCommSize`
  
  size of `stratIteratorIntraComm`.

- `int iteratorServerId`
  
  identifier for an iterator server.

- `bool iteratorDedicatedMasterFlag`
  
  signals dedicated master partitioning.

- `bool iteratorEvalSplitFlag`
  
  signals a communicator split was used.

- `bool evalMasterFlag`
  
  identifies master evaluation processors.

- `bool iteratorEvalMessagePass`
  
  flag for message passing at `ie` level.

- `MPI_Comm evalIntraComm`
  
  intracomm for each fn. eval. partition.

- `MPI_Comm iteratorEvalIntraComm`
  
  intracomm for all `evalCommRank==0` w/i `iteratorIntraComm`.

- `MPI_Comm iteratorEvalInterComm`
  
  intercomm between a fn. eval. & master iterator (on fn. eval. partitions only).

- `MPI_Comm * iteratorEvalInterComms`
  
  intercomm array on master iterator.

- `int numEvalServers`
  
  number of evaluation servers.

- `int procsPerEval`
  
  processors per evaluation server.

- `int evalCommRank`
  
  rank in `evalIntraComm`.

- `int evalCommSize`
  
  size of `evalIntraComm`.

- `int iteratorEvalCommRank`
  
  rank in `iteratorEvalIntraComm`. 
- `int iteratorEvalCommSize`
  size of `iteratorEvalIntraComm`.

- `int evalServerId`
  identifier for an evaluation server.

- `bool evalDedicatedMasterFlag`
  signals dedicated master partitioning.

- `bool evalAnalysisSplitFlag`
  signals a communicator split was used.

- `bool analysisMasterFlag`
  identifies master analysis processors.

- `bool evalAnalysisMessagePass`
  flag for message passing at ea level.

- `MPI_Comm analysisIntraComm`
  intracomm for each analysis partition.

- `MPI_Comm evalAnalysisIntraComm`
  intracomm for all analysisCommRank==0 w/i evalIntraComm.

- `MPI_Comm evalAnalysisInterComm`
  intercomm between an analysis & master fn. eval. (on analysis partitions only).

- `MPI_Comm * evalAnalysisInterComms`
  intercomm array on master fn. eval.

- `int numAnalysisServers`
  number of analysis servers.

- `int procsPerAnalysis`
  processors per analysis server.

- `int analysisCommRank`
  rank in `analysisIntraComm`.

- `int analysisCommSize`
  size of `analysisIntraComm`.

- `int evalAnalysisCommRank`
  rank in `evalAnalysisIntraComm`.

- `int evalAnalysisCommSize`
  size of `evalAnalysisIntraComm`.

- `int analysisServerId`
  identifier for an analysis server.
6.74.1 Detailed Description

Class for managing partitioning of multiple levels of parallelism and message passing within the levels.

The ParallelLibrary class encapsulates all of the details of performing message passing within multiple levels of parallelism. It provides functions for partitioning of levels according to user configuration input and functions for passing messages within and across MPI communicators for each of the parallelism levels. If support for other message-passing libraries beyond MPI becomes needed, then ParallelLibrary should become a class hierarchy with virtual functions to encapsulate the library-specific syntax.

6.74.2 Constructor & Destructor Documentation

6.74.2.1 ParallelLibrary::ParallelLibrary (int & argc, char **& argv)

constructor.

This constructor is the one used by main.C. It calls MPI_Init conditionally based on whether a parallel launch is detected.

6.74.2.2 ParallelLibrary::ParallelLibrary ()

default constructor.

This constructor provides a library mode and is used by the SIERRA Adak application. It does not call MPI_Init, but rather gathers data from MPI_COMM_WORLD if MPI_Init has been called elsewhere.

6.74.2.3 ParallelLibrary::ParallelLibrary (int dummy)

dummy constructor (used for dummy_lib).

This constructor is used for creation of the global dummy_lib object, which is used to satisfy initialization requirements when the real ParallelLibrary object is not available.

6.74.3 Member Function Documentation

6.74.3.1 void ParallelLibrary::init_iterator_communicators (const ProblemDescDB & problem_db)

split MPI_COMM_WORLD into iterator communicators.

Split MPI_COMM_WORLD into the specified number of subcommunicators to set up concurrent iterator partitions serving a strategy. This constructs new iterator intra-communicators and strategy-iterator inter-communicators. The init_iterator_communicators() and free_iterator_communicators() functions are both called from main.C, and init_iterator_communicators() is called prior to output and restart management since output and restart files are tagged based on iterator server id.
split an iterator communicator into evaluation communicators.

Split iteratorIntraComm (=MPI_COMM_WORLD if no concurrence in iterators) as specified by the passed parameters to set up concurrent evaluation partitions serving an iterator. This constructs new evaluation intra-communicators and iterator-evaluation inter-communicators. **init_evaluation_communicators**() is called from **ApplicationInterface::init_communicators**() and **free_evaluation_communicators**() function is called from **ApplicationInterface::free_communicators**(). eval_servers, asynch_local_eval_concurrency, and eval_scheduling come from the interface keyword specification. procs_per_eval is not directly user-specified, rather it contains the minimum procs_per_eval required to support any lower level user requests (such as procs_per_analysis). max_concurrency is passed in via the function DakotaIterator::max_concurrency(), which queries individual methods for their gradient configuration, population size, etc. These partitions can be reconfigured for each iterator/model pair within a strategy (e.g. interface 1 uses 4 by 256 while interface 2 uses 2 by 512) – see **DakotaStrategy::run_iterator**().

split an evaluation communicator into analysis communicators.

Split evalIntraComm as indicated by the passed parameters to set up concurrent analysis partitions serving a function evaluation. This constructs new analysis intra-communicators and evaluation-analysis inter-communicators. **init_analysis_communicators**() is called from **ApplicationInterface::init_communicators**() following the call to **init_evaluation_communicators**() and **free_analysis_communicators**() is called from **ApplicationInterface::free_communicators**() preceding the call to **free_evaluation_communicators**(). The analysis_servers, procs_per_analysis, asynch_local_analysis_concurrency, and analysis_scheduling attributes come from the interface keyword specification, and max_concurrency contains the length of analysis_drivers from the interface keyword specification. The analysis partitions can be reconfigured for each iterator/model pair within a strategy.

manage output streams and restart file(s) using command line inputs (normal mode).

Get the -output, -error, -read_restart, and -write_restart filenames and the -stop_restart limit from the command line. Defaults for the filenames from the command line handler are NULL for the filenames and 0 for restart_evals if no user specification. Only worldRank==0 has access to command line arguments and must Bcast this data to all iterator masters.

manage output streams and restart file(s) using external inputs (library mode).

If the user has specified the use of files for DAKOTA standard output and/or standard error, then bind these filenames to the Cout/Cerr macros. In addition, if concurrent iterators are to be used, create and tag multiple output streams in order to prevent jumbled output. Manage restart file(s) by processing any incoming evaluations from an old restart file and by setting up the binary output stream for new evaluations.
Only master iterator processor(s) read & write restart information. This function must follow init_iterator_ communicators so that restart can be managed properly for concurrent iterator strategies. In the case of concurrent iterators, each iterator has its own restart file tagged with iterator number.

### 6.74.3.6 void ParallelLibrary::close_streams ()

close streams, files, and any other services.

Close streams associated with manage_outputs and manage_restart and terminate any additional services that may be active.

### 6.74.3.7 bool ParallelLibrary::resolve_inputs (int & num_servers, int & procs_per_server, const int & avail_procs, int & proc_remainder, const int & max_concurrency, const int & capacity_multiplier, const DakotaString & default_config, const DakotaString & scheduling_override) [private]

Resolve user inputs into a sensible partitioning scheme.

This function is responsible for the "auto-configure" intelligence of DAKOTA. It resolves a variety of inputs and overrides into a sensible partitioning configuration for a particular parallelism level. It also handles the general case in which a user’s specification request does not divide out evenly with the number of available processors for the level. If num_servers & procs_per_server are both nondefault, then the former takes precedence.

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

- ParallelLibrary.H
- ParallelLibrary.C
6.75  **ParamResponsePair Class Reference**

Container class for a variables object, a response object, and an evaluation id.

**Public Methods**

- `ParamResponsePair ()`
  default constructor.

- `ParamResponsePair (const DakotaVariables &vars, const DakotaResponse &response)`
  alternate constructor for temporaries.

- `ParamResponsePair (const DakotaVariables &vars, const DakotaResponse &response, const int id)`
  standard constructor for history uses.

- `ParamResponsePair (const ParamResponsePair &pair)`
  copy constructor.

- `~ParamResponsePair ()`
  destructor.

- `ParamResponsePair & operator= (const ParamResponsePair &pair)`
  assignment operator.

- `void read (istream &s)`
  read a ParamResponsePair object from an istream.

- `void write (ostream &s) const`
  write a ParamResponsePair object to an ostream.

- `void read_annotated (istream &s)`
  read a ParamResponsePair object in annotated format from an istream.

- `void write_annotated (ostream &s) const`
  write a ParamResponsePair object in annotated format to an ostream.

- `void write_tabular (ostream &s) const`
  write a ParamResponsePair object in tabular format to an ostream.

- `void read (DakotaBiStream &s)`
  read a ParamResponsePair object from the binary restart stream.

- `void write (DakotaBoStream &s) const`
  write a ParamResponsePair object to the binary restart stream.

- `void read (UnPackBuffer &s)`
read a ParamResponsePair object from a packed MPI buffer.

- void write (PackBuffer &s) const
  write a ParamResponsePair object to a packed MPI buffer.

- int eval_id () const
  return the evaluation identifier.

- const DakotaVariables & prp_parameters () const
  return the parameters object.

- const DakotaResponse & prp_response () const
  return the response object.

- void prp_response (const DakotaResponse &response)
  set the response object.

- const DakotaIntArray & active_set_vector () const
  return the active set vector from the response object.

- void active_set_vector (const DakotaIntArray &asv)
  set the active set vector in the response object.

- const DakotaString & interface_id () const
  return the interface identifier from the response object.

Private Attributes

- DakotaVariables prPairParameters
  the set of parameters for the function evaluation.

- DakotaResponse prPairResponse
  the response set for the function evaluation.

- int evalId
  the function evaluation identifier (assigned from ApplicationInterface::fnEvalId).

Friends

- bool operator== (const ParamResponsePair &pair1, const ParamResponsePair &pair2)
  equality operator.

- bool operator!= (const ParamResponsePair &pair1, const ParamResponsePair &pair2)
  inequality operator.
6.75 ParamResponsePair Class Reference

6.75.1 Detailed Description

Container class for a variables object, a response object, and an evaluation id.

ParamResponsePair provides a container class for association of the input for a particular function evaluation (a variables object) with the output from this function evaluation (a response object), along with an evaluation identifier. This container defines the basic unit used in the data_pairs list, in restart file operations, and in a variety of scheduling algorithm bookkeeping operations. With the advent of STL, replacement of this class with the pair<> template construct may be possible (using pair<int, pair<vars,response> >, for example), assuming that deep copies, I/O, alternate constructors, etc., can be adequately addressed.

6.75.2 Constructor & Destructor Documentation

6.75.2.1 ParamResponsePair::ParamResponsePair (const DakotaVariables & vars, const DakotaResponse & response) [inline]

alternate constructor for temporaries.

This constructor can use the standard DakotaVariables and DakotaResponse copy constructors to share representations since this constructor is used for search_pairs (which are local instantiations that go out of scope prior to any changes to values; i.e., they are not used for history).

6.75.2.2 ParamResponsePair::ParamResponsePair (const DakotaVariables & vars, const DakotaResponse & response, const int id) [inline]

standard constructor for history uses.

This constructor cannot share representations since it involves a history mechanism (beforeSynchPRPLList or data_pairs). Deep copies must be made.

6.75.3 Member Data Documentation

6.75.3.1 int ParamResponsePair::evalId [private]

the function evaluation identifier (assigned from ApplicationInterface::fnEvalId).

evalId belongs here rather than in DakotaResponse since some DakotaResponse objects involve consolidation of several fn. evals. (e.g., synchronize_fd_gradients). The prPair, on the other hand, is used for storage of all low level fn. evals. that get evaluated, so evalId is meaningful.

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

- ParamResponsePair.H
- ParamResponsePair.C
6.76 ParamStudy Class Reference

Class for vector, list, centered, and multidimensional parameter studies.

Inheritance diagram for ParamStudy::

```
    DakotaIterator
     |__________
       ParamStudy
```

**Public Methods**

- **ParamStudy** (DakotaModel &model)
  *constructor.*

- **~ParamStudy** ()
  *destructor.*

- void **run_iterator** ()
  *run the iterator.*

- const DakotaVariables & **iterator_variable_results** () const
  *return the final iterator solution (variables).*

- const DakotaResponse & **iterator_response_results** () const
  *return the final iterator solution (response).*

- void **print_iterator_results** (ostream &s) const
  *print the final iterator results.*

**Private Methods**

- void **compute_vector_steps** ()
  *computes stepVector and numSteps from initialPoint, finalPoint, and either numSteps or stepLength (pStudy-Type is 1 or 2).*

- void **vector_loop** (const DakotaRealVector &start, const DakotaRealVector &step_vect, const int &num_steps)
  *performs the parameter study by looping from start in num_steps increments of step_vect. Total number of evaluations is num_steps + 1.*

- void **sample** (const DakotaRealVector &list_of_points)
  *performs the parameter study by sampling from a list of points.*
void centered_loop (const DakotaRealVector &start, const Real &percent_delta, const int &deltas_per_variable)

performs a number of plus and minus offsets for each parameter centered about start.

void multidim_loop (const DakotaIntArray &var_partitions)

performs vector loops recursively in multiple dimensions.

void recurse (int nloop, int nindex, DakotaIntArray &current_index, const DakotaIntArray &max_index, const DakotaRealVector &start, const DakotaRealVector &step_vect)

used by multidim_loop to enable a variable number of nested loops.

void update_best (const DakotaRealVector &vars, const DakotaResponse &response, const int eval_num)

compares current evaluation to best evaluation and updates best.

Private Attributes

- DakotaRealVector list_of_points
  list of evaluation points for the list parameter study.

- DakotaRealVector initial_point
  the starting point for vector and centered parameter studies.

- DakotaRealVector final_point
  the ending point for vector parameter study (a specification option).

- DakotaRealVector step_vector
  the n-dimensional increment in vector parameter study.

- int num_steps
  the number of times step_vector is applied in vector parameter study.

- int p_study_type
  internal code for parameter study type: -1 (list), 1,2,3 (different vector specifications), 4 (centered), or 5 (multidim).

- int deltas_per_variable
  number of offsets in the plus and the minus direction for each variable in a centered parameter study.

- bool nested_flag
  flag set by parameter studies which call other parameter studies in loops.

- Real step_length
  the Cartesian length of multidimensional steps in vector parameter study (a specification option).

- Real percent_delta
  size of relative offsets in percent for each variable in a centered parameter study.
- DakotaIntArray variablePartitions
  number of partitions for each variable in a multidim-parameter study.

- DakotaVariables bestVariables
  best variables found during the study.

- DakotaResponse bestResponses
  best responses found during the study.

- Real bestObjectiveFn
  best objective function found during the study.

- Real bestViolations
  best constraint violations found during the study. In the current approach, constraint violation reduction takes strict precedence over objective function reduction.

- size_t numObjectiveFunctions
  number of objective functions. Used in update_best.

- size_t numNonlinearIneqConstraints
  number of nonlinear inequality constraints. Used in update_best.

- size_t numNonlinearEqConstraints
  number of nonlinear equality constraints. Used in update_best.

- DakotaRealVector multiObjWeights
  vector of multiobjective weights. Used in update_best.

- DakotaRealVector nonlinearIneqLowerBnds
  vector of nonlinear inequality constraint lower bounds. Used in update_best.

- DakotaRealVector nonlinearIneqUpperBnds
  vector of nonlinear inequality constraint upper bounds. Used in update_best.

- DakotaRealVector nonlinearEqTargets
  vector of nonlinear equality constraint targets. Used in update_best.

- int psCounter
  class-scope counter (needed for asynchronous multidim_loop).

### 6.76.1 Detailed Description

Class for vector, list, centered, and multidimensional parameter studies.

The ParamStudy class contains several algorithms for performing parameter studies of different types. It is not a wrapper for an external library, rather its algorithms are self-contained. The vector parameter study steps along an n-dimensional vector from an arbitrary initial point to an arbitrary final point in a specified number of steps. The centered parameter study performs a number of plus and minus offsets in each coordinate direction around a center point. A multidimensional parameter study fills an n-dimensional
hypercube based on a specified number of intervals for each dimension. It is a nested study in that it utilizes
the vector parameter study internally as it recurses through the variables. And the list parameter study
provides for a user specification of a list of points to evaluate, which allows general parameter investigations
not fitting the structure of vector, centered, or multidim parameter studies.

6.76.2 Member Function Documentation

6.76.2.1 void ParamStudy::run_iterator () [virtual]

run the iterator.

This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine
it.

Reimplemented from DakotaIterator.

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

- ParamStudy.H
- ParamStudy.C
6.77 ProblemDescDB Class Reference

The database containing information parsed from the DAKOTA input file.

Public Methods

- **ProblemDescDB** (ParallelLibrary &parallel\_lib)
  
  constructor.

- **\~ProblemDescDB** ()
  
  destructor.

- void **manage\_inputs** (int argc, char *\*argv, CommandLineHandler &cmd\_line\_handler)
  
  parses the input file and populates the problem description database. This version reads from the dakota input filename passed with the "-input" option on the DAKOTA command line.

- void **manage\_inputs** (const char *dakota\_input\_file)
  
  parses the input file and populates the problem description database. This version reads from the dakota input filename passed in.

- void **check\_input** ()
  
  verifies that there was at least one of each of the required keywords in the dakota input file. Used by manage\_inputs().

- void **set\_db\_list\_nodes** (const DakotaString &method\_tag)
  
  set methodIter based on the method identifier string to activate a particular method specification in methodList and use pointers from this method specification to set the other list iterators.

- void **set\_db\_list\_nodes** (const size\_t &method\_index)
  
  set methodIter based on the active index to activate a particular method specification in methodList and use pointers from this method specification to set the other list iterators.

- size\_t **get\_db\_list\_nodes** () const
  
  return the index of the active node in methodList.

- void **set\_db\_interface\_node** (const DakotaString &interface\_tag)
  
  set interfaceIter based on the interface identifier string.

- void **set\_db\_responses\_node** (const DakotaString &responses\_tag)
  
  set responsesIter based on the responses identifier string.

- void **set\_db\_model\_type** (const DakotaString &model\_type)
  
  set the model type.

- ParallelLibrary & parallel\_library () const
  
  return the parallelLib reference.
- `const DakotaRealVector & get_drv (const DakotaString &entry_name) const` get a DakotaRealVector out of the database based on an identifier string.
- `const DakotaIntVector & get_div (const DakotaString &entry_name) const` get a DakotaIntVector out of the database based on an identifier string.
- `const DakotaRealArray & get_dra (const DakotaString &entry_name) const` get a DakotaRealArray out of the database based on an identifier string.
- `const DakotaIntArray & get_dia (const DakotaString &entry_name) const` get a DakotaIntArray out of the database based on an identifier string.
- `const DakotaRealMatrix & get_drm (const DakotaString &entry_name) const` get a DakotaRealMatrix out of the database based on an identifier string.
- `const DakotaRealVectorArray & get_drva (const DakotaString &entry_name) const` get a DakotaRealVectorArray out of the database based on an identifier string.
- `const DakotaIntList & get_dil (const DakotaString &entry_name) const` get a DakotaIntList out of the database based on an identifier string.
- `const DakotaStringArray & get_dsa (const DakotaString &entry_name) const` get a DakotaStringArray out of the database based on an identifier string.
- `const DakotaStringList & get.dsl (const DakotaString &entry_name) const` get a DakotaStringList out of the database based on an identifier string.
- `const DakotaString & get_string (const DakotaString &entry_name) const` get a DakotaString out of the database based on an identifier string.
- `const Real & get_real (const DakotaString &entry_name) const` get a Real out of the database based on an identifier string.
- `const int & get_int (const DakotaString &entry_name) const` get an int out of the database based on an identifier string.
- `const size_t & get_sizet (const DakotaString &entry_name) const` get a size_t out of the database based on an identifier string.
- `const bool & get_bool (const DakotaString &entry_name) const` get a bool out of the database based on an identifier string.
- `void insert_node (const DataStrategy &data_strategy)` set the DataStrategy object.
- `void insert_node (const DataMethod &data_method)` add a DataMethod object to the methodList.
- `void insert_node (const DataVariables &data_variables)` add a DataVariables object to the variablesList.
void insert_node (const DataInterface &data_interface)
add a DataInterface object to the interfaceList.

void insert_node (const DataResponses &data_responses)
add a DataResponses object to the responsesList.

Static Public Methods

void method_kwhandler (const struct FunctionData &parsed_data)
method keyword handler called by IDR when a complete method specification is parsed.

void variables_kwhandler (const struct FunctionData &parsed_data)
variables keyword handler called by IDR when a complete variables specification is parsed.

void interface_kwhandler (const struct FunctionData &parsed_data)
interface keyword handler called by IDR when a complete interface specification is parsed.

void responses_kwhandler (const struct FunctionData &parsed_data)
responses keyword handler called by IDR when a complete responses specification is parsed.

void strategy_kwhandler (const struct FunctionData &parsed_data)
strategy keyword handler called by IDR when a complete strategy specification is parsed.

Private Methods

void send_db_buffer ()
MPI send of a large buffer containing strategy specification attributes and all the objects in interfaceList, variablesList, methodList, and responsesList. Used by manage_inputs().

void receive_db_buffer ()
MPI receive of a large buffer containing strategy specification attributes and all the objects in interfaceList, variablesList, methodList, and responsesList. Used by manage_inputs().

void set_other_list_nodes ()
convenience function used by set_db_list_nodes(method_tag) and set_db_list_nodes(method_index) to set the other list iterators once methodIter is set (based on pointers from the method specification).

Static Private Methods

void build_label (DakotaString &label, const DakotaString &root_label, size_t tag)
create a label by appending tag to root_label.

void build_labels (DakotaStringArray &label_array, const DakotaString &root_label)
create an array of labels by tagging root_label for each entry in label_array. Uses build_label().
• void **build_labels_partial** (DakotaStringArray &label_array, const DakotaString &root_label, size_t start_index, size_t num_items)
  
  *create a partial array of labels by tagging root_label for a subset of entries in label_array. Uses build_label().*

**Private Attributes**

• **DakotaList< DataMethod >::iterator methodIter**
  
  *iterator identifying the active list node in methodList.*

• **DakotaList< DataVariables >::iterator variablesIter**
  
  *iterator identifying the active list node in variablesList.*

• **DakotaList< DataInterface >::iterator interfaceIter**
  
  *iterator identifying the active list node in interfaceList.*

• **DakotaList< DataResponses >::iterator responsesIter**
  
  *iterator identifying the active list node in responsesList.*

• **bool dbLocked**
  
  *prevents use of get<type> data retrieval functions prior to a set<lib>nodes invocation.*

• **ParallelLibrary & parallelLib**
  
  *reference to the parallelLib object passed from main.*

**Static Private Attributes**

• **DataStrategy strategySpec**
  
  *the strategy specification (only one allowed) resulting from a call to strategyKwHandler() or insertNode().*

• **DakotaList< DataMethod > methodList**
  
  *list of method specifications, one for each call to methodKwHandler() or insertNode().*

• **DakotaList< DataVariables > variablesList**
  
  *list of variables specifications, one for each call to variablesKwHandler() or insertNode().*

• **DakotaList< DataInterface > interfaceList**
  
  *list of interface specifications, one for each call to interfaceKwHandler() or insertNode().*

• **DakotaList< DataResponses > responsesList**
  
  *list of responses specifications, one for each call to responsesKwHandler() or insertNode().*

• **size_t strategyCntr**
  
  *counter for strategy specifications used in check_input.*
6.77.1 Detailed Description

The database containing information parsed from the DAKOTA input file.

The ProblemDescDB class is a database for DAKOTA input file data that is populated by the Input Deck Reader (IDR) parser. When the parser reads a complete keyword (delimited by a newline), it calls the corresponding kwhandler function from this class, which (for method, variables, interface, or responses specifications) populates a data class object (DataMethod, DataVariables, DataInterface, or DataResponses) and appends the object to a linked list (methodList, variablesList, interfaceList, or responsesList). The strategy_kwhandler is the exception to this, since the restriction of only allowing one strategy specification means there’s no need for a DataStrategy class or a strategyList (instead, strategy attributes are members of ProblemDescDB). For information on modifying the input parsing procedures, refer to Dakota/docs/spec-change_instructions.txt

6.77.2 Member Function Documentation

6.77.2.1 void ProblemDescDB::manage_inputs (int argc, char ** argv, CommandLineHandler & cmdlinehandler)

parses the input file and populates the problem description database. This version reads from the dakota input filename passed with the "-input" option on the DAKOTA command line.

Manage command line inputs using the CommandLineHandler class and parse the input file using the Input Deck Reader (IDR) parsing system. IDR populates the ProblemDescDB object with the input file data.

6.77.2.2 void ProblemDescDB::manage_inputs (const char * dakota_input_file)

parses the input file and populates the problem description database. This version reads from the dakota input filename passed in.

Parse the input file using the Input Deck Reader (IDR) parsing system. IDR populates the ProblemDescDB object with the input file data.

6.77.2.3 void ProblemDescDB::set_db_model_type (const DakotaString & model_type) [inline]

set the model type.

Used to avoid recursion in DakotaModel::get_model() by a sub model when get_string("method.model-type") is not reset by a sub iterator. Note: if more needs of this type arise, could add set_<type> member functions to parallel the existing get_<type> member functions.

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

- ProblemDescDB.H
- ProblemDescDB.C
6.78 RespSurf Class Reference

Derived approximation class for polynomial regression.

Inheritance diagram for RespSurf:

```
DakotaApproximation
    RespSurf
```

Public Methods

- `RespSurf` (const `ProblemDescDB`& `problem`db, const `size_t`& `num_acv`)
  
  Constructor.

- `~RespSurf` ()
  
  Destructor.

Protected Methods

- void `findCoefficients` ()

  Least squares fit to data using a singular value decomposition.

- int `required_samples` ()

  Return the minimum number of samples required to build the derived class approximation type in `numVars` dimensions.

- const DakotaRealVector & `approximationCoefficients` ()

  Return the coefficient array computed by `findCoefficients()`.

- Real `getValue` (const DakotaRealVector & `x`)

  Retrieve the approximate function value for a given parameter vector.

- const DakotaRealVector & `getGradient` (const DakotaRealVector & `x`)

  Retrieve the approximate function gradient for a given parameter vector.

Private Attributes

- int `numCoeffs`

  Number of coefficients used by the polynomial model.

- DakotaRealVector `polyCoeffs`
vector of polynomial coefficients.

- int polyOrder
  flag to indicate a linear (value = 1), quadratic (value = 2), or cubic (value = 3) polynomial model.

### 6.78.1 Detailed Description

Derived approximation class for polynomial regression.

The RespSurf class computes a linear, quadratic, or cubic polynomial fit to data. The polynomial has either \(n+1\) (linear case), \((n+1)(n+2)/2\) (quadratic case), or \((n^3+6n^2+11n+6)/6\) (cubic case) coefficients for \(n\) variables. A least squares estimation of the polynomial coefficients is performed using LAPACK’s linear least squares subroutine DGELSS which uses a singular value decomposition method.

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

- RespSurf.H
- RespSurf.C
6.79  rSQPOptimizer Class Reference

Inheritance diagram for rSQPOptimizer::

```
<table>
<thead>
<tr>
<th>Dakotalterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>DakotaOptLeastSq</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
</tr>
<tr>
<td>rSQPOptimizer</td>
</tr>
</tbody>
</table>
```

Public Methods

- `rSQPOptimizer (DakotaModel &model)`
- `~rSQPOptimizer ()`
- `int num_objectives () const`
- `const DakotaRealVector & lin_ineq_lb () const`
- `const DakotaRealVector & lin_ineq_ub () const`
- `const DakotaRealVector & nonlin_ineq_lb () const`
- `const DakotaRealVector & nonlin_ineq_ub () const`
- `const DakotaRealVector & lin_eq_targ () const`
- `const DakotaRealVector & nonlin_eq_targ () const`
- `const DakotaRealMatrix & lin_eq_jac () const`
- `const DakotaRealMatrix & nonlin_eq_jac () const`

Overridden from DakotaOptimizer

- `void find optimum ()`

  *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

Private Attributes

- `DakotaModel * model_`
- `NLPInterfacePack::NLPDakota nlp_`

6.79.1 Detailed Description

Wrapper class for the rSQP++ optimization library.

The rSQPOptimizer class provides a wrapper for rSQP++, a C++ sequential quadratic programming library written by Roscoe Bartlett. rSQP++ can currently be used in NAND mode, although use of its SAND
mode for reduced-space SQP is planned. rSQPOptimizer uses a NLPDakota object to perform the function evaluations.

The user input mappings will ultimately include: max_iterations, convergence_tolerance, output_verbosity.

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

- rSQPOptimizer.H
- rSQPOptimizer.C
6.80 SGOPTApplication Class Reference

Maps the evaluation functions used by SGOPT algorithms to the DAKOTA evaluation functions.

Public Methods

- `SGOPTApplication (DakotaModel &model, DakotaResponse multiobj_mod_ptr)(const DakotaResponse &), int type)`
  constructor.

- `~SGOPTApplication ()`
  destructor.

- `int DoEval (OptPoint &pt, OptResponse *response, int synch_flag)`
  launch a function evaluation either synchronously or asynchronously.

- `int synchronize ()`
  blocking retrieval of all pending jobs.

- `int next_eval (int &id)`
  nonblocking query and retrieval of a job if completed.

- `void dakota_asynch_flag (const bool &asynch_flag)`
  set dakotaModelAsynchFlag.

Private Methods

- `void copy (const DakotaResponse &, OptResponse &)`
  copy data from a DakotaResponse object to an SGOPT OptResponse object.

Private Attributes

- `DakotaModel & userDefinedModel`
  reference to the SGOPTOptimizer's model passed in the constructor.

- `DakotaIntArray activeSetVector`
  copy/conversion of the SGOPT request vector.

- `bool dakotaModelAsynchFlag`
  a flag for asynchronous DAKOTA evaluations.

- `DakotaResponseList dakotaResponseList`
  list of DAKOTA responses returned by synchronize_nowait().
6.80.1 Detailed Description

Maps the evaluation functions used by SGOPT algorithms to the DAKOTA evaluation functions.

SGOPTApplication is a DAKOTA class that is derived from SGOPT’s AppInterface hierarchy. It redefines a variety of virtual SGOPT functions to use the corresponding DAKOTA functions. This is a more flexible algorithm library interfacing approach than can be obtained with the function pointer approaches used by NPSOLOptimizer and SNLLOptimizer.

6.80.2 Member Function Documentation

6.80.2.1 int SGOPTApplication::DoEval (OptPoint & pt, OptResponse * prob_response, int synch_flag)

launch a function evaluation either synchronously or asynchronously.

Converts SGOPT variables and request vector to DAKOTA variables and active set vector, performs a DAKOTA function evaluation with synchronization governed by synch_flag, and then copies the DakotaResponse data to the SGOPT response (synchronous) or bookkeeps the SGOPT response object (asynchronous).

6.80.2.2 int SGOPTApplication::synchronize ()

blocking retrieval of all pending jobs.

Blocking synchronize of asynchronous DAKOTA jobs followed by conversion of the DakotaResponse objects to SGOPT response objects.

6.80.2.3 int SGOPTApplication::next_eval (int & id)

nonblocking query and retrieval of a job if completed.

Nonblocking job retrieval. Finds a completion (if available), populates the SGOPT response, and sets id to the completed job’s id. Else set id = -1.
6.80.2.4  void SGOPTApplication::dakota_asynch_flag (const bool & asynch_flag)  [inline]

set dakotaModelAsynchFlag.

This function is needed to publish the iterator’s asynchFlag at run time (asynchFlag not available at con-
struction).

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

- SGOPTApplication.H
- SGOPTApplication.C
6.81 SGOPTOptimizer Class Reference

Wrapper class for the SGOPT optimization library.

Inheritance diagram for SGOPTOptimizer:

```
SGOPTOptimizer
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
DakotaOptimizer
|     |
|     |
DakotaOptLeastSq
|     |
|     |
DakotaIterator
|     |
|     |
SGOPTOptimizer
```

**Public Methods**

- `SGOPTOptimizer (DakotaModel &model)`
  
  constructor.

- `~SGOPTOptimizer ()`
  
  destructor.

- `void findOptimum ()`
  
  Performs the iterations to determine the optimal solution.

**Private Methods**

- `void setMethodOptions ()`
  
  sets options for the methods based on user specifications.

**Private Attributes**

- `DakotaString exploratoryMoves`
  
  user input for desired pattern search algorithm variant.

- `bool discreteAppFlag`
  
  convenience flag for integer vs. real applications.

- `PM_LCG * linConGenerator`
  
  Pointer to random number generator.
6.81 SGOPTOptimizer Class Reference

- **BaseOptimizer** * baseOptimizer
  
  Pointer to SGOPT base optimizer object.

- **AppInterface** * sgoptApplication
  
  Pointer to SGOPTApplication object.

- **RealOptProblem** * realProblem
  
  Pointer to RealOptProblem object.

- **IntOptProblem** * intProblem
  
  Pointer to IntOptProblem object.

- **PGAreal** * pGArealOptimizer
  
  Pointer to PGAreal object.

- **PGAint** * pGAintOptimizer
  
  Pointer to PGAint object.

- **EPSA** * ePSAOptimizer
  
  Pointer to EPSA object.

- **PatternSearch** * patternSearchOptimizer
  
  Pointer to PatternSearch object.

- **APPSopt** * aPPSoptimizer
  
  Pointer to APPSopt object.

- **SWOpt** * sWOptimizer
  
  Pointer to SWOpt object.

- **sMCreal** * sMCrealOptimizer
  
  Pointer to sMCreal object.

### 6.81.1 Detailed Description

Wrapper class for the SGOPT optimization library.

The SGOPTOptimizer class provides a wrapper for SGOPT, a Sandia-developed C++ optimization library of genetic algorithms, pattern search methods, and other nongradient-based techniques. It uses an SGOPTApplication object to perform the function evaluations.

The user input mappings are as follows: max_iterations, max_function_evaluations, convergence_tolerance, solution_accuracy and max_cpu_time are mapped into SGOPT’s max_iters, max_neval, ftol, accuracy, and max_time data attributes. An output setting of verbose is passed to SGOPT’s set_output() function and a setting of debug activates output of method initialization and sets the SGOPT debug attribute to 10000. SGOPT methods assume asynchronous operations whenever the algorithm has independent evaluations which can be performed simultaneously (implicit parallelism). Therefore, parallel configuration is not mapped into the method, rather it is used in SGOPTApplication to control whether or not an asynchronous evaluation request from the method is honored by the model (exception: pattern search exploratory moves is set to best_all for parallel function evaluations). Refer to [Hart, W.E., 1997] for additional information on SGOPT objects and controls.
6.81.2 Constructor & Destructor Documentation

6.81.2.1 SGOPTOptimizer::SGOPTOptimizer (DakotaModel & model)

constructor.
The constructor allocates the objects and populates the class member pointer attributes.

6.81.2.2 SGOPTOptimizer::~SGOPTOptimizer ()

destructor.
The destructor deallocates the class member pointer attributes.

6.81.3 Member Function Documentation

6.81.3.1 void SGOPTOptimizer::find_optimum () [virtual]

Performs the iterations to determine the optimal solution.
find_optimum redefines the DakotaOptimizer virtual function to perform the optimization using SGOPT. It
first sets up the problem data, then executes minimize() on the SGOPT algorithm, and finally catalogues
the results.
Implements DakotaOptimizer.

6.81.3.2 void SGOPTOptimizer::set_method_options () [private]

sets options for the methods based on user specifications.
set_method_options propagates DAKOTA user input to the appropriate SGOPT objects.

6.81.4 Member Data Documentation

6.81.4.1 AppInterface* SGOPTOptimizer::sgoptApplication [private]

pointer to the SGOPTApplication object.
SGOPTApplication is a DAKOTA class derived from the SGOPT AppInterface class. It redefines the
virtual SGOPT evaluation functions to use DAKOTA evaluation functions.
The documentation for this class was generated from the following files:

- SGOPTOptimizer.H
- SGOPTOptimizer.C
6.82 SingleMethodStrategy Class Reference

Simple fall-through strategy for running a single iterator on a single model.

Inheritance diagram for SingleMethodStrategy::

```
DakotaStrategy

SingleMethodStrategy
```

**Public Methods**

- `SingleMethodStrategy (ProblemDescDB &problem_db)`
  - *constructor.*

- `~SingleMethodStrategy ()`
  - *destructor.*

- `void run_strategy ()`
  - *Perform the strategy by executing selectedIterator on userDefinedModel.*

- `const DakotaVariables & strategy_variable_results () const`
  - *return the final solution from selectedIterator (variables).*

- `const DakotaResponse & strategy_response_results () const`
  - *return the final solution from selectedIterator (response).*

**Private Attributes**

- `DakotaModel userDefinedModel`
  - *the model to be iterated.*

- `DakotaIterator selectedIterator`
  - *the iterator.*

6.82.1 Detailed Description

Simple fall-through strategy for running a single iterator on a single model.

This strategy executes a single iterator on a single model. Since it does not provide coordination for multiple iterators and models, it can considered to be a “fall-through” strategy in that it allows control to fall through immediately to the iterator.

The documentation for this class was generated from the following files:
- SingleMethodStrategy.H
- SingleMethodStrategy.C
6.83  SingleModel Class Reference

Derived model class which utilizes a single interface to map variables into responses.

Inheritance diagram for SingleModel:

```
DakotaModel
  SingleModel
```

Public Methods

- **SingleModel** (ProblemDescDB &problem_db)
  
  *constructor.*

- **~SingleModel** ()
  
  *destructor.*

- **void derived\_compute\_response** (const DakotaIntArray &asv)
  
  *portion of compute\_response() specific to SingleModel (invokes a synchronous map() on userDefinedInterface).*

- **void derived\_asynch\_compute\_response** (const DakotaIntArray &asv)
  
  *portion of asynch\_compute\_response() specific to SingleModel (invokes an asynchronous map() on userDefinedInterface).*

- **const DakotaResponseArray & derived\_synchronize** ()
  
  *portion of synchronize() specific to SingleModel (invokes synch() on userDefinedInterface).*

- **const DakotaResponseList & derived\_synchronize\_nowait** ()
  
  *portion of synchronize\_nowait() specific to SingleModel (invokes synch\_nowait() on userDefinedInterface).*

- **DakotaString local\_eval\_synchronization** ()
  
  *return userDefinedInterface synchronization setting.*

- **const DakotaIntList & synchronize\_nowait\_completions** ()
  
  *return completion id’s matching response list from synchronize\_nowait (request forwarded to userDefinedInterface).*

- **bool derived\_master\_overload** () const
  
  *flag which prevents overloading the master with a multiprocessor evaluation (request forwarded to userDefinedInterface).*

- **void derived\_init\_communicators** (const DakotaIntArray &message\_lengths, const int &max\_iterator\_concurrency)
portion of init_communicators() specific to SingleModel (request forwarded to userDefinedInterface).

- **void free_communicators ()**
  deallocate communicator partitions for the SingleModel (request forwarded to userDefinedInterface).

- **void serve ()**
  Service job requests received from the master. Completes when a termination message is received from stop_servers() (request forwarded to userDefinedInterface).

- **void stop_servers ()**
  executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (request forwarded to userDefinedInterface).

- **int total_eval_counter () const**
  return the total evaluation count for the SingleModel (request forwarded to userDefinedInterface).

- **int new_eval_counter () const**
  return the new evaluation count for the SingleModel (request forwarded to userDefinedInterface).

**Private Attributes**

- **DakotaInterface userDefinedInterface**
  the interface used for mapping variables to responses.

**6.83.1 Detailed Description**

Derived model class which utilizes a single interface to map variables into responses.

The SingleModel class is the simplest of the derived model classes. It provides the capabilities the old DakotaModel class, prior to the development of layered and nested model extensions. The derived response computation and synchronization functions utilize a single interface to perform the function evaluations.

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

- SingleModel.H
- SingleModel.C
6.84  SNLLBase Class Reference

Base class for OPT++ optimization and least squares methods.

Inheritance diagram for SNLLBase::

```
SNLLBase
  ▼
   ▼
SNLLLeastSq     SNLLOptimizer
```

Public Methods

- **SNLLBase ()**
  
  *default constructor.*

- **SNLLBase (DakotaModel &model)**
  
  *standard constructor.*

- **~SNLLBase ()**
  
  *destructor.*

Protected Methods

- **void pre_instantiate (const DakotaString &merit_fn, bool bound_constr_flag, const int &num_constr)**
  
  *convenience function for setting OPT++ options prior to the method instantiation.*

- **void post_instantiate (const int &num_cv, bool vendor_num_grad_flag, const DakotaString &finite_diff_type, const Real &fdss, const int &maxiter, const int &maxvals, const Real &conv_tol, const Real &grad_tol, const int &maxstep, bool bound_constr_flag, const int &num_constr, bool debug_output, OptimizeClass *the_optimizer, NLP0 *nlf_objective, FDNLF1 *fd_nlf1, FDNLF1 *fd_nlf1_con)**
  
  *convenience function for setting OPT++ options after the method instantiation.*

- **void pre_run (NLP0 *nlf_objective, NLP *nlp_constraint, const DakotaModel &model, bool bound_constr_flag, const DakotaRealVector &nln_i neoliberal_bnds, const DakotaRealVector &nln_i neoliberal_u_bnds, const DakotaRealVector &nln_i neoliberal_targets)**
  
  *convenience function for OPT++ configuration prior to the method invocation.*

- **void post_run (NLP0 *nlf_objective, DakotaVariables &best_vars)**
  
  *convenience function for setting OPT++ options after the method instantiations.*
Static Protected Methods

- void init_fn (int n, ColumnVector &x)
  An initialization mechanism provided by OPT++ (not currently used).

- void copy_con_vals (const DakotaRealVector &local_fn_vals, ColumnVector &g, const size_t &offset)
  convenience function for copying local_fn_vals to g; used by constraint evaluator functions.

- void copy_con_vals (const ColumnVector &g, DakotaRealVector &local_fn_vals, const size_t &offset)
  convenience function for copying g to local_fn_vals; used in final solution logging.

- void copy_con_grad (const DakotaRealMatrix &local_fn_grads, Matrix &grad_g, const size_t &offset)
  convenience function for copying local_fn_grads to grad_g; used by constraint evaluator functions.

- void copy_con_hess (const DakotaRealMatrixArray &local_fn_hessians, OptppArray< SymmetricMatrix > &hess_g, const size_t &offset)
  convenience function for copying local_fn_hessians to hess_g; used by constraint evaluator functions.

Protected Attributes

- DakotaString searchMethod
  value_based_line_search, gradient_based_line_search, trust_region, or tr_pds.

- SearchStrategy s
  enum: LineSearch, TrustRegion, or TrustPDS.

- MeritFcn mfcn
  enum: NormFmu, ArgaezTapia, or VanShanno.

- bool constantASVFlag
  flags a user selection of active_set_vector == constant. By mapping this into mode override, reliance on duplicate detection can be avoided.

Static Protected Attributes

- EvalType staticLastFnEvalLocn
  an enum used to track whether an nlf evaluator or a constraint evaluator was the last location of a function evaluation.

- bool staticDebugOutput
  static copy of debugOutput.

- bool staticModeOverrideFlag
  flags OPT++ mode override (for combining value, gradient, and Hessian requests).
6.84 SNLLBase Class Reference

- **int staticConEvalMode**
  
  *static copy of mode from constraint evaluators.*

- **DakotaRealVector staticConEvalVars**
  
  *static copy of variables from constraint evaluators.*

- **int staticNumNonlinearEqConstraints**
  
  *number of nonlinear equality constraints.*

- **int staticNumNonlinearIneqConstraints**
  
  *number of nonlinear inequality constraints.*

### 6.84.1 Detailed Description

Base class for OPT++ optimization and least squares methods.

The SNLLBase class provides a common base class for **SNLLOptimizer** and **SNLLLeastSq**, both of which are wrappers for OPT++, a C++ optimization library from the Computational Sciences and Mathematics Research (CSMR) department at Sandia's Livermore CA site.

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

- **SNLLBase.H**
- **SNLLBase.C**
6.85 SNLLLeastSq Class Reference

Wrapper class for the OPT++ optimization library.

Inheritance diagram for SNLLLeastSq::

```
SNLLLeastSq
  ↓
DakotaLeastSq
  ↓
DakotaOptLeastSq
  ↓
DakotaIterator
  ↓
SNLLBase
  ↓
SNLLLeastSq
```

Public Methods

- **SNLLLeastSq** (DakotaModel &model)
  
  *constructor.*

- ~**SNLLLeastSq** ()
  
  *destructor.*

- void **minimize_residuals** ()
  
  *Performs the iterations to determine the least squares solution.*

Static Private Methods

- void **nlf2_evaluator_gn** (int mode, int n, const ColumnVector &x, Real &f, ColumnVector &grad_f, SymmetricMatrix &hess_f, int &result_mode)
  
  *objective function evaluator function which obtains values and gradients for least square terms and computes objective function value, gradient, and Hessian using the Gauss-Newton approximation.*

- void **constraint2_evaluator_gn** (int mode, int n, const ColumnVector &x, ColumnVector &g, Matrix &grad_g, OptppArray< SymmetricMatrix > &hess_g, int &result_mode)
  
  *constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ Gauss-Newton methods. While it does not employ the Gauss-Newton approximation, it is distinct from constraint2_evaluator() due to its need to anticipate the required modes for the least squares terms.*

Private Attributes

- NLP0 * nlfObjective
  
  *objective NLF base class pointer.*
6.85 SNLLLeastSq Class Reference

- NLP0 * nlfConstraint
  
  constraint NLF base class pointer.

- NLP * nlpConstraint
  
  constraint NLP pointer.

- NLF2 * nlf2
  
  pointer to objective NLF for full Newton optimizers.

- NLF2 * nlf2Con
  
  pointer to constraint NLF for full Newton optimizers.

- OptimizeClass * theOptimizer
  
  optimizer base class pointer.

- OptNewton * optnewton
  
  Newton optimizer pointer.

- OptBCNewton * optbcnewton
  
  Bound constrained Newton optimizer pointer.

- OptNIPS * optnips
  
  NIPS optimizer pointer.

6.85.1 Detailed Description

Wrapper class for the OPT++ optimization library.

The SNLLLeastSq class provides a wrapper for OPT++, a C++ optimization library of nonlinear programming and pattern search techniques from the Computational Sciences and Mathematics Research (CSMR) department at Sandia’s Livermore CA site. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or static as well. To isolate the effect of these static requirements from the rest of the iterator hierarchy, static copies are made of many non-static attributes inherited from above.

The user input mappings are as follows: max_iterations, max_function_evaluations, convergence_tolerance, max_step, gradient_tolerance, search_method, and search_scheme_size are set using OPT++’s setMaxIter(), setMaxFeval(), setFcnnTol(), setMaxStep(), setGradTol(), setSearchStrategy(), and setSSS() member functions, respectively; output verbosity is used to toggle OPT++’s debug mode using the setDebug() member function. Internal to OPT++, there are 3 search strategies, while the DAKOTA search_method specification supports 4 (value_based_line_search, gradient_based_line_search, trust_region, or tr_pds). The difference stems from the "is_expensive" flag in OPT++. If the search strategy is LineSearch and "is_expensive" is turned on, then the value_based_line_search is used. Otherwise (the "is_expensive" default is off), the algorithm will use the gradient_based_line_search. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/VendorOptimizers/opt++ directory for information on OPT++ class member functions.
6.85.2 Member Function Documentation

6.85.2.1 void SNLLLeastSq::nlf2_evaluator_gn (int mode, int n, const ColumnVector & x, Real & f, ColumnVector & grad_f, SymmetricMatrix & hess_f, int & result_mode) [static, private]

objective function evaluator function which obtains values and gradients for least square terms and computes objective function value, gradient, and Hessian using the Gauss-Newton approximation.

This nlf2 evaluator function is used for the Gauss-Newton method in order to exploit the special structure of the nonlinear least squares problem. Here, \( f(x) = \sum (T_i - T_{bar,i})^2 \) and DakotaResponse is made up of residual functions and their gradients along with any nonlinear constraints. The objective function and its gradient vector and Hessian matrix are computed directly from the residual functions and their derivatives (which are returned from the DakotaResponse object).

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

- SNLLLeastSq.H
- SNLLLeastSq.C
6.86 SNLLOptimizer Class Reference

Wrapper class for the OPT++ optimization library.

Inheritance diagram for SNLLOptimizer::

```
SNLLOptimizer
   |       |
   |       |
DakotaOptLeastSq
   |       |
DakotaOptimizer
   |       |
SNLLBase
   |       |
SNLLOptimizer
```

Public Methods

- **SNLLOptimizer (DakotaModel &model)**
  
  *constructor.*

- **~SNLLOptimizer ()**
  
  *destructor.*

- **void find_optimum ()**
  
  *Performs the iterations to determine the optimal solution.*

Static Private Methods

- **void nlf0_evaluator (int n, const ColumnVector &x, Real &f, int &result_mode)**
  
  *objective function evaluator function for OPT++ methods which require only function values.*

- **void nlf1_evaluator (int mode, int n, const ColumnVector &x, Real &f, ColumnVector &grad_f, int &result_mode)**
  
  *objective function evaluator function which provides function values and gradients to OPT++ methods.*

- **void nlf2_evaluator (int mode, int n, const ColumnVector &x, Real &f, ColumnVector &grad_f, SymmetricMatrix &hess_f, int &result_mode)**
  
  *objective function evaluator function which provides function values, gradients, and Hessians to OPT++ methods.*

- **void constraint0_evaluator (int n, const ColumnVector &x, ColumnVector &g, int &result_mode)**
  
  *constraint evaluator function for OPT++ methods which require only constraint values.*
void constraint1_evaluator (int mode, int n, const ColumnVector &x, ColumnVector &g, Matrix &grad_g, int &result_mode)

constraint evaluator function which provides constraint values and gradients to OPT++ methods.

void constraint2_evaluator (int mode, int n, const ColumnVector &x, ColumnVector &g, Matrix &grad_g, OptppArray< SymmetricMatrix > &hess_g, int &result_mode)

constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ methods.

Private Attributes

- NLP0 * nlfObjective
  objective NLF base class pointer.

- NLP0 * nlfConstraint
  constraint NLF base class pointer.

- NLP * nlpConstraint
  constraint NLP pointer.

- NLF0 * nlf0
  pointer to objective NLF for nongradient optimizers.

- NLF1 * nlf1
  pointer to objective NLF for (analytic) gradient-based optimizers.

- NLF1 * nlf1Con
  pointer to constraint NLF for (analytic) gradient-based optimizers.

- FDNLF1 * fdnlf1
  pointer to objective NLF for (finite diff) gradient-based optimizers.

- FDNLF1 * fdnlf1Con
  pointer to constraint NLF for (finite diff) gradient-based optimizers.

- NLF2 * nlf2
  pointer to objective NLF for full Newton optimizers.

- NLF2 * nlf2Con
  pointer to constraint NLF for full Newton optimizers.

- OptimizeClass * theOptimizer
  optimizer base class pointer.

- OptPDS * optpds
  PDS optimizer pointer.

- OptCG * optcg
  CG optimizer pointer.
6.86 SNLLOptimizer Class Reference

- OptNewton * optnewton
  Newton optimizer pointer.

- OptQNewton * optqnewton
  Quasi-Newton optimizer pointer.

- OptFDNewton * optfdnewton
  Finite Difference Newton optimizer pointer.

- OptBCNewton * optbcnewton
  Bound constrained Newton optimizer pointer.

- OptBCQNewton * optbcqnewton
  Bnd constrained Quasi-Newton optimizer ptr.

- OptBCFDNewton * optbcfdnewton
  Bnd constrained FD-Newton optimizer ptr.

- OptNIPS * optnips
  NIPS optimizer pointer.

- OptQNIPS * optqnips
  Quasi-Newton NIPS optimizer pointer.

- OptFDNIPS * optfdnips
  Finite Difference NIPS optimizer pointer.

6.86.1 Detailed Description

Wrapper class for the OPT++ optimization library.

The SNLLOptimizer class provides a wrapper for OPT++, a C++ optimization library of nonlinear programming and pattern search techniques from the Computational Sciences and Mathematics Research (CSMR) department at Sandia’s Livermore CA site. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or static as well. To isolate the effect of these static requirements from the rest of the iterator hierarchy, static copies are made of many non-static attributes inherited from above.

The user input mappings are as follows: max_iterations, max_function_evaluations, convergence_tolerance, max_step, gradient_tolerance, search_method, and search_scheme_size are set using OPT++’s setMaxIter(), setMaxFeval(), setFcnTol(), setMaxStep(), setGradTol(), setSearchStrategy(), and setSSS() member functions, respectively; output verbosity is used to toggle OPT++’s debug mode using the setDebug() member function. Internal to OPT++, there are 3 search strategies, while the DAKOTA search_method specification supports 4 (value_based_line_search, gradient_based_line_search, trust_region, or tr_pds). The difference stems from the "is_expensive" flag in OPT++. If the search strategy is LineSearch and "is_expensive" is turned on, then the value_based_line_search is used. Otherwise (the "is_expensive" default is off), the algorithm will use the gradient_based_line_search. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/VendorOptimizers/opt++ directory for information on OPT++ class member functions.
6.86.2 Member Function Documentation

6.86.2.1 void SNLLOptimizer::nlf0_evaluator (int n, const ColumnVector & x, Real & f, int & result_mode) [static, private]

objective function evaluator function for OPT++ methods which require only function values.
For use when DAKOTA computes f and gradients are not directly available. This is used by nongradient-based optimizers such as PDS and by gradient-based optimizers in vendor numerical gradient mode (opt++’s internal finite difference routine is used).

6.86.2.2 void SNLLOptimizer::nlf1_evaluator (int mode, int n, const ColumnVector & x, Real & f, ColumnVector & grad_f, int & result_mode) [static, private]

objective function evaluator function which provides function values and gradients to OPT++ methods.
For use when DAKOTA computes f and df/dX (regardless of gradientType). Vendor numerical gradient case is handled by nlf0_evaluator.

6.86.2.3 void SNLLOptimizer::nlf2_evaluator (int mode, int n, const ColumnVector & x, Real & f, ColumnVector & grad_f, SymmetricMatrix & hess_f, int & result_mode) [static, private]

objective function evaluator function which provides function values, gradients, and Hessians to OPT++ methods.
For use when DAKOTA receives f, df/dX, & \(\frac{d^2f}{dx^2}\) from the ApplicationInterface (analytic only). Finite differencing does not make sense for a full Newton approach, since lack of analytic gradients & Hessian should dictate the use of quasi-newton or fd-newton. Thus, there is no fdnlf2_evaluator for use with full Newton approaches, since it is preferable to use quasi-newton or fd-newton with nlf1. Gauss-Newton does not fit this model; it uses nlf2_evaluator_gn instead of nlf2_evaluator.

6.86.2.4 void SNLLOptimizer::constraint0_evaluator (int n, const ColumnVector & x, ColumnVector & g, int & result_mode) [static, private]

constraint evaluator function for OPT++ methods which require only constraint values.
For use when DAKOTA computes g and gradients are not directly available. This is used by nongradient-based optimizers and by gradient-based optimizers in vendor numerical gradient mode (opt++’s internal finite difference routine is used).

6.86.2.5 void SNLLOptimizer::constraint1_evaluator (int mode, int n, const ColumnVector & x, ColumnVector & g, Matrix & grad_g, int & result_mode) [static, private]

constraint evaluator function which provides constraint values and gradients to OPT++ methods.
For use when DAKOTA computes g and dg/dX (regardless of gradientType). Vendor numerical gradient case is handled by constraint0_evaluator.
void SNLLOptimizer::constraint2_evaluate (int mode, int n, const ColumnVector & x, ColumnVector & g, Matrix & grad_g, OptppArray< SymmetricMatrix > & hess_g, int & result_mode) [static, private]

constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ methods. For use when DAKOTA computes g, dg/dX, & d^2g/dx^2 (analytic only).

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

- SNLLOptimizer.H
- SNLLOptimizer.C
6.87 SOLBase Class Reference

Base class for Stanford SOL software.

Inheritance diagram for SOLBase::

```
SOLBase
 |   |
 v   v
NLSSOLLeastSq NPSOLOptimizer
```

Public Methods

- `SOLBase()`
  *default constructor.*

- `SOLBase(DakotaModel &model)`
  *standard constructor.*

- `~SOLBase()`
  *destructor.*

Protected Methods

- `void allocate_arrays(const int &num_cv, const size_t &num_nln_ineq_con, const size_t &num_nln_eq_con, const size_t &num_nln_eq_con, const size_t &num_lin_ineq_con, const size_t &num_lin_eq_con, const DakotaRealMatrix &lin_ineq_coeffs, const DakotaRealMatrix &lin_eq_coeffs)`
  *Allocates miscellaneous arrays for the SOL algorithms.*

- `void deallocate_arrays()`
  *Deallocates memory previously allocated by allocate_arrays().*

- `void allocate_workspace(const int &num_cv, const int &num_nln_con, const int &num_lin_con, const int &num_lsq)`
  *Allocates real and integer workspaces for the SOL algorithms.*

- `void set_options(bool speculative_flag, bool vendor_num_grad_flag, bool verbose_output, const int &verify_lev, const Real &fn_prec, const Real &linesrch_tol, const int &max_iter, const Real &constr_tol, const Real &conv_tol, const DakotaString &grad_type, const Real &fdss)`
  *Sets SOL method options using calls to npoptn2.*

- `void augment_bounds(DakotaRealVector &augmented_lbnds, DakotaRealVector &augmented_ubnds, const DakotaRealVector &linineq_lbnds, const DakotaRealVector &linineq_ubnds, const DakotaRealVector &lineq_lbnds, const DakotaRealVector &lineq_ubnds, const DakotaRealVector &nlnineq_lbnds, const DakotaRealVector &nlnineq_ubnds)`
  *Sets SOL bound constraints.*
augments variable bounds with linear and nonlinear constraint bounds.

Static Protected Methods

- void `constraint_eval` (int &mode, int &ncnln, int &n, int &nrowj, int *needc, Real *x, Real *c, Real *jac, int &nstate)

  `CONFUN` in `NPSOL` manual: computes the values and first derivatives of the nonlinear constraint functions.

Protected Attributes

- int `realWorkSpaceSize`
  size of `realWorkSpace`.

- int `intWorkSpaceSize`
  size of `intWorkSpace`.

- DakotaRealArray `realWorkSpace`
  real work space for `NPSOL/NLSSOL`.

- DakotaIntArray `intWorkSpace`
  int work space for `NPSOL/NLSSOL`.

- int `nlnConstraintArraySize`
  used for non-zero array sizing (nonlinear constraints).

- int `linConstraintArraySize`
  used for non-zero array sizing (linear constraints).

- DakotaRealArray `cLambda`
  `CLAMBDA` from `NPSOL` manual: Langrange multipliers.

- DakotaIntArray `constraintState`
  `ISTATE` from `NPSOL` manual: constraint status.

- int `informResult`
  `INFORM` from `NPSOL` manual: optimization status on exit.

- int `numberIterations`
  `ITER` from `NPSOL` manual: number of (major) iterations performed.

- int `boundsArraySize`
  length of augmented bounds arrays (variable bounds plus linear and nonlinear constraint bounds).

- double * `linConstraintMatrixF77`
• double * upperFactorHessianF77
  \([R]\) matrix from NPSOL manual: upper Cholesky factor of the Hessian of the Lagrangian.

• double * constraintJacMatrixF77
  \([CJAC]\) matrix from NPSOL manual: nonlinear constraint Jacobian.

### Static Protected Attributes

• int fnEvalCntr
  counter for testing against staticMaxFnEvals.

• size_t staticConstrOffset
  used in constraint_eval() to bridge NLSSOLLeastSq::numLeastSqTerms and NPSOLOptimizer::numObjectiveFunctions.

• DakotaModel & staticSOLModel = dummy model
  static reference to userDefinedModel used in constraint_eval().

• int staticMaxFnEvals
  static copy of DakotaIterator::maxFunctionEvals.

• int staticVendorNumericalGradFlag
  static copy of DakotaOptimizer::vendorNumericalGradFlag.

### 6.87.1 Detailed Description

Base class for Stanford SOL software.

The SOLBase class provides a common base class for NPSOLOptimizer and NLSSOLLeastSq, both of which are Fortran 77 sequential quadratic programming algorithms from Stanford University marketed by Stanford Business Associates.

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

• SOLBase.H
• SOLBase.C
6.88 SortCompare Class Template Reference

Public Methods

- `SortCompare (bool(*func)(const T &, const T &))`  
  Constructor that defines the pointer to function.

- bool `operator()` (const T &p1, const T &p2) const  
  The operator() must be defined. Calls the defined sortFunction.

Private Attributes

- bool(*)(const T &, const T &) `sortFunction`  
  Pointer to test function.

6.88.1 Detailed Description

template<class T> class SortCompare<T>

Internal functor used in the sort algorithm to sort using a specified compare method. The class holds a pointer to the sort function.

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

- DakotaList.H
6.89 SurrBasedOptStrategy Class Reference

Strategy for provably-convergent surrogate-based optimization.

Inheritance diagram for SurrBasedOptStrategy::

```
DakotaStrategy
```

```
SurrBasedOptStrategy
```

Public Methods

- **SurrBasedOptStrategy** (ProblemDescDB &problem_db)
  
  Constructor.

- **~SurrBasedOptStrategy**()
  
  Destructor.

- **void run_strategy**()
  
  Performs the surrogate-based optimization strategy by optimizing local, global, or hierarchical surrogates over a series of trust regions.

- **const DakotaVariables & strategy_variable_results**() const
  
  Return the SBO final solution (variables).

- **const DakotaResponse & strategy_response_results**() const
  
  Return the SBO final solution (response).

Private Methods

- **Real compute_penalty_function** (const DakotaRealVector &fn_vals)
  
  Compute a penalty function from a set of function values.

- **void hard_convergence_check** (const DakotaResponse &response_center_truth)
  
  Check for hard convergence (zero gradient of penalty function).

- **void soft_convergence_check** (const DakotaRealVector &c_vars_center, const DakotaRealVector &c_vars_star, const DakotaResponse &response_center_truth, const DakotaResponse &response_center_approx, const DakotaResponse &response_star_truth, const DakotaResponse &response_star_approx)
  
  Check for soft convergence (diminishing returns).
Private Attributes

- **DakotaModel approximateModel**
  the surrogate model (a LayeredModel object).

- **DakotaIterator selectedIterator**
  the optimizer used on approximateModel.

- **DakotaRealVector trustRegionSize**
  the size of the current trust region is computed by multiplying the trustRegionFactor value by the difference between the lower and upper bounds.

- **Real trustRegionFactor**
  the trust region factor is used to compute the size of the trust region – it is a percentage, e.g. for trustRegionFactor = 0.1 the actual size of the trust region will be 10% of the global bounds (upper bound - lower bound for each design variable).

- **Real minTrustRegionFactor**
  a soft convergence control: stop SBO when the trust region factor is reduced below the value of minTrustRegionFactor.

- **Real convergenceTol**
  the optimizer convergence tolerance; used in several SBO hard and soft convergence checks.

- **Real constraintTol**
  a tolerance specifying the distance from a constraint boundary that is allowed before an active constraint is considered to be a violated constraint (only violated constraints are used in penalty function computations).

- **Real penaltyParameter**
  the penalization factor for violated constraints used in penalty function calculations; increases exponentially with iteration count.

- **Real trRatioContractValue**
  trust region ratio min value: contract tr if ratio below this value.

- **Real trRatioExpandValue**
  trust region ratio sufficient value: expand tr if ratio above this value.

- **Real gammaContract**
  trust region contraction factor.

- **Real gammaExpand**
  trust region expansion factor.

- **Real gammaNoChange**
  factor for maintaining the current trust region size (normally 1.0).

- **int iterMax**
  maximum number of SBO iterations.

- **short convergenceFlag**
code indicating satisfaction of hard or soft convergence conditions.

- **int** `numFns`  
  number of response functions.

- **int** `numVars`  
  number of active continuous variables.

- **short** `softConvCount`  
  number of consecutive candidate point rejections. If the count reaches `softConvLimit`, stop SBO.

- **short** `softConvLimit`  
  the limit on consecutive candidate point rejections. If exceeded by `softConvCount`, stop SBO.

- **bool** `gradientFlag`  
  flags the use of gradients throughout the SBO process.

- **bool** `correctionFlag`  
  flags the use of surrogate correction techniques at the center of each trust region.

- **bool** `globalApproxFlag`  
  flags the use of a global data fit surrogate (rsm, ann, mars, kriging).

- **bool** `localApproxFlag`  
  flags the use of a local data fit surrogate (Taylor series).

- **bool** `hierarchApproxFlag`  
  flags the use of a hierarchical surrogate.

- **bool** `newCenterFlag`  
  flags the acceptance of a candidate point and the existence of a new trust region center.

- **bool** `daceCenterPtFlag`  
  flags the availability of the center point in the DACE evaluations for global approximations (CCD, Box-Behnken).

- **size_t** `numObjFns`  
  number of objective functions.

- **size_t** `numNonlinIneqConstr`  
  number of nonlinear inequality constraints.

- **size_t** `numNonlinEqConstr`  
  number of nonlinear equality constraints.

- **DakotaRealVector** `multiObjWts`  
  vector of multiobjective weights.

- **DakotaRealVector** `nonlinIneqLowerBnds`  
  vector of nonlinear inequality constraint lower bounds.
- DakotaRealVector nonlinIneqUpperBnds
  <em>vector of nonlinear inequality constraint upper bounds.</em>

- DakotaRealVector nonlinEqTargets
  <em>vector of nonlinear equality constraint targets.</em>

- DakotaVariables bestVariables
  <em>best variables found in SBO.</em>

- DakotaResponse bestResponses
  <em>best responses found in SBO.</em>

### 6.89.1 Detailed Description

Strategy for provably-convergent surrogate-based optimization.

This strategy uses a LayeredModel to perform optimization based on local, global, or hierarchical surrogates. It achieves provable convergence through the use of a sequence of trust regions and the application of surrogate corrections at the trust region centers.

### 6.89.2 Member Function Documentation

#### 6.89.2.1 void SurrBasedOptStrategy::run_strategy () [virtual]

Performs the surrogate-based optimization strategy by optimizing local, global, or hierarchical surrogates over a series of trust regions.

Trust region-based strategy to perform surrogate-based optimization in subregions (trust regions) of the parameter space. The optimizer operates on approximations in lieu of the more expensive simulation-based response functions. The size of the trust region is varied according to the goodness of the agreement between the approximations and the true response functions.

Reimplemented from DakotaStrategy.

#### 6.89.2.2 Real SurrBasedOptStrategy::compute_penalty_function (const DakotaRealVector & fn_vals) [private]

compute a penalty function from a set of function values.

The penalty function computation applies a penalty multiplier to any constraint violations and adds this to the objective function. This implementation supports multiple objectives, equality constraints, and 2-sided inequalities. A negative constraintTol can be used to provide a push-back into the feasible region.

#### 6.89.2.3 void SurrBasedOptStrategy::hard_convergence_check (const DakotaResponse & response_center_truth) [private]

check for hard convergence (zero gradient of penalty function).
The hard convergence check computes the 2-norm of the gradient of the penalty function at the trust region center and signals convergence if the 2-norm is close to zero.

6.89.2.4 **void SurrBasedOptStrategy::soft_convergence_check** (const DakotaRealVector & c_vars_center, const DakotaRealVector & c_vars_star, const DakotaResponse & response_center_truth, const DakotaResponse & response_center_approx, const DakotaResponse & response_star_truth, const DakotaResponse & response_star_approx) [private]

check for soft convergence (diminishing returns).

Compute soft convergence metrics (trust region ratio, number of consecutive failures, min trust region size, etc.) and use them to assess whether the convergence rate has decreased to a point where the process should be terminated (diminishing returns).

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

- SurrBasedOptStrategy.H
- SurrBasedOptStrategy.C
6.90  SurrLayeredModel Class Reference

Derived model class within the layered model branch for managing data fit surrogates (global and local).

Inheritance diagram for SurrLayeredModel::

```
  DakotaModel
   |   |
   v   v
LayeredModel
   |   |
   v   v
SurrLayeredModel
```

Public Methods

- **SurrLayeredModel (ProblemDescDB &problem_db)**
  constructor.

- **~SurrLayeredModel ()**
  destructor.

Protected Methods

- void **derived_compute_response** (const DakotaIntArray &asv)
  portion of `compute_response()` specific to SurrLayeredModel.

- void **derived_asynch_compute_response** (const DakotaIntArray &asv)
  portion of `asynch_compute_response()` specific to SurrLayeredModel.

- const DakotaResponseArray & **derived_synchronize** ()
  portion of `synchronize()` specific to SurrLayeredModel.

- const DakotaResponseList & **derived_synchronize_nowait** ()
  portion of `synchronize_nowait()` specific to SurrLayeredModel.

- bool **derived_master_overload** () const
  flag which prevents overloading the master with a multiprocessor evaluation.

- DakotaModel & **subordinate_model** ()
  returns `actualModel` to `SurrBasedOptStrategy`.

- DakotaIterator & **subordinate_iterator** ()
  return `daceIterator` to `SurrBasedOptStrategy`. 
• int maximum_concurrency () const
  return the maximum concurrency available for actualModel computations during global approximation builds.

• void build_approximation ()
  Builds the local/multipoint/global approximation using dacelIterator/actualModel.

• const DakotaIntList & synchronize_nowait_completions ()
  return completion id’s matching response list from synchronize_nowait (request forwarded to approxInterface).

• void update_approximation (const DakotaRealVector &x_star, const DakotaResponse &response_star)
  Adds a point to a global approximation (request forwarded to approxInterface).

• const DakotaRealVectorArray & approximation_coefficients ()
  return the approximation coefficients from each DakotaApproximation (request forwarded to approxInterface).

• int total_eval_counter () const
  return the total evaluation count for the SurrLayeredModel (request forwarded to approxInterface).

• int new_eval_counter () const
  return the new evaluation count for the SurrLayeredModel (request forwarded to approxInterface).

• void derived_init_communicators (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  portion of init_communicators() specific to SurrLayeredModel.

• void free_communicators ()
  deallocate communicator partitions for the SurrLayeredModel (request forwarded to actualModel).

• void serve ()
  Service job requests received from the master. Completes when a termination message is received from stop_servers() (request forwarded to actualModel).

• void stop_servers ()
  Executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (request forwarded to actualModel).

Private Attributes

• DakotaInterface approxInterface
  manages the building and subsequent evaluation of the approximations (required for both global and local).

• DakotaString actualInterfacePointer
  string identifier for the actual interface from the local approximation specification (required for local); used to build actualModel for local approximations.
**DakotaString daceMethodPointer**

String identifier for the dace method from the global approximation specification; used in building daceIterator and actualModel for global approximations (optional for global since restart data may also be used).

**DakotaModel actualModel**

The truth model which provides evaluations for building the surrogate (optional for global since restart data may also be used, required for local).

**DakotaIterator daceIterator**

Selects parameter sets on which to evaluate actualModel in order to generate the necessary data for building global approximations (optional for global since restart data may also be used).

### 6.90.1 Detailed Description

Derived model class within the layered model branch for managing data fit surrogates (global and local).

The SurrLayeredModel class manages global or local approximations (surrogates that involve data fits) that are used in place of an expensive model. The class contains an approxInterface (required for both global and local) which manages the approximate function evaluations, an actualModel (optional for global, required for local) which provides truth evaluations for building the surrogate, and a daceIterator (optional for global, not used for local) which selects parameter sets on which to evaluate actualModel in order to generate the necessary data for building global approximations.

### 6.90.2 Member Function Documentation

#### 6.90.2.1 void SurrLayeredModel::derived_compute_response (const DakotaIntArray & asv)

[protected, virtual]

Portion of compute_response() specific to SurrLayeredModel.

Build the approximation (if needed), evaluate the approximate response using approxInterface, and, if correction is active, correct the results.

Reimplemented from DakotaModel.

#### 6.90.2.2 void SurrLayeredModel::derived_asynch_compute_response (const DakotaIntArray & asv)

[protected, virtual]

Portion of asynch_compute_response() specific to SurrLayeredModel.

Build the approximation (if needed) and evaluate the approximate response using approxInterface in a quasi-asynchronous approach (approximationInterface::map() performs the map synchronously and bookkeeps the results for return in derived_synchronize() below).

Reimplemented from DakotaModel.
6.90.2.3 const DakotaResponseArray & SurrLayeredModel::derived_synchronize ()
[protected, virtual]

portion of synchronize() specific to SurrLayeredModel.
Retrieve quasi-asynchronous evaluations from approxInterface and, if correction is active, apply correction to each response in the array.
Reimplemented from DakotaModel.

6.90.2.4 const DakotaResponseList & SurrLayeredModel::derived_synchronize_nowait ()
[protected, virtual]

portion of synchronize_nowait() specific to SurrLayeredModel.
Retrieve quasi-asynchronous evaluations from approxInterface and, if correction is active, apply correction to each response in the list.
Reimplemented from DakotaModel.

6.90.2.5 bool SurrLayeredModel::derived_master_overload () const [inline, protected, virtual]

flag which prevents overloading the master with a multiprocessor evaluation.
compute_response calls never overload the master since there is no parallelism in the use of approxInterface.
Reimplemented from DakotaModel.

6.90.2.6 int SurrLayeredModel::maximum_concurrency () const [protected, virtual]

return the maximum concurrency available for actualModel computations during global approximation builds.
Return the greater of the dace samples user-specification or the min_samples approximation requirement. min_samples does not have to account for reuse_samples, since this will vary (assume 0).
Reimplemented from DakotaModel.

6.90.2.7 void SurrLayeredModel::build_approximation () [protected, virtual]

Builds the local/multipoint/global approximation using daceIterator/actualModel.
Build either a global approximation using daceIterator or a local approximation using actualModel. Selection triggers on actualInterfacePointer (required specification for local approximation interfaces, not used in global specification).
Reimplemented from DakotaModel.

6.90.2.8 void SurrLayeredModel::derived_init_communicators (const DakotaIntArray & message_sizes, const int & max_iterator_concurrency) [inline, protected, virtual]

portion of init_communicators() specific to SurrLayeredModel.
asynchronous flags need to be initialized for the sub-models. In addition, max\_iterator\_concurrency is the outer level iterator concurrency, not the DACE concurrency that actualModel will see, and recomputing the message\_lengths on the sub-model is probably not a bad idea either. Therefore, recompute everything on actualModel using init\_communicators.

Reimplemented from DakotaModel.

### 6.90.3 Member Data Documentation

#### 6.90.3.1 DakotaString SurrLayeredModel::actualInterfacePointer [private]

string identifier for the actual interface from the local approximation specification (required for local); used to build actualModel for local approximations. Specification is used only for local approximations, since the dace\_method\_pointer in the global approximation specification is responsible for identifying all actualModel components.

#### 6.90.3.2 DakotaModel SurrLayeredModel::actualModel [private]

the truth model which provides evaluations for building the surrogate (optional for global since restart data may also be used, required for local).

There are no restrictions on actualModel in the global case, so arbitrary nestings are possible. In the local case, model\_type must be set to "single" to avoid recursion on SurrLayeredModel, since there is no additional method specification.

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

- SurrLayeredModel.H
- SurrLayeredModel.C
6.91  SurrogateDataPoint Class Reference

Simple container class encapsulating basic parameter and response data for defining a "truth" data point.

Public Methods

- **SurrogateDataPoint ()**
  default constructor.

- **SurrogateDataPoint (const DakotaRealVector &x, const Real &f, const DakotaRealVector &grad_f)**
  standard constructor.

- **SurrogateDataPoint (const SurrogateDataPoint &sdp)**
  copy constructor.

- **~SurrogateDataPoint ()**
  destructor.

- **int operator==(const SurrogateDataPoint &sdp) const**
  equality operator.

- **SurrogateDataPoint & operator=(const SurrogateDataPoint &sdp)**
  assignment operator.

Public Attributes

- **DakotaRealVector continuousVars**
  continuous variables.

- **Real responseFn**
  truth response function value.

- **DakotaRealVector responseGrad**
  truth response function gradient.

6.91.1  Detailed Description

Simple container class encapsulating basic parameter and response data for defining a "truth" data point.

A list of these data points is contained in each DakotaApproximation instance (DakotaApproximation::currentPoints) and provides the data to build the approximation. Data is public to avoid maintaining set/get functions, but is still encapsulated within DakotaApproximation since DakotaApproximation::currentPoints is protected (a similar model is used with with Data class objects contained in ProblemDescDB and with ParallelismLevel objects contained in ParallelLibrary).
The documentation for this class was generated from the following file:

- DakotaApproximation.H
6.92 SysCallAnalysisCode Class Reference

Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls.

Inheritance diagram for SysCallAnalysisCode:

```
AnalysisCode
    ↓
SysCallAnalysisCode
```

**Public Methods**

- **SysCallAnalysisCode** (const ProblemDescDB &problem_db)
  - *constructor.*

- **~SysCallAnalysisCode** ()
  - *destructor.*

- void **spawn_evaluation** (const bool block_flag)
  - *spawn a complete function evaluation.*

- void **spawn_input_filter** (const bool block_flag)
  - *spawn the input filter portion of a function evaluation.*

- void **spawn_analysis** (const int &analysis_id, const bool block_flag)
  - *spawn a single analysis as part of a function evaluation.*

- void **spawn_output_filter** (const bool block_flag)
  - *spawn the output filter portion of a function evaluation.*

- const DakotaString & **command_usage** () const
  - *return commandUsage.*

**Private Attributes**

- DakotaString **commandUsage**
  - *optional command usage string for supporting nonstandard command syntax (supported only by SysCall analysis codes).*
6.92 SysCallAnalysisCode Class Reference

6.92.1 Detailed Description

Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls.
SysCallAnalysisCode creates separate simulation processes using the C system() command. It utilizes CommandShell to manage shell syntax and asynchronous invocations.

6.92.2 Member Function Documentation

6.92.2.1 void SysCallAnalysisCode::spawn_evaluation (const bool block flag)

spawn a complete function evaluation.
Put the SysCallAnalysisCode to the shell using either the default syntax or specified commandUsage syntax. This function is used when all portions of the function evaluation (i.e., all analysis drivers) are executed on the local processor.

6.92.2.2 void SysCallAnalysisCode::spawn_input_filter (const bool block flag)

spawn the input filter portion of a function evaluation.
Put the input filter to the shell. This function is used when multiple analysis drivers are spread between processors. No need to check for a Null input filter, as this is checked externally. Use of nonblocking shells is supported in this fn, although its use is currently prevented externally.

6.92.2.3 void SysCallAnalysisCode::spawn_analysis (const int & analysis id, const bool block flag)

spawn a single analysis as part of a function evaluation.
Put a single analysis to the shell using the default syntax (no commandUsage support for analyses). This function is used when multiple analysis drivers are spread between processors. Use of nonblocking shells is supported in this fn, although its use is currently prevented externally.

6.92.2.4 void SysCallAnalysisCode::spawn_output_filter (const bool block flag)

spawn the output filter portion of a function evaluation.
Put the output filter to the shell. This function is used when multiple analysis drivers are spread between processors. No need to check for a Null output filter, as this is checked externally. Use of nonblocking shells is supported in this fn, although its use is currently prevented externally.

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

- SysCallAnalysisCode.H
- SysCallAnalysisCode.C
6.93 SysCallApplicInterface Class Reference

Derived application interface class which spawns simulation codes using system calls.

Inheritance diagram for SysCallApplicInterface::

```
DakotaInterface
  |
  v
ApplicationInterface
  |
  v
SysCallApplicInterface
```

Public Methods

- **SysCallApplicInterface** (const ProblemDescDB &problem_db, const size_t &num_fns)
  
  *constructor.*

- **~SysCallApplicInterface** ()
  
  *destructor.*

- void **derived_map** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int fn_eval_id)
  
  Called by `map()` and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- void **derived_map_asynch** (const ParamResponsePair &pair)
  
  Called by `map()` and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- void **derived_synch** (DakotaPRPList &prp_list)
  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- void **derived_synch_nowait** (DakotaPRPList &prp_list)
  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- int **derived_synchronous_local_analysis** (const int &analysis_id)
  
  Execute a particular analysis (identified by `analysis_id`) synchronously on the local processor. Used for the derived class specifics within `ApplicationInterface::serve_analyses_synch()`.
6.93 SysCallApplicInterface Class Reference

Private Methods

- **void spawn_application (const bool block_flag)**
  
  *Spawn the application by managing the input filter, analysis drivers, and output filter.* Called from `derived_map() & derived_map_synch()`.

- **void derived_synch_kernel (DakotaPRPList &prp_list)**
  
  *Convenience function for common code between derived_synch() & derived_synch_nowait().*

- **bool system_call_file_test (const DakotaString &root_file)**
  
  *Detect completion of a function evaluation through existence of the necessary results file(s).*

Private Attributes

- **SysCallAnalysisCode sysCallSimulator**
  
  *SysCallAnalysisCode provides convenience functions for passing the input filter, the analysis drivers, and the output filter to a CommandShell in various combinations.*

- **DakotaIntList sysCallList**
  
  *List of function evaluation id's for active asynchronous system call evaluations.*

- **DakotaIntList failIdList**
  
  *List of function evaluation id's for tracking response file read failures.*

- **DakotaIntList failCountList**
  
  *List containing the number of response read failures for each function evaluation identified in failIdList.*

6.93.1 Detailed Description

Derived application interface class which spawns simulation codes using system calls.

SysCallApplicInterface uses a **SysCallAnalysisCode** object for performing simulation invocations.

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

- SysCallApplicInterface.H
- SysCallApplicInterface.C

---

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2002
6.94 TaylorSurf Class Reference

Derived approximation class for 1st order Taylor series (local approximation).

Inheritance diagram for TaylorSurf:

```
DakotaApproximation
    TaylorSurf
```

**Public Methods**

- **TaylorSurf** (const ProblemDescDB &problem_db, const size_t &num_acv)
  constructor.
- **~TaylorSurf** ()
  destructor.

**Protected Methods**

- void **find_coefficients** ()
  calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

- int **required_samples** ()
  return the minimum number of samples required to build the derived class approximation type in numVars dimensions.

- Real **get_value** (const DakotaRealVector &x)
  retrieve the approximate function value for a given parameter vector.

- const DakotaRealVector & **get_gradient** (const DakotaRealVector &x)
  retrieve the approximate function gradient for a given parameter vector.

6.94.1 Detailed Description

Derived approximation class for 1st order Taylor series (local approximation).

The TaylorSurf class provides a local approximation based on data from a single point in parameter space. It uses a first order Taylor series expansion: \( f(x) = f(x_c) + \text{grad}(x_c) \cdot (x - x_c) \)

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

- TaylorSurf.H
- TaylorSurf.C
6.95 VariablesUtil Class Reference

Utility class for the DakotaVariables and DakotaVarConstraints hierarchies which provides convenience functions for variable vectors and label arrays for combining design, uncertain, and state variable types and merging continuous and discrete variable domains.

Inheritance diagram for VariablesUtil:

```
VariablesUtil
  |______________
  |             |
  |             |
  |             |
  |             |
  |             |
  AllMergedVarConstraints
  |______________
  |             |
  |             |
  |             |
  |             |
  AllMergedVariables
  |______________
  |             |
  |             |
  |             |
  |             |
  AllVarConstraints
  |______________
  |             |
  |             |
  |             |
  |             |
  AllVariables
  |______________
  |             |
  |             |
  |             |
  |             |
  FundamentalVarConstraints
  |______________
  |             |
  |             |
  |             |
  |             |
  FundamentalVariables
  |______________
  |             |
  |             |
  |             |
  |             |
  MergedVarConstraints
  |______________
  |             |
  |             |
  |             |
  |             |
  MergedVariables
```

**Public Methods**

- *VariablesUtil ()*
  
  *constructor.*

- *~VariablesUtil ()*
  
  *destructor.*

**Protected Methods**

- *void update merged (const DakotaRealVector &c_array, const DakotaIntVector &d_array, DakotaRealVector &m_array)*
  
  *combine a continuous array and a discrete array into a single continuous array through promotion of integers to reals (merged view).*
• void \texttt{update\_all\_continuous} (const DakotaRealVector \&c1\_array, const DakotaRealVector \&c2\_array, const DakotaRealVector \&c3\_array, DakotaRealVector \&all\_array) const
  \textit{combine 3 continuous arrays (design, uncertain, state) into a single continuous array (all view).}

• void \texttt{update\_all\_discrete} (const DakotaIntVector \&d1\_array, const DakotaIntVector \&d2\_array, DakotaIntVector \&all\_array) const
  \textit{combine 2 discrete arrays (design, state) into a single discrete array (all view).}

• void \texttt{update\_labels} (const DakotaStringArray \&l1\_array, const DakotaStringArray \&l2\_array, DakotaStringArray \&all\_array) const
  \textit{combine 2 label arrays into a single label array (merged or all views).}

• void \texttt{update\_labels} (const DakotaStringArray \&l1\_array, const DakotaStringArray \&l2\_array, const DakotaStringArray \&l3\_array, DakotaStringArray \&all\_array) const
  \textit{combine 3 label arrays (design, uncertain, state) into a single label array (all view).}

\section{6.95.1 Detailed Description}
Utility class for the DakotaVariables and DakotaVarConstraints hierarchies which provides convenience functions for variable vectors and label arrays for combining design, uncertain, and state variable types and merging continuous and discrete variable domains.

Derived classes within the DakotaVariables and DakotaVarConstraints hierarchies use multiple inheritance to inherit these utilities.

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

• VariablesUtil.H
Chapter 7

DAKOTA File Documentation

7.1 keywordtable.C File Reference

file containing keywords for the strategy, method, variables, interface, and responses input specifications from dakota.input.spec.

Variables

- const struct KeywordHandler idrKeywordTable []

  Initialize the keyword table as a vector of KeywordHandler structures (KeywordHandler declared in idr-keyword.h). A null KeywordHandler structure signifies the end of the keyword table.

7.1.1 Detailed Description

file containing keywords for the strategy, method, variables, interface, and responses input specifications from dakota.input.spec.
7.2 main.C File Reference

file containing the main program for DAKOTA.

Functions

- int main (int argc, char *argv[])

  *The main DAKOTA program.*

Variables

- int write_precision = 10

  *used in ostream data output functions.*

7.2.1 Detailed Description

file containing the main program for DAKOTA.

7.2.2 Function Documentation

7.2.2.1 int main (int argc, char * argv[])

The main DAKOTA program.

Manage command line inputs, input files, restart file(s), output streams, and top level parallel iterator communicators. Instantiate the DakotaStrategy and invoke its run_strategy() virtual function.
7.3 restart_util.C File Reference

file containing the DAKOTA restart utility main program.

Functions

- void print_restart (int argc, char **argv, DakotaString print_dest)
  
  print a restart file.

- void print_restart_tabular (int argc, char **argv, DakotaString print_dest)
  
  print a restart file (tabular format).

- void read_neutral (int argc, char **argv)
  
  read a restart file (neutral file format).

- void repair_restart (int argc, char **argv, DakotaString identifier_type)
  
  repair a restart file by removing corrupted evaluations.

- void concatenate_restart (int argc, char **argv)
  
  concatenate multiple restart files.

- int main (int argc, char *argv[])
  
  The main program for the DAKOTA restart utility.

Variables

- int write_precision = 16
  
  used in ostream output fns. DAKOTA’s main.C sets this to 10, however "print" outputs parameters in full precision (16 digits for double).

7.3.1 Detailed Description

file containing the DAKOTA restart utility main program.

7.3.2 Function Documentation
7.3.2.1 void print_restartTime (int argc, char ** argv, DakotaString print_dest)

print a restart file.

Usage: "dakota restart_util print dakota.rst"
"dakota restart_util to_neutral dakota.rst dakota.neu"

Prints all evals. in full precision to either stdout or a neutral file. The former is useful for ensuring that
duplicate detection is successful in a restarted run (e.g., starting a new method from the previous best), and
the latter is used for translating binary files between platforms.

7.3.2.2 void print_restartTime_tabular (int argc, char ** argv, DakotaString print_dest)

print a restart file (tabular format).

Usage: "dakota restart_util to_pdb dakota.rst dakota.pdb"
"dakota restart_util to_tabular dakota.rst dakota.txt"

Unrolls all data associated with a particular tag for all evaluations and then writes this data in a tabular
format (e.g., to a PDB database or MATLAB/TECPLLOT data file).

7.3.2.3 void read_neural (int argc, char ** argv)

read a restart file (neutral file format).

Usage: "dakota restart_util from_neural dakota.neu dakota.rst"

Reads evaluations from a neutral file. This is used for translating binary files between platforms.

7.3.2.4 void repair_restartTime (int argc, char ** argv, DakotaString identifier_type)

repair a restart file by removing corrupted evaluations.

Usage: "dakota restart_util remove 0.0 dakota_old.rst dakota_new.rst"
"dakota restart_util remove_ids 2 7 13 dakota_old.rst dakota_new.rst"

Repairs a restart file by removing corrupted evaluations. The identifier for evaluation removal can be either
a double precision number (all evaluations having a matching response function value are removed) or a
list of integers (all evaluations with matching evaluation ids are removed).

7.3.2.5 void concatenate_restartTime (int argc, char ** argv)

concatenate multiple restart files.

Usage: "dakota restart_util cat dakota\_1.rst ... dakota\_n.rst dakota\_new.rst"

Combines multiple restart files into a single restart database.

7.3.2.6 int main (int argc, char * argv[])

The main program for the DAKOTA restart utility.

Parse command line inputs and invoke the appropriate utility function (print_restartTime(),
print_restartTime_tabular(), read_neural(), repair_restartTime(), or concatenate_restartTime()).
Chapter 8

Interfacing with DAKOTA as a Library

8.1 Introduction

Some users may be interested in linking the DAKOTA toolkit into another application for use as an algorithm library. While this is not the primary use model for DAKOTA, certain facilities are in place to allow this type of integration.

As part of the normal DAKOTA build process, a libdakota.a is created and a copy of it is placed in Dakota/lib. This library contains all source files from Dakota/src excepting the main.C and restart_util.C main programs. This library may be linked with another application through inclusion of -ldakota on the link line. Library and header paths may also be specified using the -L and -I compiler options. Depending on the configuration used when building this library, other libraries for the vendor optimizers and vendor packages will also be needed to resolve DAKOTA symbols for DOT, NPSOL, OPT++, SGOPT, LHS, Epetra, etc. Copies of these libraries are also placed in Dakota/lib. An XML specification of library names and paths is also available in Dakota/dependency.

Warning:
While users are free to interface DAKOTA as a library within other software applications for their own internal use, the GNU GPL license stipulates that any application linked with DAKOTA in this way defines a "derivative work" and can only be distributed externally under the same GNU GPL open source license. Refer to http://www.gnu.org/licenses/gpl.html or contact the DAKOTA team for additional information.

Attention:
The use of DAKOTA as an algorithm library should be distinguished from the linking of simulations within DAKOTA using the direct application interface (see DirectFnApplicInterface). In the former, DAKOTA is providing algorithm services to another software application, and in the latter, a linked simulation is providing analysis services to DAKOTA.

The procedure for linking DAKOTA within another application is most easily explained with reference to main.C. The basic steps of executing DAKOTA include management of command line inputs and input files (ProblemDescDB::manage_inputs()), managing restart files and output streams (ParallelLibrary::manage_outputs Restart()), initializing and freeing top level parallel iterator communicators (ParallelLibrary::init_iterator_communicators() and ParallelLibrary::free_iterator_communicators), and instantiating the DakotaStrategy and running it (DakotaStrategy::run_strategy()). When using DAKOTA
as an algorithm library, these same basic operations must still be performed, although the syntax will be different from that in `main.C`. In particular, `main.C` can pass command line attributes to `ProblemDescDB::manage_inputs()` and `ParallelLibrary::manage_outputs_restart()`, whereas in an algorithm library approach, command line information will not in general be accessible.

To replace information previously obtained from the command line, overloaded forms of these functions have been developed in which the required information is passed through the parameter lists. In the case of managing restart files and output streams, the call to

```c
parallel_lib.manage_outputs_restart(cmd_line_handler);
```

should be replaced with its overloaded form

```c
parallel_lib.manage_outputs_restart(std_output_filename,
   std_error_filename, read_restart_filename, write_restart_filename,
   restart_evals);
```

where file names for standard output and error and restart read and write as well as the integer number of restart evaluations are passed through the parameter list rather than read from the command line of the main DAKOTA program. The definition of these attributes is performed elsewhere in the parent application (e.g., specified in the parent application input file or GUI).

With respect to modifying `ProblemDescDB::manage_inputs()`, the two following sections describe different approaches to populating data within DAKOTA's problem description database. It is this database from which all DAKOTA objects draw data upon instantiation.

### 8.2 Problem database populated through input file parsing

The simplest approach to linking an application with the DAKOTA library is to rely on DAKOTA's normal parsing system to populate DAKOTA's problem database (`ProblemDescDB`) through the reading of an input file. The disadvantage to this approach is the requirement for an additional input file beyond those already required by the parent application.

In this approach, the call to

```c
problem_db.manage_inputs(argc, argv, cmd_line_handler);
```

should be replaced with its overloaded form

```c
problem_db.manage_inputs(dakota_input_file);
```

where the file name for the DAKOTA input is passed through the parameter list rather than read from the command line of the main DAKOTA program. Again, the definition of the DAKOTA input file name is performed elsewhere in the parent application (e.g., specified in the parent application input file or GUI).

### 8.3 Problem database populated through external means

This approach is more involved than the previous approach, but it allows the application to publish all needed data to DAKOTA's database directly, thereby eliminating the need for the parsing of a separate
DAKOTA input file. In this case, `ProblemDescDB::manage_inputs()` is not called. Rather, `DataStrategy`, `DataMethod`, `DataVariables`, `DataInterface`, and `DataResponses` objects must be instantiated and populated with the desired problem data. These objects are then published to the problem database using `ProblemDescDB::insert_node()`, e.g.:

```cpp
// instantiate the data object
DataMethod data_method;

// set the attributes within the data object
data_method.methodName = "nond_sampling";
...

// publish the data object to the ProblemDescDB
problem_db.insert_node(data_method);
```

The data objects are populated with their default values upon instantiation, so only the non-default values need to be specified. Refer to the `DataStrategy`, `DataMethod`, `DataVariables`, `DataInterface`, and `DataResponses` class documentation and source code for lists of attributes and their defaults.

The default strategy is `single_method`, which runs a single iterator on a single model, so it is not necessary to instantiate and publish a `DataStrategy` object if coordination of multiple iterators and models is not required. Rather, instantiation and insertion of a single `DataMethod`, `DataVariables`, `DataInterface`, and `DataResponses` object is sufficient for basic DAKOTA capabilities.

Once the data objects have been published to the `ProblemDescDB` object, a call to

```cpp
problem_db.check_input();
```

will perform basic database error checking.

### 8.4 Performing an iterative study

With the `ProblemDescDB` object populated with problem data, the next step is to instantiate and run the strategy:

```cpp
// instantiate the strategy
DakotaStrategy selected_strategy(problem_db);

// run the strategy
selected_strategy.run_strategy();
```

### 8.5 Retrieving data after a run

After executing the strategy, final results can be obtained through the use of `DakotaStrategy::strategy_variable_results()` and `DakotaStrategy::strategy_response_results()`, e.g.:

```cpp
// retrieve the final parameter values
const DakotaVariables& vars = selected_strategy.strategy_variable_results();

// retrieve the final response values
const DakotaResponse& resp = selected_strategy.strategy_response_results();
```
In the case of optimization, the final design is returned, and in the case of uncertainty quantification, the final statistics are returned.

### 8.6 Summary

To utilize the DAKOTA library within a parent software application, the basic steps of *main.C* and the order of invocation of these steps should be mimicked from within the parent application. Of these steps, 

*ProblemDescDB::manage_inputs()* and *ParallelILibrary::manage_outputs_restart()* require the modifications described herein in order to perform in an environment without direct command line access and, potentially, without file parsing.

DAKOTA's library mode is a relatively new capability and feedback from the user community for making it more useful is welcome.
Chapter 9

Performing Function Evaluations

Performing function evaluations is one of the most critical functions of the DAKOTA software. It can also be one of the most complicated, as a variety of scheduling approaches and parallelism levels are supported. This complexity manifests itself in the code through a series of cascaded member functions, from the top level model evaluation functions, through various scheduling routines, to the low level details of performing a system call, fork, or direct function invocation. This section provides an overview of the primary classes and member functions involved.

9.1 Synchronous function evaluations

For a synchronous (i.e., blocking) mapping of parameters to responses, an iterator invokes DakotaModel::compute_response() to perform a function evaluation. This function is all that is seen from the iterator level, as underlying complexities are isolated. The binding of this top level function with lower level functions is as follows:

- DakotaModel::compute_response() utilizes DakotaModel::derived::compute_response() for portions of the response computation specific to derived model classes.
- DakotaModel::derived::compute_response() directly or indirectly invokes DakotaInterface::map().
- DakotaInterface::map() utilizes ApplicationInterface::derived::map() for portions of the mapping specific to derived application interface classes.

9.2 Asynchronous function evaluations

For an asynchronous (i.e., nonblocking) mapping of parameters to responses, an iterator invokes DakotaModel::asynch_compute_response() multiple times to queue asynchronous jobs and then invokes either DakotaModel::synchronize() or DakotaModel::synchronize_nowait() to schedule the queued jobs in blocking or nonblocking fashion. Again, these functions are all that is seen from the iterator level, as underlying complexities are isolated. The binding of these top level functions with lower level functions is as follows:

- DakotaModel::asynch_compute_response() utilizes DakotaModel::derived::asynch_compute_response() for portions of the response computation specific to derived model classes.
This derived model class function directly or indirectly invokes `DakotaInterface::map()` in asynchronous mode, which adds the job to a scheduling queue.

- `DakotaModel::synchronize()` or `DakotaModel::synchronize_nowait()` utilize `DakotaModel::derived_synchronize()` or `DakotaModel::derived_synchronize_nowait()` for portions of the scheduling process specific to derived model classes.

- These derived model class functions directly or indirectly invoke `DakotaInterface::synch()` or `DakotaInterface::synch_nowait()`.

- For application interfaces, these interface synchronization functions are responsible for performing evaluation scheduling in one of the following modes:
  
  - asynchronous local mode (using `ApplicationInterface::asynchronous_local_evaluations()` or `ApplicationInterface::asynchronous_local_evaluations_nowait()`)
  
  - message passing mode (using `ApplicationInterface::self_schedule_evaluations()` or `ApplicationInterface::static_schedule_evaluations()` on the iterator master and `ApplicationInterface::serve_evaluations_synch()` or `ApplicationInterface::serve_evaluations_peer()` on the servers)
  
  - hybrid mode (using `ApplicationInterface::self_schedule_evaluations()` or `ApplicationInterface::static_schedule_evaluations()` on the iterator master and `ApplicationInterface::serve_evaluations_asynch()` on the servers)

- These scheduling functions utilize `ApplicationInterface::derived_map()` and `ApplicationInterface::derived_map_asynch()` for portions of asynchronous job launching specific to derived application interface classes, as well as `ApplicationInterface::derived_synch()` and `ApplicationInterface::derived_synch_nowait()` for portions of job capturing specific to derived application interface classes.

### 9.3 Analyses within each function evaluation

The discussion above covers the parallelism level of concurrent function evaluations serving an iterator. For the parallelism level of concurrent analyses serving a function evaluation, similar schedulers are involved (`ForkApplicInterface::synchronous_local_analyses()`, `ForkApplicInterface::asynchronous_local_analyses()`, `ApplicationInterface::self_schedule_analyses()`, `ApplicationInterface::serve_analyses_synch()`, `ForkApplicInterface::serve_analyses_asynch()` to support synchronous local, asynchronous local, message passing, and hybrid modes. Not all of the schedulers are elevated to the `ApplicationInterface` level since the system call and direct function interfaces do not yet support nonblocking local analyses (and therefore support synchronous local and message passing modes, but not asynchronous local or hybrid modes). Fork interfaces, however, support all modes of analysis parallelism.
Chapter 10

Recommended Practices for DAKOTA Development

10.1 Introduction

Common code development practices can be extremely useful in multiple developer environments. Particular styles for code components lead to improved readability of the code and can provide important visual cues to other developers.

Much of this recommended practices document is borrowed from the CUBIT mesh generation project, which in turn borrows its recommended practices from other projects. As a result, C++ coding styles are fairly standard across a variety of Sandia software projects in the engineering and computational sciences.

10.2 Style Guidelines

Style guidelines involve the ability to discern at a glance the type and scope of a variable or function.

10.2.1 Class and variable styles

Class names should be composed of two or more descriptive words, with the first character of each word capitalized, e.g.:

    class ClassName;

Class member variables should be composed of two or more descriptive words, with the first character of the second and succeeding words capitalized, e.g.:

    double classMemberVariable;
Temporary (i.e. local) variables are lower case, with underscores separating words in a multiple word temporary variable, e.g.:

```c
int temporary_variable;
```

Constants (i.e. parameters) are upper case, with underscores separating words, e.g.:

```c
const double CONSTANT_VALUE;
```

### 10.2.2 Function styles

Function names are lower case, with underscores separating words, e.g.:

```c
int function_name();
```

There is no need to distinguish between member and non-member functions by style, as this distinction is usually clear by context. This style convention arose from the desire to have member functions which set and return the value of a private member variable, e.g.:

```c
int memberVariable;
void member_variable(int a) { // set
    memberVariable = a;
}
int member_variable() const { // get
    return memberVariable;
}
```

In cases where the data to be set or returned is more than a few bytes, it is highly desirable to employ const references to avoid unnecessary copying, e.g.:

```c
void continuous_variables(const DakotaRealVector& c_vars) { // set
    continuousVariables = c_vars;
}
const DakotaRealVector& continuous_variables() const { // get
    return continuousVariables;
}
```

Note that it is not necessary to always accept the returned data as a const reference. If it is desired to be able change this data, then accepting the result as a new variable will generate a copy, e.g.:

```c
const DakotaRealVector& c_vars = model.continuous_variables(); // reference to continuousVariables cannot be changed
DakotaRealVector c_vars = model.continuous_variables(); // local copy of continuousVariables can be changed
```

### 10.2.3 Miscellaneous

Appearance of typedefs to redefine or alias basic types is isolated to a few header files (`data_types.h`, `template_defs.h`), so that issues like program precision can be changed by changing a few lines of typedefs rather than many lines of code, e.g.:
typedef double Real;

**xemacs** is the preferred source code editor, as it has C++ modes for enhancing readability through color (turn on "Syntax highlighting"). Other helpful features include "Paren highlighting" for matching parentheses and the "New Frame" utility to have more than one window operating on the same set of files (note that this is still the same edit session, so all windows are synchronized with each other). Window width should be set to 80 internal columns, which can be accomplished by manual resizing, or preferably, using the following alias in your shell resource file (e.g., `.cshrc`):

```bash
alias xemacs "xemacs -g 81x63"
```

where an external width of 81 gives 80 columns internal to the window and the desired height of the window will vary depending on monitor size. This window width imposes a coding standard since you should avoid line wrapping by continuing anything over 80 columns onto the next line.

Indenting increments are 2 spaces per indent and comments are aligned with the code they describe, e.g.:

```c
void abort_handler(int code)
{
    int initialized = 0;
    MPI_Init(&initialized);
    if (initialized) {
        // comment aligned to block it describes
        int size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size>1)
            MPI_Abort(MPI_COMM_WORLD, code);
        else
            exit(code);
    }
    else
        exit(code);
}
```

Also, the continuation of a long command is indented 2 spaces, e.g.:

```c
const DakotaString& iterator_scheduling
    = problem_db.get_string("strategy.iterator_scheduling");
```

and similar lines are aligned for readability, e.g.:

```c
cout << "Numerical gradients using " << finiteDiffStepSize*100. << "." << finiteDiffType << " differences\n to be calculated by the " << methodSource << " finite difference routine." << endl;
```

Lastly, **if**def’s are not indented (to make use of syntax highlighting in **xemacs**).

### 10.3 File Naming Conventions

In addition to the style outlined above, the following file naming conventions have been established for the DAKOTA project.

File names for C++ classes should be identical to the class name defined by that file. Exceptions: in some cases it is convenient to maintain several closely related classes in a single file, in which case the
file name may reflect the top level class (e.g., DakotaResponse.C/.H files contain DakotaResponse and DakotaResponseRep classes) or some generalization of the set of classes (e.g., DakotaBinStream.C/.H files contain the DakotaBiStream/DakotaBoStream classes for binary input and binary output).

The type of file is determined by one of the four file name extensions listed below:

- **.H** A class header file ends in the suffix .H. The header file provides the class declaration. This file does not contain code for implementing the methods, except for the case of inline functions. Inline functions are to be placed at the bottom of the file with the keyword inline preceding the function name.
- **.C** A class implementation file ends in the suffix .C. An implementation file contains the definitions of the members of the class.
- **.h** A header file ends in the suffix .h. The header file contains information usually associated with procedures. Defined constants, data structures and function prototypes are typical elements of this file.
- **.c** A procedure file ends in the suffix .c. The procedure file contains the actual procedures.

### 10.4 Class Documentation Conventions

Class documentation uses the doxygen tool available from [http://www.doxygen.org](http://www.doxygen.org) and employs the JAVA-doc comment style. Brief comments appear in header files next to the attribute or function declaration. Detailed descriptions for functions should appear alongside their implementations (i.e., in the .C files for non-inlined, or in the headers next to the function definition for inlined). Detailed comments for a class or a class attribute must go in the header file as this is the only option.

**NOTE:** Previous class documentation utilities (class2frame and class2html) used the "//-" comment style and comment blocks such as this:

```plaintext
//- Class: DakotaModel
//- Description: The model to be iterated. Contains DakotaVariables, DakotaInterface, and DakotaResponse objects.
//- Owner: Mike Eldred
```

These tools are no longer used, so remaining comment blocks of this type are informational only and will not appear in the documentation generated by doxygen.
Chapter 11

Instructions for Modifying DAKOTA’s Input Specification

11.1 Modify dakota.input.spec

The master input specification resides in $DAKOTA/src/dakota.input.spec. As part of the Input Deck Reader (IDR) build process, a soft link to this file is created in $DAKOTA/VendorPackages/idr. The master input specification can be modified with the addition of new constructs using the following logical relationships:

- {} for required individual specifications
- () for required group specifications
- [] for optional individual specifications
- [[]] for optional group specifications
- | for "or" conditionals

These constructs can be used to define a variety of dependency relationships in the input specification. It is recommended that you review the existing specification and have an understanding of the constructs in use before attempting to add new constructs.

Warning:
- Do not skip this step. Attempts to modify the keywordtable.C and ProblemDescDB.C files in $DAKOTA/src without reference to the results of the code generator are very error-prone. Moreover, the input specification provides a reference to the allowable inputs of a particular executable and should be kept in sync with the parser files (modifying the parser files independent of the input specification creates, at a minimum, undocumented features).
- Since the Input Deck Reader (IDR) parser allows abbreviation of keywords, you must avoid adding a keyword that could be misinterpreted as an abbreviation for a different keyword within the same keyword handler (the term "keyword handler" refers to the strategy_khandler(), method_khandler(), variables_khandler(), interface_khandler(), and responses_khandler() member functions in the ProblemDescDB class). For example, adding the keyword "expansion" within the method specification would be a mistake if the keyword "expansion_factor" already was being used in this specification.
Since IDR input is order-independent, the same keyword may be reused multiple times in the specification if and only if the specification blocks are mutually exclusive. For example, method selections (e.g., dot_frcg, dot_bfgs) can reuse the same method setting keywords (e.g., optimization_type) since the method selection blocks are all separated by logical "or"s. If dot_frcg and dot_bfgs were not exclusive and could be specified at the same time, then association of the optimization_type setting with a particular method would be ambiguous. This is the reason why repeated specifications which are non-exclusive must be made unique, typically with a prepended identifier (e.g., cdv_initial_point, ddv_initial_point).

11.2 Rebuild IDR

cd $DAKOTA/VendorPackages/idr
make clean
make


11.3 Update keywordtable.C in $DAKOTA/src

Do not directly replace the keywordtable.C in $DAKOTA/src using the one from idr, as there are important differences in the kwhandler bindings. Rather, update the keywordtable.C in $DAKOTA/src using the one from idr as a reference. Once this step is completed, it is a good idea to verify the match by diff'ing the 2 files. The only differences should be in comments, includes, and kwhandler declarations.

11.4 Update ProblemDescDB.C in $DAKOTA/src

Find the keyword handler functions (e.g., variables kwhandler()) in $DAKOTA/VendorPackages/idr/ canonical_build_directory>/idr-gen-code.C and $DAKOTA/src/ProblemDescDB.C which correspond to your modifications to the input specification. The idr-gen-code.C file is the result of a code generator and contains skeleton constructs for extracting data from IDR. You will be copying over parts of this skeleton to ProblemDescDB.C and then adding code to populate attributes within Data class container objects.

11.4.1 Replace keyword handler declarations and counter loop

Rather than trying to update these line by line, it is recommended to delete the entire block starting with the keyword declarations and ending at the bottom of the keyword counter loop. The declarations assign -1 to keywords and look like this:
11.4 Update ProblemDescDB.C in `$DAKOTA/src`

Int cdv_descriptor = -1;
Int cdv_initial_point = -1;

They start after the line "Int cntr;". The keyword counter loop looks like this:

```c
for ( cntr=data_len; cntr--; ) {
    if ( idr_find_id( &cdv_descriptor, cntr,
        "cdv_descriptor", id_str, kw_str ) ) continue;
    ...
    if ( idr_find_id( &wuv_dist_upper_bounds, cntr,
        "wuv_dist_upper_bounds", id_str, kw_str ) ) continue;
}
```

Once the old keyword declarations and keyword counter loop have been deleted, replace them with the corresponding blocks from idr-gen-code.C containing the updated keyword declarations and counter loop.

### 11.4.2 Update keyword handler logic blocks

For the newly added or modified input specifications, copy the appropriate skeleton constructs from idr-gen-code.C and paste them into the corresponding location in ProblemDescDB.C.

The next step is to add code to these skeletons to set data attributes within the Data class object used by the keyword handler. At the top of the method, variables, interface, and responses keyword handlers, a Data class object is instantiated in order to store attributes, e.g.:

```c
DataMethod data_method;
```

and within the strategy keyword handler, the strategySpec data class object is used to store attributes. Each of these data class objects is a simple container class which contains the data from a single keyword handler invocation. Within each skeleton construct, you will extract data from the IDR data structures and then use this data to set the corresponding attribute within the Data class.

Integer, real, and string data are extracted using the `idata`, `rdata`, and `cdata` arrays provided by IDR. These arrays are indexed using a bracket operator with the keyword as an index.

Lists of integer and real data are extracted using the `idr_table` constructs provided by IDR. Unfortunately, IDR does not provide an `idr_table` for string data, so these extractions are more involved. Refer to existing `<LISTof>`<STRING> extractions for use as a model.

**Example 1:** if you added the specification:

```c
[method_setting = <REAL>]
```

you would copy over

```c
if ( method_setting >= 0 ) {
}
```

from idr-gen-code.C into ProblemDescDB.C and then populate the if block with a call to set the corresponding attribute within the `data_method` object using data extracted using the `rdata` array:

```c
if ( method_setting >= 0 ) {
    data_method.methodSetting = rdata[method_setting];
}
```
Use of a set member function within DataMethod is not needed since the data is public. The data is public since ProblemDescDB already provides sufficient encapsulation (ProblemDescDB::methodList, ProblemDescDB::variablesList, ProblemDescDB::interfaceList, ProblemDescDB::responsesList, and ProblemDescDB::strategySpec are private attributes), and no other classes have direct access. A similar model is used with SurrogateDataPoint objects contained in DakotaApproximation (DakotaApproximation::currentPoints) and with ParallelismLevel objects contained in ParallelLibrary (ParallelLibrary::parallelismLevels). Allowing public access to the Data class attributes is essentially equivalent to declaring ProblemDescDB a friend, but with the important bonus of a significant reduction in the amount of code to maintain. That is, the Data classes can be streamlined (and the work in modifying the input specification can be reduced) by omitting set/get functions.

**Example 2:** if you added the specification

```
[method_setup = <LISTof><REAL>]
```

you would copy over

```
if ( method_setup >= 0 ) {
    Int idr_table_len;
    Real** idr_table = idr_get_real_table( parsed_data, method_setup,
                                idr_table_len, 1, 1 );
}
```

from idr-gen-code.C into ProblemDescDB.C and then populate it with a loop which extracts each entry of the table and populates the corresponding attribute within the data_method object. The idr_table_len attribute is used for the loop limit and to size the data_method object.

```
if ( method_setup >= 0 ) {
    Int idr_table_len;
    Real** idr_table = idr_get_real_table( parsed_data, method_setup,
                                idr_table_len, 1, 1 );

    data_method.methodSetting.reshape(idr_table_len);
    for (int i = 0; i<idr_table_len; i++)
        data_method.methodSetting[i] = idr_table[0][i];
}
```

**Attention:**
If no new data attributes have been added, but instead there are only new settings for existing attributes, then you’re done with the database augmentation at this point (you just need to add code to use these new settings in the places where the existing attributes are used).

### 11.4.3 Augment/update get_<data_type>() functions

The final update step for ProblemDescDB.C involves extending the database retrieval functions. These retrieval functions accept an identifier string and return a database attribute of a particular type, e.g. a DakotaRealVector:

```
const DakotaRealVector& get_drv(const DakotaStrings entry_name);
```

The implementation of each of these functions has a simple series of if-else checks which return the appropriate attribute based on the identifier string. For example,
if (entry_name == "variables.continuous_design.initial_point")
    return (*variablesIter).continuousDesignVars;

appears at the top of ProblemDescDB::get_drv(). Based on the identifier string, it returns the continuousDesignVars attribute from a DataVariables object. Since there may be multiple variables specifications, the variablesIter list iterator identifies which node in the list of DataVariables objects is used. In particular, variablesList contains a list of all of the data_variables objects, one for each time variables_kwhandler() has been called by the parser. The particular variables object used for the data retrieval is managed by variablesIter, which is set in a set_db_list_nodes() operation that will not be described here.

There may be multiple DataVariables, DataInterface, DataResponses, and/or DataMethod objects. However, only one strategy specification is currently allowed so a list of DataStrategy objects is not needed. Rather, strategySpec is the lone DataStrategy object.

To augment the get_<data_type>() functions, add else blocks with new identifier strings which retrieve the appropriate data attributes from the Data class object. The style for the identifier strings is a top-down hierarchical description, with specification levels separated by periods and words separated with underscores, e.g. "keyword.group.specification.individual_specification". Use the (*listIter).attribute syntax for variables, interface, responses, and method specifications. For example, the method_setting example attribute would be added to get_drv() as:

    else if (entry_name == "method.method_name.method_setting")
        return (*methodIter).methodSetting;

A strategy specification addition would not use a (*listIter) syntax, but would instead look like:

    else if (entry_name == "strategy.strategy_name.strategy_setting")
        return strategySpec.strategySetting;

11.5 Update Corresponding Data Classes

In this step, we extend the Data class definitions (DataStrategy, DataMethod, DataVariables, DataInterface, and/or DataResponses) to include the new attributes referenced in Update keyword handler logic blocks and Augment/update get_<data_type>() functions.

11.5.1 Update the Data class header file

Add a new attribute to the private data for each of the new specifications. Follow the style guide for class attribute naming conventions (or mimic the existing code).

11.5.2 Update the .C file

Define defaults for the new attributes in the constructor initialization list (or in the case of DataMethod, in the body of the constructor for readability). Add the new attributes to the assign() function for use by
the copy constructor and assignment operator. Add the new attributes to the write(PackBuffer&), read(Un-PackBuffer&), and write(ostream&) functions, paying attention to using a consistent ordering.

11.6 Use get_<data_type>() Functions

At this point, the new specifications have been mapped through all of the database classes. The only remaining step is to retrieve the new data within the constructors of the classes that need it. This is done by invoking the get_<data_type>() function on the ProblemDescDB object using the identifier string you selected in Augment/update get_<data_type>() functions. For example, from DakotaModel.C:

```cpp
const DakotaString& interface_type = problem_db.get_string("interface.type");
```

passes the "interface.type" identifier string to the ProblemDescDB::get_string() retrieval function, which returns the desired attribute from the active DataInterface object.

Warning:
Use of the get_<data_type>() functions is restricted to class constructors, since only in class constructors are the data list iterators (i.e., methodIter, interfaceIter, variablesIter, and responsesIter) guaranteed to be set correctly. Outside of the constructors, the database list nodes will correspond to the last set operation, and may not return data from the desired list node.

11.7 Update the Documentation

Doxygen comments should be added to the Data class headers for the new attributes, and the reference manual sections describing the portions of dakota.input.spec that have been modified should be updated.
Index

~ANNSurf
   ANNSurf, 42
~AllMergedVarConstraints
   AllMergedVarConstraints, 25
~AllMergedVariables
   AllMergedVariables, 28
~AllVarConstraints
   AllVarConstraints, 32
~AllVariables
   AllVariables, 35
~AnalysisCode
   AnalysisCode, 40
~ApplicationInterface
   ApplicationInterface, 44
~ApproximationInterface
   ApproximationInterface, 55
~BranchBndStrategy
   BranchBndStrategy, 59
~COLINApplication
   COLINApplication, 61
~COLINOptimizer
   COLINOptimizer, 64
~CONMINOptimizer
   CONMINOptimizer, 72
~CommandLineHandler
   CommandLineHandler, 67
~CommandShell
   CommandShell, 68
~ConcurrentStrategy
   ConcurrentStrategy, 70
~CtelRegexp
   CtelRegexp, 79
~DACEIterator
   DACEIterator, 81
~DOTOptimizer
   DOTOptimizer, 212
~DakotaApproximation
   DakotaApproximation, 88
~DakotaArray
   DakotaArray, 90
~DakotaBaseVector
   DakotaBaseVector, 94
~DakotaBiStream
   DakotaBiStream, 100
~DakotaBoStream
   DakotaBoStream, 101
~DakotaGraphics
   DakotaGraphics, 104
~DakotaInterface
   DakotaInterface, 111
~DakotaIterator
   DakotaIterator, 118
~DakotaLeastSq
   DakotaLeastSq, 120
~DakotaList
   DakotaList, 122
~DakotaMatrix
   DakotaMatrix, 126
~DakotaModel
   DakotaModel, 136
~DakotaNonD
   DakotaNonD, 139
~DakotaOptLeastSq
   DakotaOptLeastSq, 147
~DakotaOptimizer
   DakotaOptimizer, 143
~DakotaResponse
   DakotaResponse, 150
~DakotaResponseRep
   DakotaResponseRep, 154
~DakotaStrategy
   DakotaStrategy, 162
~DakotaString
   DakotaString, 165
~DakotaVarConstraints
   DakotaVarConstraints, 172
~DakotaVariables
   DakotaVariables, 178
~DakotaVector
   DakotaVector, 180
~DataInterface
   DataInterface, 184
~DataMethod
   DataMethod, 189
~DataResponses
   DataResponses, 196
~DataStrategy
   DataStrategy, 199
~DataVariables
   DataVariables, 203
~DirectFnApplicInterface
  DirectFnApplicInterface, 208
~ForkAnalysisCode
  ForkAnalysisCode, 217
~ForkApplicInterface
  ForkApplicInterface, 219
~FundamentalVarConstraints
  FundamentalVarConstraints, 223
~FundamentalVariables
  FundamentalVariables, 227
~GetLongOpt
  GetLongOpt, 232
~HermiteSurf
  HermiteSurf, 236
~HierLayeredModel
  HierLayeredModel, 238
~KrigApprox
  KrigApprox, 242
~KrigingSurf
  KrigingSurf, 250
~LayeredModel
  LayeredModel, 252
~MARSurf
  MARSurf, 257
~MergedVarConstraints
  MergedVarConstraints, 259
~MergedVariables
  MergedVariables, 262
~MultilevelOptStrategy
  MultilevelOptStrategy, 266
~NLSSOLLeastSq
  NLSSOLLeastSq, 275
~NPSOLOptimizer
  NPSOLOptimizer, 294
~NestedModel
  NestedModel, 269
~NonDAdvMeanValue
  NonDAdvMeanValue, 278
~NonDLHSSampling
  NonDLHSSampling, 285
~NonDOptStrategy
  NonDOptStrategy, 287
~NonDPCESampling
  NonDPCESampling, 289
~NonDSampling
  NonDSampling, 291
~ParallelLibrary
  ParallelLibrary, 297
~ParamResponsePair
  ParamResponsePair, 309
~ParamStudy
  ParamStudy, 312
~ProblemDescDB
  ProblemDescDB, 316
~RespSurf
  RespSurf, 321
~SGOPTApplication
  SGOPTApplication, 325
~SGOPTOptimizer
  SGOPTOptimizer, 330
~SNLLBase
  SNLLBase, 335
~SNLLLeastSq
  SNLLLeastSq, 338
~SNLLOptimizer
  SNLLOptimizer, 341
~SOLBase
  SOLBase, 346
~SingleMethodStrategy
  SingleMethodStrategy, 331
~SingleModel
  SingleModel, 333
~SurrBasedOptStrategy
  SurrBasedOptStrategy, 350
~SurrLayeredModel
  SurrLayeredModel, 355
~SurrogateDataPoint
  SurrogateDataPoint, 360
~SysCallAnalysisCode
  SysCallAnalysisCode, 362
~SysCallApplicInterface
  SysCallApplicInterface, 364
~TaylorSurf
  TaylorSurf, 366
~VariablesUtil
  VariablesUtil, 367
~rSQPOptimizer
  rSQPOptimizer, 323

A
  CONMINOptimizer, 78
  KrigApprox, 248

ABOBJ1
  KrigApprox, 244
  activate_model auto graphics
  DakotaModel, 132
  active_set_vector
  DakotaResponse, 150
  ParamResponsePair, 310
  active_variables
  DakotaModel, 130
  activeSetVector
  COLINApplication, 61
  DakotaIterator, 115
  SGOPTApplication, 325
  activeSetVectorFlag
  DataInterface, 186
  actualInterfacePointer
cv, 35
discrete_variable_labels, 36
discrete_variables, 35
dv, 35
inactive_continuous_variables, 36
inactive_discrete_variables, 36
numCDV, 37
numCSV, 37
numDDV, 37
numDSV, 37
numUV, 37
operator==, 38
read, 36, 37
read_annotated, 36
tv, 35
write, 36, 37
write_annotated, 36
AllVariables, 35
AllVariables, 38
allVariables
DakotaIterator, 116
allVarsFlag
DataMethod, 194
NonDLHSampling, 285
ALPHAX
KrigApprox, 244
annFVFlag
NonDAdvMeanValue, 281
analysis_communicator_rank
ParallelLibrary, 301
analysis_communicator_size
ParallelLibrary, 301
analysis_intra_communicator
ParallelLibrary, 301
analysis_master_flag
ParallelLibrary, 301
analysis_server_id
ParallelLibrary, 301
analysis_servers
ParallelLibrary, 301
AnalysisCode
~/AnalysisCode, 40
AnalysisCode, 40
apreproFlag, 40
define_filenames, 39
filenameKey, 41
fileSaveFlag, 40
fileTagFlag, 40
iFilterName, 40
input_filter_name, 39
modified_parameters_filename, 39
modified_results_filename, 39
modifiedParamsFileName, 40
modifiedResFileName, 40
numPrograms, 40
oFilterName, 40
output_filter_name, 39
parallelLib, 41
parametersFileName, 40
parametersFNameList, 41
program_names, 39
programNames, 40
read_results_file, 39
results_filename, 39
resultsFileName, 40
resultsFNameList, 41
suppressOutputFlag, 40
write_parameters_file, 39
AnalysisCode, 39
analysisComm
ApplicationInterface, 47
analysisCommRank
ParallelLibrary, 305
analysisCommSize
ParallelLibrary, 305
analysisDrivers
ApplicationInterface, 46
DataInterface, 185
analysisIntraComm
ParallelLibrary, 305
analysisMasterFlag
ParallelLibrary, 305
analysisMessagePass
ApplicationInterface, 46
analysisScheduling
ApplicationInterface, 48
DataInterface, 186
analysisServerId
ApplicationInterface, 46
parallelLib, 305
analysisServers
DataInterface, 186
analysisUsage
DataInterface, 185
annObject
ANNSurf, 42
ANNSurf, 42
~/ANNSurf, 42
annObject, 42
ANNSurf, 42
find_coefficients, 42
get_value, 42
required_samples, 42
ApplicationInterface
~ApplicationInterface, 44
analysisComm, 47
analysisCommRank, 46
analysisCommSize, 46
analysisDrivers, 47
analysisMessagePass, 46
analysisScheduling, 48
analysisServerId, 46
ApplicationInterface, 44
asvControlFlag, 49
asynchLocalAnalysisConcurrency, 44
asynchLocalAnalysisFlag, 46
asynchLocalEvalConcurrency, 48
beforeSynchDuplicateIds, 49
beforeSynchDuplicateIndices, 49
beforeSynchDuplicateResponses, 49
beforeSynchPRPList, 50
clear_bookkeeping, 45
continuation, 48
defaultASV, 49
derived_map, 45
derived_map_asynch, 45
derived_synch, 45
derived_synch_nowait, 45
derived_synchronous_local_analysis, 45
evalAnalysisIntraComm, 47
evalCacheFlag, 49
evalComm, 47
evalCommRank, 46
evalCommSize, 46
evalDedMasterFlag, 46
evalMessagePass, 46
evalScheduling, 48
evalServerId, 46
failAction, 49
failRecoveryFnVals, 49
failRetryLimit, 49
free_communicators, 44
get_source_pair, 48
headerFlag, 49
historyDuplicateIds, 49
historyDuplicateResponses, 49
init_communicators, 44
interface_synchronization, 44
interfaceSynchronization, 49
iteratorCommRank, 46
iteratorCommSize, 46
lenPRPairMessage, 47
lenResponseMessage, 47
lenVarsASVMessage, 47
lenVarsMessage, 47
manage_failure, 44
multiProcAnalysisFlag, 47
numAnalysisDrivers, 47
numAnalysisServers, 47
numEvalServers, 48
parallelLib, 45
procsPerAnalysis, 48
restartFileFlag, 49
runningList, 50
suppressOutput, 46
worldRank, 46
worldSize, 46
ApplicationInterface, 44
asynchronous_local_evaluations, 52
duplicate_detect, 53
init_serial, 50
map, 50
self_schedule_analyses, 51
self_schedule_evaluations, 52
serve_analyses_synch, 51
serve_evaluations, 51
serve_evaluations_asynch, 53
serve_evaluations_peer, 54
serve_evaluations_synch, 53
static_schedule_evaluations, 52
stop_evaluation_servers, 51
synch, 50
synch_nowait, 51
synchronous_local_evaluations, 53
apply_correction
DakotaModel, 129
LayeredModel, 252
approxBuilds
LayeredModel, 255
approxCorrectionOrder
DataInterface, 187
approxCorrectionType
DataInterface, 187
approxDaceMethodPtr
DataInterface, 187
approxGradUsageFlag
DataInterface, 187
approximateModel
SurrBasedOptStrategy, 351
approximation_coefficients
ApproximationInterface, 55
DakotaApproximation, 85
DakotaInterface, 108
HermiteSurf, 236
RespSurf, 321
SurrLayeredModel, 356
ApproximationInterface
~ApproximationInterface, 55
approximation_coefficients, 55
INDEX

BaseConstructor, 58

boundConstraintFlag
    DakotaOptLeastSq, 148

boundsArraySize
    SOLBase, 347

branchBndNumSamplesNode
    DataStrategy, 200

branchBndNumSamplesRoot
    DataStrategy, 200

BranchBndStrategy
    BranchBndStrategy, 59

argC, 60

argV, 60

BranchBndStrategy, 59

numIteratorServers, 59

numNodeSamples, 59

numRootSamples, 59

picoComm, 59

picoCommRank, 59

picoCommSize, 60

picoListOfIntegers, 60

picoLowerBnds, 60

picoUpperBnds, 60

run_strategy, 59

selectedIterator, 59

userDefinedModel, 59

BranchBndStrategy, 59

build
    DakotaApproximation, 85

build_approximation
    DakotaModel, 128
    HierLayeredModel, 238
    SurrLayeredModel, 358

build_global_approximation
    ApproximationInterface, 55
    DakotaInterface, 108

build_label
    ProblemDescDB, 318

build_labels
    ProblemDescDB, 318

build_labels_partial
    ProblemDescDB, 319

build_local_approximation
    ApproximationInterface, 55
    DakotaInterface, 108

C

CONMINOptimizer, 77

KrigApprox, 248

cantilever
    DirectFnApplicInterface, 210

centered_loop
    ParamStudy, 313

centeringParam
    DataMethod, 192
centralPath
  DataMethod, 191
chaosCoeffs
  HermiteSurf, 237
chaosSamples
  HermiteSurf, 237
check_error
  NonDSampling, 292
check_input
  ProblemDescDB, 316
check_status
  ForkAnalysisCode, 218
check_submodel_compatibility
  LayeredModel, 252
check_usage
  CommandLineHandler, 67
choiCorrMatrix
  NonDAdvMeanValue, 280
c LDLambda
  SOLBase, 347
clear_bookkeeping
  ApplicationInterface, 45
clearErrors
  CtelRegexp, 79
close
  DakotaGraphics, 104
close_streams
  ParallelLibrary, 308
coeffArray
  NonDPCESampling, 289
COLINApplication, 61
  ~COLINApplication, 61
  activeSetVector, 61
COLINApplication, 61
dakota_asyncFlag, 62
dakota_vars, 62
dakotaCompletionList, 61
dakotaModelAsynchFlag, 61
dakotaResponseList, 61
DoEval, 62
map_response, 63
multiobjModifyPtr, 62
next_eval, 62
num_integer_params, 62
num_real_params, 62
numNonlinCons, 62
numObjFns, 61
synchronize, 62
userDefinedModel, 61
COLINOptimizer, 64
  ~COLINOptimizer, 64
  COLINOptimizer, 64
  find_optimum, 65
  get_min_point, 64
  optimizer, 64
  problem, 64
  rng, 65
  set_initial_point, 64
  set_method_options, 64
  set_rng, 64
  set_standard_method_options, 65
ColinPoint
  ivec, 66
  rvec, 66
ColinPoint, 66
command_usage
  SysCallAnalysisCode, 362
CommandLineHandler
  ~CommandLineHandler, 67
  check_usage, 67
CommandLineHandler, 67
read_restart_evals, 67
CommandLineHandler, 67
CommandShell
  ~CommandShell, 68
  asyncFlag, 68
  operator<<, 68
  suppress_output_flag, 68
  suppressOutputFlag, 68
  unixCommand, 68
CommandShell, 68
flush, 69
commandUsage
  SysCallAnalysisCode, 362
compile
  CtelRegexp, 79
completionList
  DakotaInterface, 109
NestedModel, 271
compute_correction
  DakotaModel, 129
LayeredModel, 254
compute_penalty_function
  SurrBasedOptStrategy, 353
compute_response
  DakotaModel, 129
compute_statistics
  NonDSampling, 291
compute_vector_steps
  ParamStudy, 312
concatenate_restart
  restartUtil.C, 372
concurrentParameterSets
  DataStrategy, 201
concurrentRandomJobs
  DataStrategy, 201

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (®) 1997-2002
concurrentSeed
DataStrategy, 201
ConcurrentStrategy
~ConcurrentStrategy, 70
ConcurrentStrategy, 70
iteratorServerId, 71
multiStartFlag, 70
numIteratorJobs, 70
numIteratorServers, 70
parameterSets, 70
run_strategy, 70
selectedIterator, 70
strategyDedicatedMasterFlag, 71
userDefinedModel, 70
ConcurrentStrategy, 70
conminDesVars
CONMINOptimizer, 74
conminInfo
CONMINOptimizer, 75
KrigApprox, 243
conminLowerBnds
CONMINOptimizer, 74
CONMINOptimizer, 72
~CONMINOptimizer, 72
A, 78
allocate_workspace, 72
B, 77
C, 77
conminDesVars, 74
conminInfo, 75
conminLowerBnds, 74
CONMINOptimizer, 72
conminUpperBnds, 74
CONMINOptimizer, 72
conminUpperBnds, 74
constraintMappingIndices, 76
constraintMappingMultipliers, 76
constraintMappingOffsets, 76
CT, 77
CTL, 74
CTLMIN, 74
CTMIN, 73
DABFUN, 74
DELFUN, 74
DF, 77
FDCH, 73
FDCHM, 73
findOptimum, 72
G1, 77
G2, 77
IC, 78
IPRINT, 73
ISC, 78
ITMAX, 73
localConstraintValues, 75
MS1, 77
N1, 76
N2, 76
N3, 76
N4, 76
N5, 76
NFDG, 73
optimizationType, 75
printControl, 75
S, 77
SCAL, 77
conminSingleArray
KrigApprox, 242
conminThetaLowerBnds
KrigApprox, 244
conminThetaUpperBnds
KrigApprox, 244
conminThetaVars
KrigApprox, 244
conminUpperBnds
CONMINOptimizer, 74
constraintASVFlag
SNLLBase, 336
constraint0_evaluator
SNLLOptimizer, 344
constraint1_evaluator
SNLLOptimizer, 344
constraint2_evaluator
SNLLOptimizer, 344
constraint2_evaluator_gn
SNLLLeastSq, 338
constraint_eval
SOLBase, 347
constraintJacMatrixF77
SOLBase, 348
constraintMappingIndices
CONMINOptimizer, 76
DOTOptimizer, 215
constraintMappingMultipliers
CONMINOptimizer, 76
DOTOptimizer, 215
constraintMappingOffsets
CONMINOptimizer, 76
DOTOptimizer, 215
constraintState
SOLBase, 347
constraintTol
DakotaOptLeastSq, 147
SurrBasedOptStrategy, 351
constraintTolerance
DataMethod, 190
constraintVector
KrigApprox, 246
constrCoeffs
NestedModel, 271
contains
  DakotaList, 124
  DakotaString, 166
continuation
  ApplicationInterface, 48
continuous_lower_bounds
    AllMergedVarConstraints, 25
    AllVarConstraints, 32
    DakotaModel, 131
    DakotaVarConstraints, 168
    FundamentalVarConstraints, 223
    MergedVarConstraints, 259
continuous_upper_bounds
    AllMergedVarConstraints, 25
    AllVarConstraints, 32
    DakotaModel, 131
    DakotaVarConstraints, 168
    FundamentalVarConstraints, 223
    MergedVarConstraints, 259
continuous_variable_labels
    AllMergedVariables, 28, 29
    AllVariables, 35, 36
    DakotaModel, 130
    DakotaVariables, 175
    FundamentalVariables, 227, 228
    MergedVariables, 262, 263
continuous_variables
    AllMergedVariables, 28
    AllVariables, 35
    DakotaModel, 130
    DakotaVariables, 174
    FundamentalVariables, 227
    MergedVariables, 262
continuesDesignLabels
    DataVariables, 205
    FundamentalVariables, 229
continuesDesignLowerBnds
    DataVariables, 204
    FundamentalVarConstraints, 224
continuesDesignUpperBnds
    DataVariables, 204
    FundamentalVarConstraints, 225
continuesDesignVars
    DataVariables, 204
    FundamentalVariables, 229
continuesStateLabels
    DataVariables, 207
    FundamentalVariables, 230
continuesStateLowerBnds
    DataVariables, 207
    FundamentalVarConstraints, 225
continuesStateUpperBnds
    DataVariables, 207
    FundamentalVarConstraints, 228
continuousStateVars
    DataVariables, 206
    FundamentalVariables, 229
continuousVars
    SurrogateDataPoint, 360
contractAfterFail
    DataMethod, 193
contractFactor
    DataMethod, 192
convergenceFlag
    SurrBasedOptStrategy, 351
convergenceTol
    DakotaOptLeastSq, 147
    SurrBasedOptStrategy, 351
convergenceTolerance
    DataMethod, 190
copy
    DakotaResponse, 151
    DakotaVariables, 179
    SGOPApplication, 325
copy_con_grad
    SNLLBase, 336
copy_con_hess
    SNLLBase, 336
copy_con_vals
    SNLLBase, 336
copy_rep
    AllMergedVariables, 30
    AllVariables, 37
    DakotaVariables, 177
    FundamentalVariables, 229
    MergedVariables, 264
copy_results
    DakotaResponse, 152
    DakotaResponseRep, 155
correctedRespLevel
    NonDAdvMeanValue, 284
correctedResponseArray
    LayeredModel, 252
correctedResponseList
    LayeredModel, 253
correctionComputed
    LayeredModel, 254
correctionFlag
    SurrBasedOptStrategy, 352
correctionOrder
    LayeredModel, 253
correctionType
    LayeredModel, 253
correlationFlag
    DakotaNonD, 141
correlationMatrix
    KrigApprox, 246
correlationVector
KrigingSurf, 250
create_plots_2d
   DakotaGraphics, 105
create_tabular_datastream
   DakotaGraphics, 105
crossoverRate
   DataMethod, 192
crossoverType
   DataMethod, 193
CT
   CONMINOptimizer, 77
   KrigApprox, 247
CtelRegexp
   ~CtelRegexp, 79
   clearErrors, 79
   compile, 79
   CtelRegexp, 79, 80
   getRe, 79
   getStatus, 79
   getStatusMsg, 79
   match, 79
   operator=, 80
   r, 80
   split, 79
   status, 80
   statusMsg, 80
   strPattern, 80
CtelRegexp, 79
CTL
   CONMINOptimizer, 74
   KrigApprox, 243
CTLMIN
   CONMINOptimizer, 74
   KrigApprox, 243
CTMIN
   CONMINOptimizer, 73
   KrigApprox, 243
currentResponse
   DakotaModel, 132
current_variables
   DakotaModel, 132
currentPoints
   DakotaApproximation, 86
currentResponse
   DakotaModel, 133
currentVariables
   DakotaModel, 133
cv
   AllMergedVariables, 28
   AllVariables, 35
   DakotaModel, 130
   DakotaVariables, 174
   FundamentalVariables, 227
   MergedVariables, 262
cy_khead
   DirectFnApplicInterface, 210
DABFUN
   CONMINOptimizer, 74
   KrigApprox, 243
daceCenterPtFlag
   SurrBasedOptStrategy, 352
DACEIterator, 81
   ~DACEIterator, 81
   allDataFlag, 82
   bestObjectiveFn, 82
   bestResponses, 82
   bestVariables, 82
   bestViolations, 82
   DACEIterator, 83
   daceMethod, 82
   iterator_response_results, 81
   iterator_variable_results, 81
   multiObjWeights, 83
   nonlinearEqTargets, 83
   nonlinearIneqLowerBnds, 83
   nonlinearIneqUpperBnds, 83
   numDACERuns, 82
   numNonlinearEqConstraints, 83
   numNonlinearIneqConstraints, 83
   numObjectiveFunctions, 82
   numSamples, 82
   numSymbols, 82
   originalSeed, 82
   print_iterator_results, 81
   randomSeed, 82
   resolve_samples_symbols, 84
   run_iterator, 83
   sampling_reset, 81
   sampling_scheme, 81
   update_best, 81
   varyPattern, 82
daceIterator
   SurrLayeredModel, 357
daceMethod
   DACEIterator, 82
   DataMethod, 194
daceMethodPointer
   ApproximationInterface, 57
   SurrLayeredModel, 357
dakota_asynch_flag
   COLINApplication, 62
   SGOPTApplication, 326
dakota-vars
   COLINApplication, 62
DakotaApproximation
add_point, 87
add_point_rebuild, 85
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>approximation_coefficients</td>
<td>85</td>
</tr>
<tr>
<td>approxLowerBounds</td>
<td>87</td>
</tr>
<tr>
<td>approxRep</td>
<td>87</td>
</tr>
<tr>
<td>approxType</td>
<td>86</td>
</tr>
<tr>
<td>approxUpperBounds</td>
<td>87</td>
</tr>
<tr>
<td>build</td>
<td>85</td>
</tr>
<tr>
<td>currentPoints</td>
<td>86</td>
</tr>
<tr>
<td>DakotaApproximation</td>
<td>87, 88</td>
</tr>
<tr>
<td>draw_surface</td>
<td>86</td>
</tr>
<tr>
<td>find_coefficients</td>
<td>86</td>
</tr>
<tr>
<td>get_gradient</td>
<td>85</td>
</tr>
<tr>
<td>get_value</td>
<td>85</td>
</tr>
<tr>
<td>gradientFlag</td>
<td>86</td>
</tr>
<tr>
<td>gradVector</td>
<td>86</td>
</tr>
<tr>
<td>num_variables</td>
<td>86</td>
</tr>
<tr>
<td>numCurrentPoints</td>
<td>86</td>
</tr>
<tr>
<td>numSamples</td>
<td>86</td>
</tr>
<tr>
<td>numVars</td>
<td>86</td>
</tr>
<tr>
<td>referenceCount</td>
<td>87</td>
</tr>
<tr>
<td>required_samples</td>
<td>85</td>
</tr>
<tr>
<td>set_bounds</td>
<td>86</td>
</tr>
<tr>
<td>verboseFlag</td>
<td>86</td>
</tr>
<tr>
<td>DakotaApproximation</td>
<td>85</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>90, 91</td>
</tr>
<tr>
<td>length</td>
<td>91</td>
</tr>
<tr>
<td>operator=</td>
<td>90</td>
</tr>
<tr>
<td>print</td>
<td>91</td>
</tr>
<tr>
<td>read</td>
<td>91</td>
</tr>
<tr>
<td>reshape</td>
<td>91</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>90</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>91</td>
</tr>
<tr>
<td>data</td>
<td>92</td>
</tr>
<tr>
<td>operator T *</td>
<td>92</td>
</tr>
<tr>
<td>operator()</td>
<td>92</td>
</tr>
<tr>
<td>operator=</td>
<td>92</td>
</tr>
<tr>
<td>operator[]</td>
<td>92</td>
</tr>
<tr>
<td>testClass</td>
<td>93</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>91</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>92</td>
</tr>
<tr>
<td>operator T *</td>
<td>92</td>
</tr>
<tr>
<td>operator()</td>
<td>92</td>
</tr>
<tr>
<td>operator=</td>
<td>92</td>
</tr>
<tr>
<td>operator[]</td>
<td>92</td>
</tr>
<tr>
<td>testClass</td>
<td>93</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>94</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>95</td>
</tr>
<tr>
<td>array</td>
<td>96</td>
</tr>
<tr>
<td>DakotaArray</td>
<td>96</td>
</tr>
<tr>
<td>data</td>
<td>96</td>
</tr>
<tr>
<td>length</td>
<td>96</td>
</tr>
<tr>
<td>operator()</td>
<td>96</td>
</tr>
<tr>
<td>operator[]</td>
<td>95, 96</td>
</tr>
<tr>
<td>reshape</td>
<td>96</td>
</tr>
<tr>
<td>DakotaBiStream</td>
<td>98, 99</td>
</tr>
<tr>
<td>inBuf</td>
<td>99</td>
</tr>
<tr>
<td>operator&gt;&gt;</td>
<td>98, 99</td>
</tr>
<tr>
<td>xdrInBuf</td>
<td>99</td>
</tr>
<tr>
<td>DakotaBiStream</td>
<td>98</td>
</tr>
<tr>
<td>~DakotaBiStream</td>
<td>100</td>
</tr>
<tr>
<td>DakotaBiStream</td>
<td>99</td>
</tr>
<tr>
<td>operator&gt;&gt;</td>
<td>100</td>
</tr>
<tr>
<td>DakotaBoStream</td>
<td>101</td>
</tr>
<tr>
<td>DakotaBoStream</td>
<td>101–103</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>101, 102</td>
</tr>
<tr>
<td>outBuf</td>
<td>102</td>
</tr>
<tr>
<td>xdrOutBuf</td>
<td>102</td>
</tr>
<tr>
<td>DakotaBoStream</td>
<td>101</td>
</tr>
<tr>
<td>DakotaBoStream</td>
<td>102, 103</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>103</td>
</tr>
<tr>
<td>testClass</td>
<td>103</td>
</tr>
<tr>
<td>dakotaCompletionList</td>
<td></td>
</tr>
<tr>
<td>COLINAApplication</td>
<td>61</td>
</tr>
<tr>
<td>SGOPTApplication</td>
<td>326</td>
</tr>
<tr>
<td>DakotaGraphics</td>
<td>104</td>
</tr>
<tr>
<td>~DakotaGraphics</td>
<td>104</td>
</tr>
<tr>
<td>close</td>
<td>104</td>
</tr>
<tr>
<td>DakotaGraphics</td>
<td>104</td>
</tr>
<tr>
<td>graphics2D</td>
<td>104</td>
</tr>
<tr>
<td>graphicsCntr</td>
<td>104</td>
</tr>
<tr>
<td>tabularDataFlag</td>
<td>104</td>
</tr>
<tr>
<td>tabularDataFStream</td>
<td>105</td>
</tr>
<tr>
<td>win2dOn</td>
<td>104</td>
</tr>
<tr>
<td>win3dOn</td>
<td>104</td>
</tr>
<tr>
<td>DakotaGraphics</td>
<td>104</td>
</tr>
<tr>
<td>add datapoint</td>
<td>105</td>
</tr>
<tr>
<td>create plots 2d</td>
<td>105</td>
</tr>
<tr>
<td>create tabular datastream</td>
<td>105</td>
</tr>
<tr>
<td>show data 3d</td>
<td>105</td>
</tr>
<tr>
<td>DakotaInterface</td>
<td>110, 111</td>
</tr>
<tr>
<td>debugFlag</td>
<td>110</td>
</tr>
<tr>
<td>fnEvalId</td>
<td>109</td>
</tr>
<tr>
<td>free communicators</td>
<td>108</td>
</tr>
<tr>
<td>init communicators</td>
<td>108</td>
</tr>
<tr>
<td>init serial</td>
<td>108</td>
</tr>
<tr>
<td>interface synchronziation</td>
<td>108</td>
</tr>
<tr>
<td>interface type</td>
<td>108</td>
</tr>
<tr>
<td>interfaceRep</td>
<td>110</td>
</tr>
</tbody>
</table>
ocurrencesOf, 125
operator[], 125
remove, 124
removeAt, 124
sort, 124
testClass, 123

DakotaMatrix
~DakotaMatrix, 126
DakotaMatrix, 126
num_columns, 126
num_rows, 126
print, 126
read, 126
reshape 2d, 126

DakotaMatrix, 126
operator=, 127
testClass, 127
DakotaModel
activate_model_auto_graphics, 132
active_variables, 130
apply_correction, 129
approximation_coefficients, 129
asvList, 135
asynch_compute_response, 129, 130
asynch_flag, 132
asynchEvalFlag, 134
asynchFDFlag, 134
auto_correction, 129
build_approximation, 128
compute_correction, 129
calculate_response, 129
continuous_lower_bounds, 131
continuous_upper_bounds, 131
continuous_variable_labels, 130
continuous_variables, 130
current_response, 132
current_variables, 132
currentResponse, 133
currentVariables, 133
cv, 130
DakotaModel, 136
dbFnsList, 135
dbResponseList, 135
deltaList, 135
derived_asynch_compute_response, 133
derived_compute_response, 133
derived_init_communicators, 133
derived_master_overload, 129
derived_synchronize, 133
derived_synchronize_nowait, 133
discrete_lower_bounds, 131
discrete_upper_bounds, 131
discrete_variable_labels, 130
discrete_variables, 130
dv, 130
finiteDiffSS, 135
gradient_concurrency, 132
gradient_method, 132
gradType, 135
idAnalytic, 135
inactive_continuous_lower_bounds, 131
inactive_continuous_upper_bounds, 131
inactive_continuous_variables, 131
inactive_discrete_lower_bounds, 131
inactive_discrete_upper_bounds, 131
inactive_discrete_variables, 131
initialMapList, 135
intervalType, 135
is_null, 132
linear_eq_constraint_coeffs, 132
linear_eq_constraint_targets, 132
linear_ineq_constraint_coeffs, 131
linear_ineq_constraint_lower_bounds, 132
linear_ineq_constraint_upper_bounds, 132
maximum_concurrency, 128
merged_integer_list, 132
message_lengths, 132
messageLengths, 134
methodSrc, 135
model_type, 132
modelAutoGraphicsFlag, 134
modelRep, 134
modelType, 134
new_eval_counter, 129
num_functions, 130
num_linear_eq_constraints, 131
num_linear_ineq_constraints, 131
numFns, 133
numGradVars, 133
numMapsList, 135
parallelLib, 134
prob_desc_db, 132
probDescDB, 134
quietFlag, 135
referenceCount, 134
responseArray, 135
responseList, 135
serve, 129
silentFlag, 135
stop_servers, 129
subordinate_iterator, 128
subordinate_model, 128
synchronize, 130
synchronize_nowait, 130
synchronize_nowait_completions, 129
total_eval_counter, 129
tv, 130

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2002
<table>
<thead>
<tr>
<th>Function/Variable</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>update_approximation</td>
<td>128</td>
</tr>
<tr>
<td>userDefinedVarConstraints</td>
<td>133</td>
</tr>
<tr>
<td>varsList</td>
<td>135</td>
</tr>
<tr>
<td>DakotaModel</td>
<td>128</td>
</tr>
<tr>
<td>~DakotaModel</td>
<td>136</td>
</tr>
<tr>
<td>DakotaModel</td>
<td>136</td>
</tr>
<tr>
<td>fd_gradients</td>
<td>137</td>
</tr>
<tr>
<td>get_model</td>
<td>137</td>
</tr>
<tr>
<td>initCommunicators</td>
<td>137</td>
</tr>
<tr>
<td>local_eval_synchronization</td>
<td>137</td>
</tr>
<tr>
<td>manage_asv</td>
<td>138</td>
</tr>
<tr>
<td>operator=</td>
<td>137</td>
</tr>
<tr>
<td>synchronize_fd_gradients</td>
<td>138</td>
</tr>
<tr>
<td>update_response</td>
<td>138</td>
</tr>
<tr>
<td>dakotaModelAsynchFlag</td>
<td></td>
</tr>
<tr>
<td>COLINApplication</td>
<td>61</td>
</tr>
<tr>
<td>SGOPTApplication</td>
<td>325</td>
</tr>
<tr>
<td>DakotaNonD</td>
<td>139</td>
</tr>
<tr>
<td>~DakotaNonD</td>
<td>139</td>
</tr>
<tr>
<td>correlationFlag</td>
<td>141</td>
</tr>
<tr>
<td>DakotaNonD</td>
<td>139</td>
</tr>
<tr>
<td>finalStatistics</td>
<td>141</td>
</tr>
<tr>
<td>histogramBinPairs</td>
<td>140</td>
</tr>
<tr>
<td>histogramPointPairs</td>
<td>140</td>
</tr>
<tr>
<td>iterator_response_results</td>
<td>139</td>
</tr>
<tr>
<td>lognormalDistLowerBnds</td>
<td>140</td>
</tr>
<tr>
<td>lognormalDistUpperBnds</td>
<td>140</td>
</tr>
<tr>
<td>lognormalErrFacts</td>
<td>140</td>
</tr>
<tr>
<td>lognormalMeans</td>
<td>140</td>
</tr>
<tr>
<td>lognormalStdDevs</td>
<td>140</td>
</tr>
<tr>
<td>loguniformDistLowerBnds</td>
<td>140</td>
</tr>
<tr>
<td>loguniformDistUpperBnds</td>
<td>140</td>
</tr>
<tr>
<td>mean95CIDeltas</td>
<td>141</td>
</tr>
<tr>
<td>meanStats</td>
<td>141</td>
</tr>
<tr>
<td>normalDistLowerBnds</td>
<td>140</td>
</tr>
<tr>
<td>normalDistUpperBnds</td>
<td>140</td>
</tr>
<tr>
<td>normalMeans</td>
<td>139</td>
</tr>
<tr>
<td>normalStdDevs</td>
<td>139</td>
</tr>
<tr>
<td>numHistogramVars</td>
<td>141</td>
</tr>
<tr>
<td>numLognormalVars</td>
<td>141</td>
</tr>
<tr>
<td>numLoguniformVars</td>
<td>141</td>
</tr>
<tr>
<td>numNormalVars</td>
<td>140</td>
</tr>
<tr>
<td>numResponseFunctions</td>
<td>141</td>
</tr>
<tr>
<td>numUncertainVars</td>
<td>141</td>
</tr>
<tr>
<td>numUniformVars</td>
<td>141</td>
</tr>
<tr>
<td>numWeibullVars</td>
<td>141</td>
</tr>
<tr>
<td>probMoreThanThresh</td>
<td>141</td>
</tr>
<tr>
<td>quantifyuncertainty</td>
<td>139</td>
</tr>
<tr>
<td>respThresh</td>
<td>141</td>
</tr>
<tr>
<td>run_iterator</td>
<td>139</td>
</tr>
<tr>
<td>stdDevStats</td>
<td>141</td>
</tr>
<tr>
<td>uncertainCorrelations</td>
<td>140</td>
</tr>
<tr>
<td>uniformDistLowerBnds</td>
<td>140</td>
</tr>
<tr>
<td>uniformDistUpperBnds</td>
<td>140</td>
</tr>
<tr>
<td>weibullAlphas</td>
<td>140</td>
</tr>
<tr>
<td>weibullBetas</td>
<td>140</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
<td>143</td>
</tr>
<tr>
<td>~DakotaOptimizer</td>
<td>143</td>
</tr>
<tr>
<td>find optimum</td>
<td>143</td>
</tr>
<tr>
<td>multi_objective_weights</td>
<td>143</td>
</tr>
<tr>
<td>multiObjWeights</td>
<td>144</td>
</tr>
<tr>
<td>numObjectiveFunctions</td>
<td>144</td>
</tr>
<tr>
<td>staticMultiObjWeights</td>
<td>144</td>
</tr>
<tr>
<td>staticNumObjFns</td>
<td>144</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
<td>143</td>
</tr>
<tr>
<td>~DakotaOptimizer</td>
<td>144</td>
</tr>
<tr>
<td>multi_objective_modify</td>
<td>145</td>
</tr>
<tr>
<td>print_iterator_results</td>
<td>144</td>
</tr>
<tr>
<td>run_iterator</td>
<td>144</td>
</tr>
<tr>
<td>DakotaOptLeastSq</td>
<td>146</td>
</tr>
<tr>
<td>~DakotaOptLeastSq</td>
<td>147</td>
</tr>
<tr>
<td>bestResponses</td>
<td>148</td>
</tr>
<tr>
<td>bestVariables</td>
<td>148</td>
</tr>
<tr>
<td>bigIntBoundSize</td>
<td>147</td>
</tr>
<tr>
<td>bigRealBoundSize</td>
<td>147</td>
</tr>
<tr>
<td>boundConstraintFlag</td>
<td>148</td>
</tr>
<tr>
<td>constraintTol</td>
<td>147</td>
</tr>
<tr>
<td>convergenceTol</td>
<td>147</td>
</tr>
<tr>
<td>DakotaOptLeastSq</td>
<td>146, 149</td>
</tr>
<tr>
<td>iterator_response_results</td>
<td>146</td>
</tr>
<tr>
<td>iterator_variable_results</td>
<td>146</td>
</tr>
<tr>
<td>linearEqConstraintCoeffs</td>
<td>148</td>
</tr>
<tr>
<td>linearEqTargets</td>
<td>148</td>
</tr>
<tr>
<td>linearIneqConstraintCoeffs</td>
<td>147</td>
</tr>
<tr>
<td>linearIneqLowerBnds</td>
<td>147</td>
</tr>
<tr>
<td>linearIneqUpperBnds</td>
<td>148</td>
</tr>
<tr>
<td>nonlinearEqTargets</td>
<td>147</td>
</tr>
<tr>
<td>nonlinearIneqLowerBnds</td>
<td>147</td>
</tr>
<tr>
<td>nonlinearIneqUpperBnds</td>
<td>147</td>
</tr>
<tr>
<td>numConstraints</td>
<td>147</td>
</tr>
<tr>
<td>numLinearConstraints</td>
<td>148</td>
</tr>
<tr>
<td>numLinearEqConstraints</td>
<td>148</td>
</tr>
<tr>
<td>numLinearIneqConstraints</td>
<td>147</td>
</tr>
<tr>
<td>numNonlinearConstraints</td>
<td>147</td>
</tr>
<tr>
<td>numNonlinearEqConstraints</td>
<td>147</td>
</tr>
<tr>
<td>numNonlinearIneqConstraints</td>
<td>147</td>
</tr>
<tr>
<td>speculativeFlag</td>
<td>148</td>
</tr>
<tr>
<td>staticNumContinuousVars</td>
<td>148</td>
</tr>
<tr>
<td>staticNumNonlinearConstraints</td>
<td>148</td>
</tr>
<tr>
<td>vendorNumericalGradFlag</td>
<td>148</td>
</tr>
<tr>
<td>DakotaOptLeastSq</td>
<td>146</td>
</tr>
<tr>
<td>~DakotaOptLeastSq</td>
<td>149</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>150</td>
</tr>
<tr>
<td>~DakotaResponse</td>
<td>150</td>
</tr>
<tr>
<td>activeSetVector</td>
<td>150</td>
</tr>
<tr>
<td>copy</td>
<td>151</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>copy_results</td>
<td>152</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>150, 153</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>156</td>
</tr>
<tr>
<td>dataSize</td>
<td>152</td>
</tr>
<tr>
<td>fnTags</td>
<td>150</td>
</tr>
<tr>
<td>function_gradients</td>
<td>151</td>
</tr>
<tr>
<td>function_hessians</td>
<td>151</td>
</tr>
<tr>
<td>function_values</td>
<td>150, 151</td>
</tr>
<tr>
<td>interfaceId</td>
<td>150</td>
</tr>
<tr>
<td>numFunctions</td>
<td>150</td>
</tr>
<tr>
<td>operator</td>
<td>152</td>
</tr>
<tr>
<td>operator=</td>
<td>150</td>
</tr>
<tr>
<td>operator==</td>
<td>152</td>
</tr>
<tr>
<td>overlay</td>
<td>152</td>
</tr>
<tr>
<td>purgeInactive</td>
<td>152</td>
</tr>
<tr>
<td>read</td>
<td>151</td>
</tr>
<tr>
<td>read.annotated</td>
<td>151</td>
</tr>
<tr>
<td>read_data</td>
<td>152</td>
</tr>
<tr>
<td>read_tabular</td>
<td>151</td>
</tr>
<tr>
<td>reset</td>
<td>152</td>
</tr>
<tr>
<td>responseRep</td>
<td>152</td>
</tr>
<tr>
<td>write</td>
<td>151</td>
</tr>
<tr>
<td>write.annotated</td>
<td>151</td>
</tr>
<tr>
<td>write_data</td>
<td>152</td>
</tr>
<tr>
<td>write_tabular</td>
<td>151</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>150</td>
</tr>
<tr>
<td>DakotaResponseList</td>
<td>153</td>
</tr>
<tr>
<td>dakotaResponseList</td>
<td>61</td>
</tr>
<tr>
<td>SGOPTApplication</td>
<td>325</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>~DakotaResponseRep, 154</td>
</tr>
<tr>
<td>copy_results</td>
<td>155</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>156</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>154, 156</td>
</tr>
<tr>
<td>dataSize</td>
<td>155</td>
</tr>
<tr>
<td>fnTags</td>
<td>155</td>
</tr>
<tr>
<td>functionGradients</td>
<td>155</td>
</tr>
<tr>
<td>functionHessians</td>
<td>155</td>
</tr>
<tr>
<td>functionValues</td>
<td>155</td>
</tr>
<tr>
<td>interfaceId</td>
<td>155</td>
</tr>
<tr>
<td>operator</td>
<td>156</td>
</tr>
<tr>
<td>operator==</td>
<td>156</td>
</tr>
<tr>
<td>overlay</td>
<td>155</td>
</tr>
<tr>
<td>purgeInactive</td>
<td>155</td>
</tr>
<tr>
<td>read_data</td>
<td>155</td>
</tr>
<tr>
<td>referenceCount</td>
<td>155</td>
</tr>
<tr>
<td>reset</td>
<td>155</td>
</tr>
<tr>
<td>responseASV</td>
<td>155</td>
</tr>
<tr>
<td>write_data</td>
<td>155</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>154</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>156</td>
</tr>
<tr>
<td>read</td>
<td>156–158</td>
</tr>
<tr>
<td>read.annotated</td>
<td>157</td>
</tr>
<tr>
<td>read_TABULAR</td>
<td>157</td>
</tr>
<tr>
<td>write</td>
<td>157, 158</td>
</tr>
<tr>
<td>write.annotated</td>
<td>157</td>
</tr>
<tr>
<td>write_TABULAR</td>
<td>157</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>161, 162</td>
</tr>
<tr>
<td>graphicsFlag</td>
<td>161</td>
</tr>
<tr>
<td>iteratorCommunicator</td>
<td>159</td>
</tr>
<tr>
<td>iteratorCommunicatorSize</td>
<td>160</td>
</tr>
<tr>
<td>iteratorComm</td>
<td>160</td>
</tr>
<tr>
<td>iteratorCommRank</td>
<td>160</td>
</tr>
<tr>
<td>iteratorCommSize</td>
<td>160</td>
</tr>
<tr>
<td>mpirunFlag</td>
<td>161</td>
</tr>
<tr>
<td>parallelLib</td>
<td>160</td>
</tr>
<tr>
<td>probDescDB</td>
<td>160</td>
</tr>
<tr>
<td>referenceCount</td>
<td>161</td>
</tr>
<tr>
<td>runStrategy</td>
<td>159</td>
</tr>
<tr>
<td>strategyResponse_results</td>
<td>159</td>
</tr>
<tr>
<td>strategyVariable_results</td>
<td>159</td>
</tr>
<tr>
<td>strategyName</td>
<td>160</td>
</tr>
<tr>
<td>strategyRep</td>
<td>161</td>
</tr>
<tr>
<td>tabularDataFile</td>
<td>161</td>
</tr>
<tr>
<td>tabularDataFlag</td>
<td>161</td>
</tr>
<tr>
<td>worldRank</td>
<td>159</td>
</tr>
<tr>
<td>worldRank</td>
<td>160</td>
</tr>
<tr>
<td>worldSize</td>
<td>160</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>159</td>
</tr>
<tr>
<td>~DakotaStrategy</td>
<td>162</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>161, 162</td>
</tr>
<tr>
<td>freeCommunicators</td>
<td>163</td>
</tr>
<tr>
<td>getStrategy</td>
<td>163</td>
</tr>
<tr>
<td>initCommunicators</td>
<td>163</td>
</tr>
<tr>
<td>initialize_graphics</td>
<td>163</td>
</tr>
<tr>
<td>operator=</td>
<td>162</td>
</tr>
<tr>
<td>probDescDB</td>
<td>164</td>
</tr>
<tr>
<td>run_iterator</td>
<td>162</td>
</tr>
<tr>
<td>run_iterator_repartition</td>
<td>163</td>
</tr>
<tr>
<td>DakotaString</td>
<td>165</td>
</tr>
<tr>
<td>~DakotaString</td>
<td>165</td>
</tr>
<tr>
<td>DakotaString</td>
<td>165</td>
</tr>
<tr>
<td>isNull</td>
<td>165</td>
</tr>
<tr>
<td>operator=</td>
<td>165</td>
</tr>
<tr>
<td>toLower</td>
<td>165</td>
</tr>
<tr>
<td>toUpper</td>
<td>165</td>
</tr>
<tr>
<td>DakotaString</td>
<td>165</td>
</tr>
<tr>
<td>contains</td>
<td>166</td>
</tr>
<tr>
<td>data</td>
<td>166</td>
</tr>
<tr>
<td>lower</td>
<td>166</td>
</tr>
<tr>
<td>operator</td>
<td>166</td>
</tr>
<tr>
<td>testClass</td>
<td>166</td>
</tr>
<tr>
<td>upper</td>
<td>166</td>
</tr>
<tr>
<td>DakotaVarConstraints</td>
<td>169</td>
</tr>
<tr>
<td>all_continuouslower_bounds</td>
<td>169</td>
</tr>
<tr>
<td>all_continuousupper_bounds</td>
<td>169</td>
</tr>
</tbody>
</table>
all_discrete_lower_bounds, 169
discrete_upper_bounds, 169
continuous_lower_bounds, 168
continuous_upper_bounds, 168
DakotaVarConstraints, 172
discrete_lower_bounds, 168
discrete_upper_bounds, 169
discreteFlag, 170
emptyIntVector, 171
emptyRealVector, 171
inactive_continuous_lower_bounds, 169
inactive_continuous_upper_bounds, 169
inactive_discrete_lower_bounds, 169
inactive_discrete_upper_bounds, 169
linear_eq_constraint_coeffs, 170
linear_eq_constraint_targets, 170
linear_ineq_constraint_coeffs, 170
linear_ineq_constraint_targets, 170
linear_ineq_constraint_lower_bounds, 170
linear_ineq_constraint_upper_bounds, 170
linearEqConstraintCoeffs, 171
linearEqConstraintTargets, 171
linearIneqConstraintCoeffs, 170
linearIneqConstraintLowerBnds, 171
linearIneqConstraintUpperBnds, 171
num_active_variables, 170
num_linear_eq_constraints, 170
num_linear_ineq_constraints, 169
numLinearEqConstraints, 170
numLinearIneqConstraints, 170
read, 169
referenceCount, 171
varConstraintsRep, 171
variablesType, 170
write, 169
DakotaVarConstraints, 168
~DakotaVarConstraints, 172
DakotaVarConstraints, 172
get_var_constraints, 173
manageLinearIneqConstraints, 173
operator=, 172
DakotaVariables, 175
acv, 175
adv, 175
all_continuous_variable_labels, 175
all_continuous_variables, 175
all_discrete_variable_labels, 175
discrete_variables, 175
dv, 174
emptyIntVector, 177
emptyRealVector, 177
emptyStringArray, 177
inactive_continuous_variables, 175
inactive_discrete_variables, 175
merged_integer_list, 176
mergedIntegerList, 176
operator =, 177
operator==, 177
read, 175, 176
referenceCount, 176
tv, 174
variablesType, 176
variablesRep, 177
variablesType, 176
write, 176
write.annotated, 176
write_tabular, 176
DakotaVariables, 174
~DakotaVariables, 178
copy, 179
DakotaVariables, 178
get_variables, 179
operator=, 179
DakotaVector, 180
~DakotaVector, 180, 182
DakotaVector, 180, 182
operator T *, 180
operator=, 180
print, 181, 182
print.annotated, 181
print.aprepro, 181
print.partial, 181
print.partial.aprepro, 181
read, 180, 181
read.annotated, 181
read.partial, 181
read.tabular, 181
DakotaVector, 180
DakotaVariables, 182
operator=, 182
testClass, 182
data
DakotaArray, 92
DakotaBaseVector, 96
DakotaString, 166
data_size
DakotaResponse, 152
DakotaResponseRep, 155
DataInterface
~DataInterface, 184
centringParam, 192
centralPath, 191
centralTolerance, 190
coreContractAfterFail, 193
contractFactor, 192
coreConvergenceTolerance, 190
coreCrossoverRate, 192
coreCrossoverType, 193
daceMethod, 194
deltaMethod, 189
deltaMethod, 189
deltasPerVariable, 195
deltaExpansionFlag, 193
deltaExpansionOrder, 194
deltaExpansionTerms, 194
deltaExploratoryMoves, 193
deltaFinalPoint, 194
deltaFixedSeedFlag, 194
deltaFunctionPrecision, 191
deltaGradientTolerance, 191
deltaIdMethod, 189
deltaInitDelta, 192
deltaInterfacePointer, 189
deltaLinearEqConstraintCoeffs, 191
deltaLinearEqTargets, 191
deltaLinearIneqConstraintCoeffs, 191
deltaLinearIneqLowerBnds, 191
deltaLinearIneqUpperBnds, 191
deltaLineSearchTolerance, 191
deltaListOfPoints, 195
deltaMaxCPUTime, 192
deltaMaxFunctionEvaluations, 190
deltaMaxIterations, 190
deltaMaxStep, 191
deltaMeritFn, 191
deltaMethodName, 189
deltaMethodOutput, 190
deltaMinMaxType, 191
deltaModelType, 190
deltaMutationDimRate, 192
deltaMutationMinScale, 192
deltaMutationPopRate, 192
deltaMutationRange, 193
deltaMutationScale, 192
deltaMutationType, 193
deltaNewSolnsGenerated, 192
deltaNonAdaptiveFlag, 193
deltaNumberRetained, 192
deltaNumPartitions, 193
deltaNumSamples, 194
deltaNumSteps, 195
deltaNumSymbols, 194
deltaOperator=, 189
deltaOperator==, 189
optionalInterfaceResponsesPointer, 190
paramStudyType, 194
patternBasis, 193
percentDelta, 195
populationSize, 192
primaryCoeffs, 190
probabilityLevels, 194
randomizeOrderFlag, 193
randomSeed, 194
read, 189
reliabilityMethod, 194
replacementType, 193
responseLevels, 194
responsesPointer, 190
responseThresholds, 194
sampleType, 194
searchMethod, 191
searchSchemeSize, 192
secondaryCoeffs, 190
selectionPressure, 193
solnAccuracy, 192
speculativeFlag, 192
stepLength, 195
stepLenToBoundary, 191
stepVector, 194
subMethodPointer, 190
threshDelta, 192
totalPatternSize, 193
variablesPointer, 189
varPartitions, 193
verifyLevel, 191
write, 189
DataMethod, 189
DataResponses

~DataResponses, 196
assign, 198
DataResponses, 196
fdStepSize, 197
gradientType, 197
hessianType, 197
idAnalytic, 197
idNumerical, 197
idResponses, 197
intervalType, 197
methodSource, 197
multiObjectiveWeights, 197
nonlinearEqTargets, 197
nonlinearIneqLowerBnds, 197
nonlinearIneqUpperBnds, 197
numLeastSqTerms, 196
numNonlinearEqConstraints, 196
numNonlinearIneqConstraints, 196
numObjectiveFunctions, 196
numResponseFunctions, 197
operator=, 196
operator==, 196
read, 196
responseLabels, 198
write, 196
DataResponses, 196
DataStrategy

~DataStrategy, 199
assign, 201
branchBndNumSamplesNode, 200
branchBndNumSamplesRoot, 200
concurrentParameterSets, 201
concurrentRandomJobs, 201
concurrentSeed, 201
DataStrategy, 199
graphicsFlag, 199
iteratorScheduling, 200
iteratorServers, 199
methodPointer, 200
multilevelGlobalMethodPointer, 200
multilevelLocalMethodPointer, 200
multilevelLSProb, 200
multilevelMethodList, 200
multilevelProgThresh, 200
multilevelType, 200
operator=, 199
read, 199
strategyType, 199
surrBasedOptConvTol, 200
surrBasedOptMaxIterations, 200
surrBasedOptSoftConvLimit, 200
surrBasedOptTRContract, 201
surrBasedOptTRContractTrigger, 201
surrBasedOptTRExpand, 201
surrBasedOptTRExpandTrigger, 201
surrBasedOptTRInitSize, 201
surrBasedOptTRMinSize, 201
tabularDataFile, 199
tabularDataFlag, 199
write, 199
DataStrategy, 199
DataVariables

~DataVariables, 203
assign, 207
continuousDesignLabels, 205
continuousDesignLowerBnds, 204
continuousDesignUpperBnds, 204
continuousDesignVars, 204
continuousStateLabels, 207
continuousStateLowerBnds, 207
continuousStateUpperBnds, 207
continuousStateVars, 206
DataVariables, 203
design, 203
INDEX

DirectFnApplicInterface, 208

derived_master_overload
DakotaModel, 129
HierLayeredModel, 239
NestedModel, 269
SingleModel, 333
SurrLayeredModel, 358
derived_synch
ApplicationInterface, 45
DirectFnApplicInterface, 208
ForkApplicInterface, 219
SysCallApplicInterface, 364
derived_synch_kernel
ForkApplicInterface, 220
SysCallApplicInterface, 365
derived_synch_nowait
ApplicationInterface, 45
DirectFnApplicInterface, 208
ForkApplicInterface, 219
SysCallApplicInterface, 364
derived_synchronize
DakotaModel, 133
HierLayeredModel, 240
NestedModel, 272
SingleModel, 333
SurrLayeredModel, 357
derived_synchronize_nowait
DakotaModel, 133
HierLayeredModel, 240
NestedModel, 272
SingleModel, 333
SurrLayeredModel, 358
derived_synchronous_local_analysis
ApplicationInterface, 45
DirectFnApplicInterface, 208
ForkApplicInterface, 219
SysCallApplicInterface, 364
design
DataVariables, 203
designModel
NonDOptStrategy, 287

DF
CONMINOptimizer, 77
KrigApprox, 248

DirectFnApplicInterface
~DirectFnApplicInterface, 208
cantilever, 210
cyl_head, 210
derived_map, 208
derived_map_ac, 209
derived_map_asynch, 208
derived_map_f, 209
derived_map_of, 209
derived_synch, 208
derived_synch_nowait, 208
derived_synchronous_local_analysis, 208
DirectFnApplicInterface, 208
directFnASV, 210
directFnResponse, 210
directFnVars, 210
fnGrads, 210
fnHessians, 210
fnVals, 209
gradFlag, 209
hessFlag, 209
iFilterName, 209
mc_api_run, 211
numFns, 209
numGradVars, 209
numVars, 209
oFilterName, 209
overlay_response, 209
pxcFile, 209
rosenbrock, 210
salinas, 211
set_local_data, 209
text_book, 210
text_book1, 210
text_book2, 210
text_book3, 210
text_book_ouu, 210
xVect, 209

Discrete
AllMergedVarConstraints, 26
AllVarConstraints, 32
DakotaModel, 131
DakotaVarConstraints, 169
FundamentalVarConstraints, 223
MergedVarConstraints, 259

discrete_lower_bounds
AllMergedVarConstraints, 26
AllVarConstraints, 32
DakotaModel, 131
DakotaVarConstraints, 168
FundamentalVarConstraints, 223
MergedVarConstraints, 259

discrete_upper_bounds
AllMergedVarConstraints, 26
AllVarConstraints, 32
DakotaModel, 131
DakotaVarConstraints, 169
FundamentalVarConstraints, 223
MergedVarConstraints, 259

discrete_variable_labels
AllMergedVariables, 29
AllVariables, 36
DakotaModel, 130
DakotaVariables, 175
FundamentalVariables, 228
MergedVariables, 263
discreteVariables
AllMergedVariables, 28
AllVariables, 35
DakotaModel, 130
DakotaVariables, 175
FundamentalVariables, 227
MergedVariables, 262
discreteAppFlag
SGOPTOptimizer, 328
discreteDesignLabels
DataVariables, 205
FundamentalVariables, 229
discreteDesignLowerBnds
DataVariables, 204
FundamentalVarConstraints, 225
discreteDesignUpperBnds
DataVariables, 205
FundamentalVarConstraints, 225
discreteDesignVars
DataVariables, 204
FundamentalVariables, 229
discreteFlag
DakotaVarConstraints, 170
discreteStateLabels
DataVariables, 207
FundamentalVariables, 230
discreteStateLowerBnds
DataVariables, 207
FundamentalVarConstraints, 225
discreteStateUpperBnds
DataVariables, 207
FundamentalVarConstraints, 225
discreteStateVars
DataVariables, 207
FundamentalVariables, 229
DoEval
COLINApplication, 62
SGOPTApplication, 326
dotFDSinfo
DOTOptimizer, 214
dotInfo
DOTOptimizer, 214
dotMethod
DOTOptimizer, 214
DOTOptimizer, 212
~DOTOptimizer, 212
allocate_workspace, 212
constraintMappingIndices, 215
constraintMappingMultipliers, 215
constraintMappingOffsets, 215
dotFDSinfo, 214
dotInfo, 214
dotMethod, 214
DOTOptimizer, 212
find_optimum, 212
intCntlParmArray, 214
intWorkSpace, 213
intWorkSpaceSize, 213
localConstraintValues, 215
optimizationType, 214
printControl, 214
realCntlParmArray, 214
realWorkSpace, 213
realWorkSpaceSize, 213
draw_surface
DakotaApproximation, 86
dummyFlag
ParallelLibrary, 302
duplication_detect
ApplicationInterface, 52
dv
AllMergedVariables, 28
AllVariables, 35
DakotaModel, 130
DakotaVariables, 174
FundamentalVariables, 227
MergedVariables, 262
emptyIntVector
DakotaVarConstraints, 171
DakotaVariables, 177
emptyRealVector
DakotaVarConstraints, 171
DakotaVariables, 177
emptyStringArray
DakotaVariables, 177
enroll
GetLongOpt, 234
enrollDone
GetLongOpt, 233
entries
DakotaList, 122
ePSAOptimizer
SGOPTOptimizer, 329
erfInverse
NonDAdvMeanValue, 279
errofstream
ParallelLibrary, 302
ErrorTable
msg, 216
rc, 216
ErrorTable, 216
eval_analysis_communicator_rank
ParallelLibrary, 301
eval_analysis_communicator_size
ParallelLibrary, 301
Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2002
eval_analyses_inter_communicator
    ParallelLibrary, 301
eval_analyses_inter_communicators
    ParallelLibrary, 301
eval_analyses_intra_communicator
    ParallelLibrary, 301
eval_analyses_message_pass
    ParallelLibrary, 301
eval_analyses_split_flag
    ParallelLibrary, 301
eval_id
    ParamResponsePair, 310
evalAnalysisCommRank
    ParallelLibrary, 305
evalAnalysisCommSize
    ParallelLibrary, 305
evalAnalysisInterComm
    ParallelLibrary, 305
evalAnalysisIntraComms
    ParallelLibrary, 305
evalAnalysisIntraComm
    ParallelLibrary, 305
evalAnalysisMessagePass
    ParallelLibrary, 305
evalAnalysisSplitFlag
    ParallelLibrary, 305
evalCacheFlag
    ApplicationInterface, 49
    DataInterface, 186
evalComm
    ApplicationInterface, 47
    ParallelLibrary, 304
evalCommRank
    ApplicationInterface, 46
    ParallelLibrary, 304
evalCommSize
    ApplicationInterface, 46
    ParallelLibrary, 304
evalDedicatedMasterFlag
    ParallelLibrary, 305
evalDedMasterFlag
    ApplicationInterface, 46
evalId
    ParamResponsePair, 311
evalIdList
    ForkApplicInterface, 220
    HierLayeredModel, 239
evalIntraComm
    ParallelLibrary, 304
evalMasterFlag
    ParallelLibrary, 304
evalMessagePass
    ParallelLibrary, 304
evalScheduling
    ApplicationInterface, 46
    DataInterface, 186
    ParallelLibrary, 305
evalServerId
    ApplicationInterface, 46
    ParallelLibrary, 305
evalServers
    DataInterface, 186
evaluate_parameter_sets
    DakotIterator, 118
evaluation_communicator_rank
    ParallelLibrary, 300
evaluation_communicator_size
    ParallelLibrary, 300
evaluation_dedicated_master_flag
    ParallelLibrary, 301
evaluation_intra_communicator
    ParallelLibrary, 300
evaluation_server_id
    ParallelLibrary, 300
evaluation_servers
    ParallelLibrary, 300
expandAfterSuccess
    DataMethod, 193
expansionFlag
    DataMethod, 193
expansionOrder
    DataMethod, 194
expansionTerms
    DataMethod, 194
exploratoryMoves
    DataMethod, 193
    SGOPTOptimizer, 328
failarray
    KrigingSurf, 250
failAction
    ApplicationInterface, 49
    DataInterface, 186
failCountList
    SysCallApplicInterface, 365
failIdList
    SysCallApplicInterface, 365
failRecoveryFnVals
    ApplicationInterface, 49
failRetryLimit
    ApplicationInterface, 49
fd_gradients
    DakotaModel, 137
FDCH
    CONMINOptimizer, 73
    KrigApprox, 243
    FDCHM
CONMINOptimizer, 73
KrigApprox, 243
findf1
SNLLOptimizer, 342
findf1Con
SNLLOptimizer, 342
fdStepSize
DataResponses, 197
fileNameKey
AnalysisCode, 41
fileSaveFlag
AnalysisCode, 40
DataInterface, 185
fileTagFlag
AnalysisCode, 40
DataInterface, 185
finalPoint
DataMethod, 194
ParamStudy, 313
finalStatistics
DakotaNonD, 141
find
DakotaList, 124
find_coefficients
ANNSurf, 42
DakotaApproximation, 86
HermiteSurf, 236
KrigingSurf, 250
MARRSurf, 257
RespSurf, 321
TaylorSurf, 366
find optimum
COLINOptimizer, 65
CONMINOptimizer, 72
DakotaOptimizer, 143
DOTOptimizer, 212
NPSOLOptimizer, 294
rSQPOptimizer, 323
SGOPTOptimizer, 330
SNLLOptimizer, 341
find optimum_on_model
NPSOLOptimizer, 294
find optimum_on_user_functions
NPSOLOptimizer, 294
finiteDiffSS
DakotaModel, 135
finiteDiffStepSize
DakotatIterator, 116
finiteDiffType
DakotatIterator, 115
fitInactiveCLowerBnds
LayeredModel, 253
fitInactiveCUpperBnds
LayeredModel, 253
fitInactiveCVars
LayeredModel, 253
fitInactiveDLowerBnds
LayeredModel, 253
fitInactiveDUpperBnds
LayeredModel, 253
fitInactiveDVars
LayeredModel, 253
fixedSeedFlag
DataMethod, 194
flags
MARRSurf, 257
flush
CommandShell, 69
fn_tags
DakotaResponse, 150
fnEvalCntr
SOLBase, 348
fnEvalId
DakotaInterface, 109
fnGrads
DirectFnApplicInterface, 210
fnHessians
DirectFnApplicInterface, 210
fnTags
DakotaResponseRep, 155
fnVals
DirectFnApplicInterface, 209
force_rebuild
LayeredModel, 255
fork_application
ForkApplicInterface, 217
fork_program
ForkAnalysisCode, 217
ForkAnalysisCode
~ForkAnalysisCode, 217
argList, 217
argument_list, 217
fork_program, 217
ForkAnalysisCode, 217
tag_argument_list, 217
ForkAnalysisCode, 217
tag_argument_list, 217
check_status, 218
ForkApplicInterface
~ForkApplicInterface, 219
derived_map, 219
derived_map_async, 219
derived_synch, 219
derived_synch_kernel, 220
derived_synch_nowait, 219
derived_synchronous_local_analysis, 219
evalIdList, 220
ForkApplicInterface, 219
forkSimulator, 220
processIdList, 220
ForkApplicInterface, 219
asynchronous_local_analyses, 221
fork_application, 220
serve_analyses_async, 221
synchronous_local_analyses, 221
forkSimulator
ForkApplicInterface, 220
free_analysis_communicators
ParallelLibrary, 297
free_communicators
ApplicationInterface, 44
DakotaInterface, 108
DakotaModel, 129
DakotaStrategy, 163
HierLayeredModel, 239
NestedModel, 270
SingleModel, 334
SurfLayeredModel, 356
free_evaluation_communicators
ParallelLibrary, 297
free_iterator_communicators
ParallelLibrary, 297
fRinvVector
KrigApprox, 246
function_gradients
DakotaResponse, 151
function_hessians
DakotaResponse, 151
function_values
DakotaResponse, 150, 151
FunctionCompare
FunctionCompare, 222
operator(), 222
search_val, 222
testFunction, 222
FunctionCompare, 222
functionGradients
DakotaResponseRep, 155
functionHessians
DakotaResponseRep, 155
functionPrecision
DataMethod, 191
functionSurfaceCoeffs
ApproximationInterface, 56
functionSurfaces
ApproximationInterface, 57
functionValues
DakotaResponseRep, 155
FundamentalVarConstraints
~FundamentalVarConstraints, 223
all_continuous_variable_labels, 228
all_continuous_variables, 228
all_discrete_variable_labels, 228
all_discrete_variables, 228
continuous_variable_labels, 227, 228
continuous_variables, 227
inactive_continuous_variables, 228
inactive_continuous_variable_labels, 228
inactive_continuous_upper_bounds, 224
inactive_designLabels, 229
inactive_designVars, 229
cv, 227
dv, 227
discrete_variable_labels, 228
discrete_variables, 227
discreteDesignLabels, 229
discreteDesignVars, 229
designLabels, 230
designVars, 229
dv, 227
FundamentalVariables, 227, 230
read, 228
uncertainDistLowerBnds, 225
uncertainDistUpperBnds, 225
write, 224
FundamentalVarConstraints, 223
FundamentalVarConstraints, 225
FundamentalVariables
~FundamentalVariables, 227
dv, 227
adv, 228
acv, 228
all_continuous_variable_labels, 228
all_continuous_variables, 228
all_discrete_variable_labels, 228
all_discrete_variables, 228
continuous_variable_labels, 227, 228
continuous_variables, 227
inactive_continuous_variables, 228
inactive_continuous_variable_labels, 228
inactive_continuous_upper_bounds, 224
inactive_designLabels, 229
inactive_designVars, 229
nonDFlag, 224
read, 224
uncertainDistLowerBnds, 225
uncertainDistUpperBnds, 225
write, 224
FundamentalVarConstraints, 223
FundamentalVarConstraints, 225
FundamentalVariables
~FundamentalVariables, 227
INDEX 421

read_annotated, 228

KrigApprox, 246

G1
CONMINOptimizer, 77
G2
CONMINOptimizer, 77
gammaContract
SurrBasedOptStrategy, 351
gammaExpand
SurrBasedOptStrategy, 351
gammaNoChange
SurrBasedOptStrategy, 351
get
DakotaList, 123
DakotaApproximation, 88
getbool
ProblemDescDB, 317
getchaos
HermiteSurf, 236
getdb_list_nodes
ProblemDescDB, 316
getdia
ProblemDescDB, 317
getdil
ProblemDescDB, 317
getdll
ProblemDescDB, 317
getdlliv
ProblemDescDB, 317
getdlra
ProblemDescDB, 317
getdlrm
ProblemDescDB, 317
getdlrv
ProblemDescDB, 317
getdllva
ProblemDescDB, 317
getdllsa
ProblemDescDB, 317
getdllsl
ProblemDescDB, 317
getgradient
DakotaApproximation, 85
RespSurf, 321
TaylorSurf, 366
getint
ProblemDescDB, 317
getinterface
DakotaInterface, 111
getiterator
DakotaIterator, 119
getmin_point
COLINOptimizer, 64
getmodel
DakotaModel, 137
getnum_chaos
HermiteSurf, 236
getreal
ProblemDescDB, 317
getsize
ProblemDescDB, 317
get_source_pair
ApplicationInterface, 48
get_strategy
DakotaStrategy, 163
get_string
ProblemDescDB, 317
get_value
ANNSurf, 42
DakotaApproximation, 85
HermiteSurf, 236
KrigingSurf, 250
MARRSurf, 257
RespSurf, 321
TaylorSurf, 366
get_var_constraints
DakotaVarConstraints, 173
get_variables
DakotaVariables, 179
GetLongOpt
~GetLongOpt, 232
basename, 233
enroll_done, 233
GetLongOpt, 233
last, 233
optmarker, 233
pname, 233
setcell, 233
table, 233
usage, 232
ustring, 233
GetLongOpt, 232
enroll, 234
GetLongOpt, 233
parse, 234
retrieve, 234
usage, 234

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (©) 1997-2002
getRe
   CtelRegexp, 79
getStatus
   CtelRegexp, 79
getStatusMsg
   CtelRegexp, 79
globalApproxFlag
   SurrBasedOptStrategy, 352
gradFlag
   DirectFnApplicInterface, 209
gradient
   concurrency
   DakotaModel, 132
gradient
   method
   DakotaModel, 132
gradient
   Flag
   DakotaApproximation, 86
   SurrBasedOptStrategy, 352
gradientTolerance
   DataMethod, 191
gradient
   Type
   DakotaIterator, 115
   DataResponses, 197
gradType
   DakotaModel, 135
gradVector
   DakotaApproximation, 86
graphics
   2D
   DakotaGraphics, 104
graphics
   Cntr
   DakotaGraphics, 104
graphics
   Flag
   ApproximationInterface, 56
   DakotaStrategy, 161
   DataStrategy, 199
gridHostNames
   DataInterface, 185
gridProcsPerHost
   DataInterface, 185
hard
   convergence
   check
   SurrBasedOptStrategy, 353
headerFlag
   ApplicationInterface, 49
HermiteSurf
   ~HermiteSurf, 236
   approximation
   coefficients, 236
chaosCoeffs, 237
chaosSamples, 237
find
   coefficients, 236
get
   chaos, 236
get
   num
   chaos, 236
get
   value, 236
HermiteSurf, 236
highestOrder, 237
numChaos, 237
required
   samples, 236
HermiteSurf, 236
hessFlag
   DirectFnApplicInterface, 209
hessian
   Type
   DakotaIterator, 115
   DataResponses, 197
hierachApproxFlag
   SurrBasedOptStrategy, 352
HierLayeredModel
   ~HierLayeredModel, 238
build
   approximation, 238
derived
   init
   communicators, 239
derived
   master
   overload, 239
evalIdList, 239
free
   communicators, 239
HierLayeredModel, 238
highFidelityModel, 239
highFidelityResponse, 239
local
   eval
   synchronization, 238
lowFidelityInterface, 239
new
   eval
   counter, 239
serve, 239
stop
   servers, 239
subordinate
   model, 238
synchronize
   nowait
   completions, 239
total
   eval
   counter, 239
HierLayeredModel, 238
derived
   asynch
   compute
   response, 240
derived
   compute
   response, 240
derived
   synchronize, 240
derived
   synchronize
   nowait, 240
highestOrder
   HermiteSurf, 237
   NonDPCESampling, 289
highFidelityInterface
   Ptr
   DataInterface, 187
highFidelityModel
   HierLayeredModel, 239
highFidelityResponse
   HierLayeredModel, 239
histogram
   BinPairs
   DakotaNonD, 140
histogram
   PointPairs
   DakotaNonD, 140
histogram
   UncBinPairs
   DataVariables, 206
histogram
   UncPointPairs
   DataVariables, 206
historyDuplicate
   Ids
   ApplicationInterface, 49
historyDuplicate
   Responses
   ApplicationInterface, 49
<table>
<thead>
<tr>
<th>IC</th>
<th>CONMINOptimizer, 78</th>
<th>DakotaVarConstraints, 169</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KrigApprox, 249</td>
<td>FundamentalVarConstraints, 224</td>
</tr>
<tr>
<td>ICNDIR</td>
<td>KrigApprox, 245</td>
<td>MergedVarConstraints, 260</td>
</tr>
<tr>
<td></td>
<td>idAnalytic</td>
<td>inactive discrete upper bounds</td>
</tr>
<tr>
<td></td>
<td>DakotaModel, 135</td>
<td>AllMergedVarConstraints, 26</td>
</tr>
<tr>
<td></td>
<td>DataResponses, 197</td>
<td>AllVarConstraints, 33</td>
</tr>
<tr>
<td>idInterface</td>
<td>DataInterface, 184</td>
<td>DakotaModel, 131</td>
</tr>
<tr>
<td>idMethod</td>
<td>DataMethod, 189</td>
<td>DakotaVarConstraints, 169</td>
</tr>
<tr>
<td>idNumerical</td>
<td>DataResponses, 197</td>
<td>FundamentalVarConstraints, 224</td>
</tr>
<tr>
<td>idResponses</td>
<td>DataResponses, 197</td>
<td>MergedVarConstraints, 260</td>
</tr>
<tr>
<td>idrKeywordTable</td>
<td>keywordtable.C, 369</td>
<td></td>
</tr>
<tr>
<td>idVariables</td>
<td>DataVariables, 204</td>
<td></td>
</tr>
<tr>
<td>iFilterName</td>
<td>AnalysisCode, 40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DirectFinApplicInterface, 209</td>
<td></td>
</tr>
<tr>
<td>iFlag</td>
<td>KrigApprox, 249</td>
<td></td>
</tr>
<tr>
<td>IGOTO</td>
<td>KrigApprox, 245</td>
<td></td>
</tr>
<tr>
<td>impFactor</td>
<td>NonDAdvMeanValue, 280</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inactive continuous lower bounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllMergedVarConstraints, 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllVarConstraints, 32, 33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaModel, 131</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaVarConstraints, 169</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FundamentalVarConstraints, 223, 224</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MergedVarConstraints, 259, 260</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inactive continuous upper bounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllMergedVarConstraints, 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllVarConstraints, 33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaModel, 131</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaVarConstraints, 169</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FundamentalVarConstraints, 224</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MergedVarConstraints, 260</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inactive continuous variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllMergedVariables, 29</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllVariables, 36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaModel, 131</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaVariables, 175</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FundamentalVariables, 228</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MergedVariables, 263</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inactive discrete lower bounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllMergedVarConstraints, 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllVarConstraints, 33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DakotaModel, 131</td>
<td></td>
</tr>
</tbody>
</table>

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (©) 1997-2002
insert DakotaList, 124
insert node
ProblemDescDB, 317, 318
intCntlParmArray
DOTOptimizer, 214
interface id
DakotaResponse, 150
ParamResponsePair, 310
interface kwhandler
ProblemDescDB, 318
interface synchronization
ApplicationInterface, 44
DakotaInterface, 108
interface ptype
DakotaInterface, 108
interfaceId
DakotaResponseRep, 155
interfaceIter
ProblemDescDB, 319
interfaceList
ProblemDescDB, 319
interfacePointer
DataMethod, 189
NestedModel, 271
interfaceRep
DakotaInterface, 110
interfaceResponse
NestedModel, 271
interfaceSynchronization
ApplicationInterface, 49
DataInterface, 185
interfaceType
DakotaInterface, 109
DataInterface, 184
intervalType
DakotaModel, 135
DataResponses, 197
intProblem
SGOPTOptimizer, 329
int WorkSpace
DOTOptimizer, 213
SOLBase, 347
int WorkSpaceSize
DOTOptimizer, 213
SOLBase, 347
invcorrelMatrix
KrigApprox, 246
iPivotVector
KrigApprox, 246
IPRINT
CONMINOptimizer, 73
KrigApprox, 243
irecv ea
ParallelLibrary, 298
irecv ie
ParallelLibrary, 298
irecv si
ParallelLibrary, 298
isNull
DakotaIterator, 114
DakotaModel, 132
ISC
CONMINOptimizer, 78
KrigApprox, 248
isEmpty
DakotaList, 123
isend ea
ParallelLibrary, 298
isend ie
ParallelLibrary, 298
isend si
ParallelLibrary, 298
isNull
DakotaString, 165
ITER
KrigApprox, 245
iterated mean value
NonDAdvMeanValue, 278
iterator communicator
DakotaStrategy, 159
iterator communicator rank
ParallelLibrary, 299
iterator communicator size
DakotaStrategy, 160
ParallelLibrary, 299
iterator dedicated master flag
DakotaInterface, 109
ParallelLibrary, 300
iterator eval communicator rank
ParallelLibrary, 300
iterator eval communicator size
ParallelLibrary, 300
iterator eval inter communicator
ParallelLibrary, 300
iterator eval inter communicators
ParallelLibrary, 300
iterator eval intra communicator
ParallelLibrary, 300
iterator eval message pass
ParallelLibrary, 300
iterator eval split flag
ParallelLibrary, 300
iterator intra communicator
ParallelLibrary, 299
iterator master flag
ParallelLibrary, 299
iterator response results
INDEX

DACEIterator, 81
DakotaIterator, 113
DakotaNonD, 139
DakotaOptLeastSq, 146
ParamStudy, 312
iterator_server_id
ParallelLibrary, 300
iterator_servers
ParallelLibrary, 299
iterator_variable_results
DACEIterator, 81
DakotaIterator, 113
DakotaOptLeastSq, 146
ParamStudy, 312
iteratorComm
DakotaStrategy, 160
iteratorCommRank
ApplicationInterface, 46
DakotaStrategy, 160
ParallelLibrary, 303
iteratorCommSize
ApplicationInterface, 46
DakotaStrategy, 160
ParallelLibrary, 304
iteratorDedicatedMasterFlag
ParallelLibrary, 304
iteratorDedMasterFlag
DakotaInterface, 109
iteratorEvalCommRank
ParallelLibrary, 304
iteratorEvalCommSize
ParallelLibrary, 305
iteratorEvalInterComm
ParallelLibrary, 304
iteratorEvalInterComms
ParallelLibrary, 304
iteratorEvalIntraComm
ParallelLibrary, 304
iteratorEvalMessagePass
ParallelLibrary, 304
iteratorEvalSplitFlag
ParallelLibrary, 304
iteratorIntraComm
ParallelLibrary, 303
iteratorMasterFlag
ParallelLibrary, 303
iteratorRep
DakotaIterator, 117
iteratorScheduling
DataStrategy, 200
iteratorServerId
ConcurrentStrategy, 71
ParallelLibrary, 304
iteratorServers
DataStrategy, 199
iterMax
SurrBasedOptStrategy, 351
ITMAX
CONMINOptimizer, 73
KrigApprox, 243
ITRM
KrigApprox, 244
ivec
ColinPoint, 66
iworkVector
KrigApprox, 246
jacUToX
NonDAdvMeanValue, 284
jacXToU
NonDAdvMeanValue, 283
jacXToZ
NonDAdvMeanValue, 283
jacZToX
NonDAdvMeanValue, 283

keywordtable.C, 369
iworkKeywordTable, 369
KrigApprox
~KrigApprox, 242
ABOBJ1, 244
ALPHAX, 244
betaHat, 245
comminInfo, 243
comminSingleArray, 242
comminThetaLowerBnds, 244
comminThetaUpperBnds, 244
comminThetaVars, 244
constraintVector, 246
correlationMatrix, 246
CTL, 243
CTLMIN, 243
CTMIN, 243
DABFUN, 243
DELFUN, 243
FDCH, 243
FDCHM, 243
fRinvVector, 246
fValueVector, 246
ICNDIR, 245
IGOTO, 245
INFOG, 245
invcorrelMatrix, 246
iPivotVector, 246
IPRINT, 243
ITER, 245
ITMAX, 243
ITRM, 244

Generated on Mon Apr 7 21:27:12 2003 for DAKOTA by Doxygen written by Dimitri van Heesch (c) 1997-2002
DataVariables, 206
lower
DakotaString, 166
lowerBounds
NPSOLOptimizer, 295
lowFidelityInterface
HierLayeredModel, 239
lowFidelityInterfacePtr
DataInterface, 187
main
main.C, 370
restart_util.C, 372
main.C, 370
main, 370
write_precision, 370
manage_asv
DakotaModel, 138
manage_failure
ApplicationInterface, 44
manage_inputs
ProblemDescDB, 320
manage_linear_constraints
DakotaVarConstraints, 173
manage_outputs_restart
ParallelLibrary, 307
map
ApplicationInterface, 50
ApproximationInterface, 55
DakotaInterface, 107
map_response
COLINAApplication, 63
marsObject
MARSSurf, 257
MARSSurf, 257
~MARSSurf, 257
find_coefficients, 257
flags, 257
get_value, 257
marsObject, 257
MARSSurf, 257
required_samples, 257
match
CtelRegexp, 79
maxConcurrency
DakotaIterator, 115
maxCPUTime
DataMethod, 192
maxFunctionEvals
DakotaIterator, 115
maxFunctionEvaluations
DataMethod, 190
maximum_concurrency
DakotaIterator, 114
DakotaModel, 128
SurrLayeredModel, 358
maxIterations
DakotaIterator, 115
DataMethod, 190
maxLikelihoodEst
KrigApprox, 245
maxStep
DataMethod, 191
me_api_run
DirectFnApplicInterface, 211
mean95CIDeltas
DakotaNonD, 141
mean_value
NonDAAdvMeanValue, 278
meanStats
DakotaNonD, 141
medianFnVals
NonDAAdvMeanValue, 280
merged_integer_list
DakotaModel, 132
DakotaVariables, 176
mergedDesignLabels
MergedVariables, 264
mergedDesignLowerBnds
MergedVarConstraints, 260
mergedDesignUpperBnds
MergedVarConstraints, 260
mergedDesignVars
MergedVariables, 264
mergedIntegerList
DakotaVariables, 176
mergedStateLabels
MergedVariables, 264
mergedStateLowerBnds
MergedVarConstraints, 261
mergedStateUpperBnds
MergedVarConstraints, 261
mergedStateVars
MergedVariables, 264
MergedVarConstraints
~MergedVarConstraints, 259
all_continuous_lower_bounds, 260
all_continuous_upper_bounds, 260
all_discrete_lower_bounds, 260
all_discrete_upper_bounds, 260
continuous_lower_bounds, 259
continuous_upper_bounds, 259
discrete_lower_bounds, 259
discrete_upper_bounds, 259
inactive_continuous_lower_bounds, 259, 260
inactive_continuous_upper_bounds, 260
inactive_discrete_lower_bounds, 260
inactive_discrete_upper_bounds, 260
inactive_discrete_upper_bounds, 260
mergedDesignLowerBnds, 260
mergedDesignUpperBnds, 260
mergedStateLowerBnds, 261
mergedStateUpperBnds, 261
MergedVarConstraints, 261
read, 260
uncertainDistLowerBnds, 261
uncertainDistUpperBnds, 261
write, 260
MergedVarConstraints, 259
MergedVarConstraints, 261
MergedVariables
~MergedVariables, 262
acv, 263
adv, 263
all_continuous_variable_labels, 263
all_continuous_variables, 263
all_discrete_variable_labels, 263
all_discrete_variables, 263
continuous_variable_labels, 262, 263
continuous_variables, 262
copy_rep, 264
cv, 262
discrete_variable_labels, 263
discrete_variables, 262
dv, 262
inactive_continuous_variables, 263
inactive_discrete_variables, 263
mergedDesignLabels, 264
mergedDesignVars, 264
mergedStateLabels, 264
mergedStateVars, 264
MergedVariables, 262, 265
operator==, 264
read, 263, 264
read_annotated, 263
tv, 262
uncertainLabels, 264
uncertainVars, 264
write, 263, 264
write_annotated, 263
MergedVariables, 262
MergedVariables, 265
meritFn
DataMethod, 191
message_lengths
DakotaModel, 132
messageLengths
DakotaModel, 134
method_khandler
ProblemDescDB, 318
method_name
DakotaIterator, 114
methodIter
ProblemDescDB, 319
methodList
MultilevelOptStrategy, 267
ProblemDescDB, 319
methodName
DakotaIterator, 115
DataMethod, 189
methodOutput
DataMethod, 190
methodPointer
DataStrategy, 200
methodSource
DakotaIterator, 116
DataResponses, 197
methodSrc
DakotaModel, 135
mfcn
SNLLBase, 336
minimize_residuals
DakotaLeastSq, 120
NLSSOLLeastSq, 275
SNLLLeastSq, 338
minimum_samples
ApproximationInterface, 55
DakotaInterface, 108
minMaxType
DataMethod, 191
minSamples
ApproximationInterface, 56
minTrustRegionFactor
SurrBasedOptStrategy, 351
mixedGradAnalyticIds
DakotaIterator, 116
mixedGradNumericIds
DakotaIterator, 116
model_rSQPOptimizer, 323
modelType
DakotaModel, 132
ModelApply
KrigApprox, 247
modelAutoGraphicsFlag
DakotaModel, 134
ModelBuild
KrigApprox, 247
modelCenterFile
DataInterface, 185
modelRep
DakotaModel, 134
modelType
DakotaModel, 134
DataMethod, 190
modified_parameters_filename
AnalysisCode, 39
modified_res_filename
AnalysisCode, 39
modifiedParamsFileName
AnalysisCode, 40
modifiedResFileName
AnalysisCode, 40
mostProbPointU
NonDAAdvMeanValue, 281
mostProbPointX
NonDAAdvMeanValue, 281
mpirun_flag
ParallelLibrary, 299
mpirunFlag
DakotaStrategy, 161
ParallelLibrary, 302
MS1
CONMINOptimizer, 77
KrigApprox, 248
msg
ErrorTable, 216
multiobjective_modify
DakotaOptimizer, 145
multiobjective_weights
DakotaIterator, 114
DakotaOptimizer, 143
multiProcEvalFlag
DakotaInterface, 109
multidim_loop
ParamStudy, 313
multilevelGlobalMethodPointer
DataStrategy, 200
multilevelLocalMethodPointer
DataStrategy, 200
multilevelLSProb
DataStrategy, 200
multilevelMethodList
DataStrategy, 200
MultilevelOptStrategy
DataStrategy, 200
~MultilevelOptStrategy, 266
localSearchProb, 267
methodList, 267
MultilevelOptStrategy, 266
multiLevelType, 267
numIterators, 267
progressMetric, 267
progressThreshold, 267
run_strategy, 266
selectedIterators, 267
strategy_response_results, 266
strategy_variable_results, 266
userDefinedModels, 267
MultilevelOptStrategy, 266
run_coupled, 267
run_uncoupled, 268
run_uncoupledadaptive, 268
multilevelProgThresh
DataStrategy, 200
multiLevelType
MultilevelOptStrategy, 267
multiLevelType
DataStrategy, 200
multiObjectiveWeights
DataResponses, 197
multiobjModifyPtr
COLINApplication, 62
SGOPTApplication, 326
multiObjWeights
DACEIterator, 83
DakotaOptimizer, 144
ParamStudy, 314
multiObjWts
SurfBasedOptStrategy, 352
multiProcAnalysisFlag
ApplicationInterface, 47
multiProcEvalFlag
DakotaInterface, 109
multiStartFlag
ConcurrentStrategy, 70
mutationDimRate
DataMethod, 192
mutationMinScale
DataMethod, 192
mutationPopRate
DataMethod, 192
mutationRange
DataMethod, 193
mutationScale
DataMethod, 192
mutationType
DataMethod, 193
N1
CONMINOptimizer, 76
KrigApprox, 247
N2
CONMINOptimizer, 76
KrigApprox, 247
N3
CONMINOptimizer, 76
KrigApprox, 247
N4
CONMINOptimizer, 76
KrigApprox, 247
N5
CONMINOptimizer, 76
KrigApprox, 247
NAC
KrigApprox, 245
NACMX1
KrigApprox, 244
nestedFlag
ParamStudy, 313
NestedModel
~NestedModel, 269
asv_mapping, 270
completionList, 271
constrCoeffs, 271
derived_master_overload, 269
free_communicators, 270
interfacePointer, 271
interfaceResponse, 271
NestedModel, 269
new_eval_counter, 270
numInterfEqConstr, 271
numInterfIneqConstr, 271
numInterfObjFns, 271
numSubIteratorEqConstr, 270
numSubIteratorIneqConstr, 270
objCoeffs, 271
optionalInterface, 271
responseArray, 271
responseList, 271
serve, 270
stop_servers, 270
subIterator, 270
subordinate_model, 269
total_eval_counter, 270
NestedModel, 269
derived_asynch_compute_response, 272
derived_compute_response, 272
derived_init_communicators, 272
derived_synchronize, 272
derived_synchronize_nowait, 272
response_mapping, 273
subModel, 273
synchronize_nowait_completions, 272
new_eval_counter
DakotaInterface, 108
DakotaModel, 129
HierLayeredModel, 239
NestedModel, 270
SingleModel, 334
SurrLayeredModel, 356
newCenterFlag
SurrBasedOptStrategy, 352
newFnEvalId
DakotaInterface, 109
newSolsGenerated
DataMethod, 192
next_eval
COLINApplication, 62
SGOPTApplication, 326
NFDG
CONMINOptimizer, 73
KrigApprox, 242
nlf0
SNLLOptimizer, 342
nlf0_evaluator
SNLLOptimizer, 344
nlf1
SNLLOptimizer, 342
nlf1_evaluator
SNLLOptimizer, 344
nlf1Con
SNLLOptimizer, 342
nlf2
SNLLLeastSq, 339
SNLLOptimizer, 342
nlf2_evaluator
SNLLOptimizer, 344
nlf2_evaluator_gn
SNLLLeastSq, 340
nlf2Con
SNLLLeastSq, 339
SNLLOptimizer, 342
nlfConstraint
SNLLLeastSq, 339
SNLLOptimizer, 342
nlfObjective
SNLLLeastSq, 338
SNLLOptimizer, 342
nlfNConstraint
NLSSOLeastSq, 339
SNLLOptimizer, 342
NLSSOLeastSq
~NLSSOLeastSq, 275
least_sq_eval, 275
minimize_residuals, 275
NLSSOLeastSq, 275
NLSSOLeastSq, 275
NoDBBaseConstructor
NoDBBaseConstructor, 277
nonAdaptiveFlag
DataMethod, 193
NonDAdvMeanValue
~NonDAdvMeanValue, 278
amvFlag, 281
cholCorrMatrix, 280
erfInverse, 279
impFactor, 280
nonlinearIneqUpperBnds
   DACEIterator, 83
   DakotaOptLeastSq, 147
   DataResponses, 197
   ParamStudy, 314
nonlinEqTargets
   SurrBasedOptStrategy, 353
nonlinIneqLowerBnds
   SurrBasedOptStrategy, 352
nonlinIneqUpperBnds
   SurrBasedOptStrategy, 353
normalDistLowerBnds
   DakotaNonD, 140
normalDistUpperBnds
   DakotaNonD, 140
normalMeans
   DakotaNonD, 139
normalStdDevs
   DakotaNonD, 139
normalUncDistLowerBnds
   DataVariables, 205
normalUncDistUpperBnds
   DataVariables, 205
normalUncMeans
   DataVariables, 205
normalUncStdDevs
   DataVariables, 205
NPSOLOptimizer, 294
   ~NPSOLOptimizer, 294
   find optimum, 294
   find optimum on model, 294
   find optimum on user functions, 294
   initialPoint, 295
   lowerBounds, 295
   NPSOLOptimizer, 294
   objective eval, 295
   setUpType, 295
   upperBounds, 295
   userConstraintEval, 295
   userObjectiveEval, 295
NSCAL
   KrigApprox, 244
NSIDE
   KrigApprox, 244
   num_active_variables
   DakotaVarConstraints, 170
   num_columns
   DakotaMatrix, 126
   num_continuous_variables
   DataVariables, 204
   num_discrete_variables
   DataVariables, 204
   num_functions
   DakotaModel, 130
   DakotaResponse, 150
   num_integer_params
   COLINAApplication, 62
   num_linear_eq_constraints
   DakotaModel, 131
   DakotaVarConstraints, 170
   num_linear_ineq_constraints
   DakotaModel, 131
   DakotaVarConstraints, 169
   num_objectives
   rSQPOptimizer, 323
   num_real_params
   COLINAApplication, 62
   num_rows
   DakotaMatrix, 126
   num_variables
   DakotaApproximation, 86
   DataVariables, 203
   numActiveVars
   NonDSampling, 292
   numAnalysisDrivers
   ApplicationInterface, 47
   numAnalysisServers
   ApplicationInterface, 47
   ParallelLibrary, 305
   numberIterations
   SOLBase, 347
   numberRetained
   DataMethod, 192
   numCDV
   AllVarConstraints, 34
   AllVariables, 37
   numChaos
   HermiteSurf, 237
   NonDPCESampling, 289
   numCoeffs
   RespSurf, 321
   numcon
   KrigApprox, 242
   numConstraints
   DakotaOptLeastSq, 147
   numContinuousDesVars
   DataVariables, 204
   numContinuousStateVars
   DataVariables, 204
   numContinuousVars
   DakotaIterator, 115
   numCSV
   AllVarConstraints, 34
   AllVariables, 37
   numCurrentPoints
   DakotaApproximation, 86
   numDACERuns
   DACEIterator, 82
numDDV
  AllVarConstraints, 34
  AllVariables, 37
numDesignVars
  NonDSampling, 292
numDiscreteDesVars
  DataVariables, 204
numDiscreteVars
  DakotaIterator, 115
numDSV
  AllVarConstraints, 34
  AllVariables, 37
numEvalServers
  ApplicationInterface, 48
  ParallelLibrary, 304
numFns
  DakotaModel, 133
  DirectFnApplicInterface, 209
  SurrBasedOptStrategy, 352
numFunctions
  DakotaIterator, 115
numGradVars
  DakotaModel, 133
  DirectFnApplicInterface, 209
numHistogramUncVars
  DataVariables, 204
numHistogramVars
  DakotaNonD, 141
numInterfEqConstr
  NestedModel, 271
numInterfIneqConstr
  NestedModel, 271
numInterfObjFns
  NestedModel, 271
numIteratorJobs
  ConcurrentStrategy, 70
numIterators
  MultilevelOptStrategy, 267
numIteratorServers
  BranchBndStrategy, 59
  ConcurrentStrategy, 70
  ParallelLibrary, 303
numLeastSqTerms
  DakotaLeastSq, 120
  DataResponses, 196
numLevels
  NonDAvMeanValue, 280
numLSRuns
  NonDSampling, 292
numLinearConstraints
  DakotaOptLeastSq, 148
numLinearEqConstraints
INDEX

DataMethod, 193
numPrograms
  AnalysisCode, 40
numRelEqConstr
  NonDAvMeanValue, 280
numResponseFunctions
  DakotaNonD, 141
  DataResponses, 197
numRootSamples
  BranchBndStratagy, 59
numSamples
  DACEIterator, 82
  DakotaApproximation, 86
  DataMethod, 194
numSampQuad
  KrigApprox, 245
numStateVars
  NonDSampling, 292
numSteps
  DataMethod, 195
  ParamStudy, 313
numSubIteratorEqConstr
  NestedModel, 270
numSubIteratorIneqConstr
  NestedModel, 270
numSymbols
  DACEIterator, 82
  DataMethod, 194
numUncertainVars
  DakotaNonD, 141
numUniformUncVars
  DataVariables, 204
numUniformVars
  DakotaNonD, 141
numUV
  AllVarConstraints, 34
  AllVariables, 37
numVars
  DakotaApproximation, 86
  DakotaIterator, 115
  DirectFnApplicInterface, 209
  SurrBasedOptStrategy, 352
numWeibullUncVars
  DataVariables, 204
numWeibullVars
  DakotaNonD, 141
objCoeffs
  NestedModel, 271
objective_eval
  NPSOLOptimizor, 295
occurrencesOf
  DakotaList, 125
offsetValues

LayeredModel, 254
oFilterName
  AnalysisCode, 40
  DirectFnApplicInterface, 209
operator const char *
  DakotaString, 166
operator T *
  DakotaArray, 92
  DakotaVector, 180
operator!=
  DakotaResponse, 152
  DakotaVariables, 177
  ParamResponsePair, 310
operator()
  DakotaArray, 92
  DakotaBaseVector, 96
  FunctionCompare, 222
  SortCompare, 349
operator<<
  CommandShell, 68
  DakotaBosStream, 101–103
operator=
  CtelRegexp, 80
  DakotaApproximation, 88
  DakotaArray, 90, 92
  DakotaBaseVector, 94
  DakotaInterface, 111
  DakotaIterator, 118
  DakotaList, 122
  DakotaMatrix, 127
  DakotaModel, 137
  DakotaResponse, 150
  DakotaStrategy, 162
  DakotaString, 165
  DakotaVarConstraints, 172
  DakotaVariables, 179
  DakotaVector, 180, 182
  DataInterface, 184
  DataMethod, 189
  DataResponses, 196
  DataStrategy, 199
  DataVariables, 203
  ParamResponsePair, 309
  SurrogateDataPoint, 360
operator==
  AllMergedVariables, 30
  AllVariables, 38
  DakotaResponse, 152
  DakotaResponseRep, 156
  DakotaVariables, 177
  DataInterface, 184
  DataMethod, 189
  DataResponses, 196
  DataVariables, 203
ParallelLibrary, 306
resolve_inputs, 308
parameterSets
ConcurrentStrategy, 70
parametersFile
DataInterface, 185
parametersFileName
AnalysisCode, 40
parametersFNameList
AnalysisCode, 41
ParamResponsePair
~ParamResponsePair, 309
active_set, vector, 310
eval_id, 310
interface_id, 310
operator =, 310
operator=, 309
operator==, 310
ParamResponsePair, 309, 311
prp_parameters, 310
prp_response, 310
prPairParameters, 310
prPairResponse, 310
read, 309
read_annotated, 309
write, 309, 310
write_annotated, 309
write_tabular, 309
ParamResponsePair, 309
evalId, 311
ParamResponsePair, 311
ParamStudy
~ParamStudy, 312
bestObjectiveFn, 314
bestResponses, 314
bestVariables, 314
bestViolations, 314
centered_loop, 313
compute_vector_steps, 312
deltasPerVariable, 313
finalPoint, 313
initialPoint, 313
iterator_response_results, 312
iterator_variable_results, 312
listOfPoints, 313
multidim_loop, 313
multiObjWeights, 314
nestedFlag, 313
nonlinearEqTargets, 314
nonlinearIneqLowerBnds, 314
nonlinearIneqUpperBnds, 314
numNonlinearEqConstraints, 314
numNonlinearIneqConstraints, 314
numObjectiveFunctions, 314
numSteps, 313
ParamStudy, 312
percentDelta, 313
print_iterator_results, 312
pCounter, 314
pStudyType, 313
recurse, 313
sample, 312
stepLength, 313
stepVector, 313
update_best, 313
variablePartitions, 314
vector_loop, 312
ParamStudy, 312
run_iterator, 315
paramStudyType
DataMethod, 194
parse
GetLongOpt, 234
patternBasis
DataMethod, 193
patternSearchOptimizer
SGOPTOptimizer, 329
penaltyParameter
surrogateOptStrategy, 351
percentDelta
DataMethod, 195
ParamStudy, 313
petraCorrMatrix
NonDAdvMeanValue, 280
petraProbLevels
NonDAdvMeanValue, 281
petraRespLevels
NonDAdvMeanValue, 281
pGAIntOptimizer
SGOPTOptimizer, 329
pGAREalOptimizer
SGOPTOptimizer, 329
PHI
KrigApprox, 244
picoComm
BranchBndStrategy, 59
picoCommRank
BranchBndStrategy, 59
picoCommSize
BranchBndStrategy, 60
picoListOfIntegers
BranchBndStrategy, 60
picoLowerBnds
BranchBndStrategy, 60
picoUpperBnds
BranchBndStrategy, 60
pname
GetLongOpt, 233
polyCoeffs
RespSurf, 321
polynomialOrder
DataInterface, 187
polyOrder
RespSurf, 322
populate_gradient_vars
DakotaIterator, 119
populationSize
DataMethod, 192
post_instantiate
SNLLBase, 335
post_run
SNLLBase, 335
pre_instantiate
SNLLBase, 335
pre_run
SNLLBase, 335
primaryCoeffs
DataMethod, 190
print
DakotaArray, 91
DakotaList, 122
DakotaMatrix, 126
DakotaVector, 181, 182
print_annotated
DakotaVector, 181
print_aprepro
DakotaVector, 181
print_configuration
ParallelLibrary, 297
print_iterator_results
DACEIterator, 81
DakotaIterator, 113
DakotaLeastSq, 121
DakotaOptimizer, 144
NonDAAdvMeanValue, 278
NonDLSSampling, 285
NonDPCESampling, 289
ParamStudy, 312
print_partial
DakotaVector, 181
print_partial_aprepro
DakotaVector, 181
print_restart
restart_util.C, 371
print_restart_tabular
restart_util.C, 372
print_statistics
NonDSampling, 291
printControl
CONMINOptimizer, 75
DOTOptimizer, 214
prob_desc_db
DakotaModel, 132
DakotaStrategy, 164
probabilityLevels
DataMethod, 194
probDescDB
DakotaIterator, 115
DakotaModel, 134
DakotaStrategy, 160
problem
COLINOptimizer, 64
ProblemDescDB
~ProblemDescDB, 316
build_label, 318
build_labels, 318
check_input, 316
dbLocked, 319
get_bool, 317
get_db_list_nodes, 316
get_dia, 317
get_dil, 317
get_div, 317
get_dtra, 317
get_drm, 317
get_drv, 317
get_drva, 317
get_dvs, 317
get_dsl, 317
get_int, 317
get_real, 317
get_sizet, 317
get_string, 317
insert_node, 317, 318
interface_kwhandler, 318
interfaceIter, 319
interfaceList, 319
method_kwhandler, 318
methodIter, 319
methodList, 319
parallel_library, 316
parallelLib, 319
ProblemDescDB, 316
receive_db_buffer, 318
responses_kwhandler, 318
responsesIter, 319
responsesList, 319
send_db_buffer, 318
set_db_interface_node, 316
set_db_list_nodes, 316
set_db_responses_node, 316
set_other_list_nodes, 318
strategy_kwhandler, 318
strategyCntr, 319
strategySpec, 319
variables_khandler, 318
variablesIter, 319
variablesList, 319
ProblemDescDB, 316
   manageInputs, 320
   setDbModelType, 320
probLevels
   NonDAdvMeanValue, 280
probMoreThanThresh
   DakotaNonD, 141
processIdList
   ForkApplicInterface, 220
procsPerAnalysis
   ApplicationInterface, 48
   DataInterface, 185
   ParallelLibrary, 305
procsPerEval
   ParallelLibrary, 304
procsPerIterator
   ParallelLibrary, 303
programNames
   AnalysisCode, 39
   programNames
   AnalysisCode, 40
progressMetric
   MultilevelOptStrategy, 267
progressThreshold
   MultilevelOptStrategy, 267
prp prp
   ParamResponsePair, 310
prp response
   ParamResponsePair, 310
prPairParameters
   ParamResponsePair, 310
prPairResponse
   ParamResponsePair, 310
psCounter
   ParamStudy, 314
pStudyType
   ParamStudy, 313
purge inactive
   DakotaResponse, 152
   DakotaResponseRep, 155
pxcFile
   DirectFnApplicInterface, 209
quantify uncertainty
   DakotaNonD, 139
   NonDAdvMeanValue, 278
   NonDLHSSampling, 286
   NonDPCESampling, 289
quietFlag
   DakotaInterface, 110
DakotaModel, 135
quietOutput
   DakotaIterator, 116
r
   CтелRegexp, 80
randomizeOrderFlag
   DataMethod, 193
randomSeed
   DACEIterator, 82
   DataMethod, 194
   NonDSampling, 292
ranVarMeans
   NonDAdvMeanValue, 281
ranVarSigmas
   NonDAdvMeanValue, 281
ranVarType
   NonDAdvMeanValue, 281
rawCVarsList
   LayeredModel, 253
rawResponseArray
   DakotaInterface, 112
rawResponseList
   DakotaInterface, 112
rc
   ErrorTable, 216
read
   AllMergedVarConstraints, 27
   AllMergedVariables, 29, 30
   AllVarConstraints, 33
   AllVariables, 36, 37
   DakotaArray, 91
   DakotaList, 122
   DakotaMatrix, 126
   DakotaResponse, 151
   DakotaResponseRep, 156–158
   DakotaVarConstraints, 169
   DakotaVariables, 175, 176
   DakotaVector, 180, 181
   DataInterface, 184
   DataMethod, 189
   DataResponses, 196
   DataStrategy, 199
   DataVariables, 203
   FundamentalVarConstraints, 224
   FundamentalVariables, 228, 229
   MergedVarConstraints, 260
   MergedVariables, 263, 264
   ParamResponsePair, 309
read_annotated
   AllMergedVariables, 29
   AllVariables, 36
   DakotaResponse, 151
   DakotaResponseRep, 157
responsesIter
  ProblemDescDB, 319
responsesList
  ProblemDescDB, 319
responsesPointer
  DataMethod, 190
responseThresholds
  DataMethod, 194
RespSurf
 ~RespSurf, 321
  approximation coefficients, 321
  find coefficients, 321
  get gradient, 321
  get value, 321
  numCoeffs, 321
  polyCoeffs, 321
  polyOrder, 322
  required samples, 321
RespSurf, 321
RespSurf, 321
respThresh
  DakotaNonD, 141
restart_util.C, 371
  concatenate restart, 372
  main, 372
  print restart, 371
  print restart tabular, 372
  read neutral, 372
  repair restart, 372
  write precision, 371
restartFileFlag
  ApplicationInterface, 49
  DataInterface, 186
results_fName
  AnalysisCode, 39
resultsFile
  DataInterface, 185
resultsFileName
  AnalysisCode, 40
resultsFNameList
  AnalysisCode, 41
retrieve
  GetLongOpt, 234
retryLimit
  DataInterface, 186
rhsTermsVector
  KrigApprox, 246
rng
  COLINOptimizer, 65
rosenbrock
  DirectFnApplicInterface, 210
rSQPOptimizer
  ~rSQPOptimizer, 323
  find optimum, 323
lin_eq_jac, 323
lin_eq_jarg, 323
lin_ineq_jac, 323
lin_ineq_lb, 323
lin_ineq_ub, 323
model, 323
nlp, 323
nonlin_eq_jarg, 323
nonlin_ineq_lb, 323
nonlin_ineq_ub, 323
num_objectives, 323
rSQPOptimizer, 323
run_coupled
  MultilevelOptStrategy, 267
run_iterator
  DACEIterator, 83
  DakotaIterator, 118
  DakotaLeastSq, 121
  DakotaNonD, 139
  DakotaOptimizer, 144
  DakotaStrategy, 162
  ParamStudy, 315
run_iterator_repartition
  DakotaStrategy, 163
run_lhs
  NonDSampling, 291
run_strategy
  BranchBndStrategy, 59
  ConcurrentStrategy, 70
  DakotaStrategy, 159
  MultilevelOptStrategy, 266
  NonDOptStrategy, 287
  SingleMethodStrategy, 331
  SurrBasedOptStrategy, 353
run_uncoupled
  MultilevelOptStrategy, 268
runUncoupledAdaptive
  MultilevelOptStrategy, 268
runConminFlag
  KrigingSurf, 251
runningList
  ApplicationInterface, 50
rvec
  ColinPoint, 66
rXhatVector
  KrigApprox, 246
S
  CONMINOptimizer, 77
  KrigApprox, 247
s
  SNLLBase, 336
salinas
DirectFnApplicInterface, 211
sample
ParamStudy, 312
sampleReuse
ApproximationInterface, 56
sampleReuseFile
ApproximationInterface, 56
sampleType
DataMethod, 194
NonDSampling, 292
sampling_reset
DACEIterator, 81
DakotaIterator, 114
NonDSampling, 293
sampling_scheme
DACEIterator, 81
DakotaIterator, 114
NonDSampling, 291
SCAL
CONMINOptimizer, 77
KrigApprox, 248
scaleFactors
LayeredModel, 254
search_val
FunctionCompare, 222
searchMethod
DataMethod, 191
SNLLBase, 336
searchSchemeSize
DataMethod, 192
secondaryCoeffs
DataMethod, 190
selectedIterator
BranchBndStrategy, 59
ConcurrentStrategy, 70
SingleMethodStrategy, 331
SurrBasedOptStrategy, 351
selectedIterators
MultilevelOptStrategy, 267
selectionPressure
DataMethod, 193
self_schedule_analyses
ApplicationInterface, 51
self_schedule_evaluations
ApplicationInterface, 52
send_db_buffer
ProblemDescDB, 318
send_ea
ParallelLibrary, 298
send_ie
ParallelLibrary, 298
send_si
ParallelLibrary, 298
serve
DakotaModel, 129
HierLayeredModel, 239
NestedModel, 270
SingleModel, 334
SurrLayeredModel, 356
serve_analyses_async
ForkApplicInterface, 221
serve_analyses_sync
ApplicationInterface, 51
serve_evaluations
ApplicationInterface, 51
DakotaInterface, 107
serve_evaluations_async
ApplicationInterface, 53
serve_evaluations_peer
ApplicationInterface, 54
serve_evaluations_sync
ApplicationInterface, 53
set_bounds
DakotaApproximation, 86
set_db_interface_node
ProblemDescDB, 316
set_db_list_nodes
ProblemDescDB, 316
set_db_model_type
ProblemDescDB, 320
set_db_responses_node
ProblemDescDB, 316
set_initial_point
COLINOptimizer, 64
set_local_data
DirectFnApplicInterface, 209
set_method_options
COLINOptimizer, 64
SGOPTOptimizer, 330
set_options
SOLBase, 346
set_other_list_nodes
ProblemDescDB, 318
set_rng
COLINOptimizer, 64
set_standard_method_options
COLINOptimizer, 65
setCell
GetLongOpt, 233
setUpType
NPSOLOptimizer, 295
SGOPTApplication, 325
~SGOPTApplication, 325
activeSetVector, 325
copy, 325
dakotaAsyncFlag, 326
dakotaCompletionList, 326
dakotaModelAsyncFlag, 325
dakotaResponseList, 325
DoEval, 326
multiobjModifyPtr, 326
nextEval, 326
numNonlinCons, 326
numObjFns, 326
SGOPTApplication, 325
synchronize, 326
userDefinedModel, 325
sgoptApplication
SGOPTOptimizer, 330
SGOPTOptimizer, 328
~SGOPTOptimizer, 330
aPPS Optimizer, 329
baseOptimizer, 329
discreteAppFlag, 328
ePSAOptimizer, 329
exploratoryMoves, 328
find_optimum, 330
intProblem, 329
linConGenerator, 328
patternSearchOptimizer, 329
pGAIntOptimizer, 329
pGAREalOptimizer, 329
realProblem, 329
set_method_options, 330
sgoptApplication, 330
SGOPTOptimizer, 330
sMCrealOptimizer, 329
sWOptimizer, 329
show_data_3d
DakotaGraphics, 105
silentFlag
DakotaInterface, 109
DakotaModel, 135
silentOutput
DakotaIterator, 116
SingleMethodStrategy
~SingleMethodStrategy, 331
run_strategy, 331
selectedIterator, 331
SingleMethodStrategy, 331
strategy_response_results, 331
strategy_variable_results, 331
userDefinedModel, 331
SingleMethodStrategy, 331
SingleModel
~SingleModel, 333
derived_asynch_compute_response, 333
derived_computed_response, 333
derived_init_communicators, 333
derived_master_overload, 333
derived_synchronize, 333
derived_synchronize_nowait, 333
free_communicators, 334
local_eval_synchronization, 333
new_eval_counter, 334
serve, 334
SingleModel, 333
stop_servers, 334
synchronize_nowait_completions, 333
total_eval_counter, 334
userDefinedInterface, 334
SingleModel, 333
sMCrealOptimizer
SGOPTOptimizer, 329
SNLLBase, 335
~SNLLBase, 335
constantASVFlag, 336
copy_con_grad, 336
copy_con_hess, 336
copy_con_vals, 336
init_fn, 336
mfcn, 336
postInstantiate, 335
post_run, 335
preInstantiate, 335
pre_run, 335
s, 336
searchMethod, 336
SNLLBase, 335
staticConEvalMode, 337
staticConEvalVars, 337
staticDebugOutput, 336
staticLastFnEvalLocn, 336
staticModeOverrideFlag, 336
staticNumNonlinearEqConstraints, 337
staticNumNonlinearIneqConstraints, 337
SNLLLeastSq
~SNLLLeastSq, 338
constraint2_evaluator_gn, 338
minimize_residuals, 338
nlf2, 339
nlf2Con, 339
nlfConstraint, 339
nlfObjective, 338
nlpConstraint, 339
opbcnewton, 339
optnewton, 339
optnips, 339
SNLLLeastSq, 338
theOptimizer, 339
SNLLLeastSq, 338
nlf2_evaluator_gn, 340
SNLLOptimizer, 341
~SNLLOptimizer, 341
constraint0_evaluator, 344
constraint1_evaluator, 344
constraint2_evaluator, 344
fdnlf1, 342
fdnlf1Con, 342
findoptimum, 341
nlf0, 342
nlf0_evaluator, 344
nlf1, 342
nlf1_evaluator, 344
nlf1Con, 342
nlf2, 342
nlf2_evaluator, 344
nlf2Con, 342
nlfConstraint, 342
nlfObjective, 342
nlpConstraint, 342
optbcnfdnewton, 343
optbcnewton, 343
optbcqnewton, 343
optcg, 342
optdfnnewton, 343
optdfdnips, 343
optnewton, 343
optnips, 343
optpds, 342
optqnips, 343
SNLLOptimizer, 341
setoptions, 346
SNLLBase, 346
staticConstrOffset, 348
staticMaxFnEvals, 348
staticSOLModel, 348
staticVendorNumericalGradFlag, 348
upperFactorHessianF77, 348
soInAccuracy
DataMethod, 192
sort
DakotaList, 124
SortCompare
operator(), 349
SortCompare, 349
sortFunction, 349
SortCompare, 349
sortFunction
SortCompare, 349
spawn_analysis
SysCallAnalysisCode, 363
spawn_application
SysCallApplicInterface, 365
spawn_evaluation
SysCallAnalysisCode, 363
spawn_input_filter
SysCallAnalysisCode, 363
spawn_output_filter
SysCallAnalysisCode, 363
speculativeFlag
DakotaOptLeastSq, 148
DataMethod, 190
split
CtelRegexp, 79
split_communicator, dedicated_master
ParallelLibrary, 302
split_communicator, peer_partition
ParallelLibrary, 302
startClock
ParallelLibrary, 303
startCPUTime
ParallelLibrary, 303
startMPTime
ParallelLibrary, 303
startWCTime
ParallelLibrary, 303
state
DataVariables, 203
static_schedule_evaluations
ApplicationInterface, 52
staticConEvalMode
SNLLBase, 337
staticConEvalVars
SNLLBase, 337
DataStrategy, 201
SurfLayeredModel
~SurfLayeredModel, 355
approximation of coefficients, 356
approxInterface, 356
daceIterator, 357
daceMethodPointer, 357
free communicators, 356
new eval counter, 356
serve, 356
stop servers, 356
subordinate iterator, 355
subordinate model, 355
SurfLayeredModel, 355
synchronize nowait completions, 356
total eval counter, 356
update approximation, 356
SurfLayeredModel, 355
actualInterfacePointer, 359
actualModel, 359
build approximation, 358
derived asynch compute response, 357
derived compute response, 357
derived init communicators, 358
derived master overload, 358
derived synchronize, 357
derived synchronize nowait, 358
maximum concurrency, 358
SurrogateDataPoint
~SurrogateDataPoint, 360
continuousVars, 360
operator=, 360
operator==, 360
responseFn, 360
responseGrad, 360
SurrogateDataPoint, 360
SurrogateDataPoint, 360
sWOptimizer
SGOPTOptimizer, 329
synch
ApplicationInterface, 50
ApproximationInterface, 56
DakotaInterface, 107
synch nowait
ApplicationInterface, 51
ApproximationInterface, 56
DakotaInterface, 107
synch nowait completions
DakotaInterface, 108
synchronize
COLINApplication, 62
DakotaModel, 130
SGOPTApplication, 326
synchronize_fd_gradients
DakotaModel, 138
synchronize nowait
DakotaModel, 130
synchronize nowait completions
DakotaModel, 129
HierLayeredModel, 239
NestedModel, 272
SingleModel, 333
SurfLayeredModel, 356
synchronous local analyses
ForkApplicInterface, 221
synchronous local evaluations
ApplicationInterface, 53
SysCallAnalysisCode
~SysCallAnalysisCode, 362
command usage, 362
commandUsage, 362
SysCallAnalysisCode, 362
SysCallAnalysisCode, 362
spawn analysis, 363
spawn evaluation, 363
spawn input filter, 363
spawn output filter, 363
SysCallApplicInterface
~SysCallApplicInterface, 364
derived map, 364
derived_map async, 364
derived sync, 364
derived sync kernel, 365
derived sync nowait, 364
derived synchronous local analysis, 364
failCountList, 365
failIdList, 365
spawn application, 365
SysCallApplicInterface, 364
sysCallList, 365
dysCallSimulator, 365
sysCallSimulator, 365
system call file test, 365
SysCallApplicInterface, 364
sysCallList
SysCallApplicInterface, 365
dysCallSimulator
SysCallApplicInterface, 365
system call file test
SysCallApplicInterface, 365
table
GetLongOpt, 233
tabularDataFile
DakotaStrategy, 161
DataStrategy, 199
tabularDataFlag
DakotaGraphics, 104
DakotaStrategy, 161
DataStrategy, 199
tabularDataStream
   DakotaGraphics, 105
tag_argument_list
   ForkAnalysisCode, 217
TaylorSurf
   ~TaylorSurf, 366
   find_coefficients, 366
   get_gradient, 366
   get_value, 366
   required_samples, 366
   TaylorSurf, 366
TaylorSurf, 366
testClass
   DakotaArray, 93
   DakotaBoStream, 103
   DakotaList, 123
   DakotaMatrix, 127
   DakotaString, 166
   DakotaVector, 182
testFunction
   FunctionCompare, 222
text_book
   DirectFnApplicInterface, 210
text_book1
   DirectFnApplicInterface, 210
text_book2
   DirectFnApplicInterface, 210
text_book3
   DirectFnApplicInterface, 210
text_book_uuu
   DirectFnApplicInterface, 210
theOptimizer
   SNLLLeastSq, 339
   SNLLOptimizer, 342
THETA
   KrigApprox, 244
   thetaLoBndVector
      KrigApprox, 245
   thetaUpBndVector
      KrigApprox, 246
   thetaVector
      KrigApprox, 245
   threshDelta
      DataMethod, 192
toLower
   DakotaString, 165
total_eval_counter
   DakotaInterface, 108
   DakotaModel, 129
   HierLayeredModel, 239
   NestedModel, 270
   SingleModel, 334
   SurrLayeredModel, 356
   totalPatternSize
      DataMethod, 193
toUpper
   DakotaString, 165
   transNataf
      NonDAvMeanValue, 284
   transUToX
      NonDAvMeanValue, 278, 282
   transUToZ
      NonDAvMeanValue, 283
   transXToU
      NonDAvMeanValue, 282
   transXToZ
      NonDAvMeanValue, 282
   transZToU
      NonDAvMeanValue, 283
   trRatioContractValue
      SurrBasedOptStrategy, 351
   trRatioExpandValue
      SurrBasedOptStrategy, 351
   trustRegionFactor
      SurrBasedOptStrategy, 351
   trustRegionSize
      SurrBasedOptStrategy, 351
tv
   AllMergedVariables, 28
   AllVariables, 35
   DakotaModel, 130
   DakotaVariables, 174
   FundamentalVariables, 227
   MergedVariables, 262
   uncertain
      DataVariables, 203
   uncertainCorrelations
      DakotaNonD, 140
      DataVariables, 206
   uncertainDistLowerBnds
      DataVariables, 206
      FundamentalVarConstraints, 225
      MergedVarConstraints, 261
   uncertainDistUpperBnds
      DataVariables, 206
      FundamentalVarConstraints, 225
      MergedVarConstraints, 261
   uncertainLabels
      DataVariables, 206
      FundamentalVariables, 230
      MergedVariables, 264
   uncertainVars
      DataVariables, 206
      FundamentalVariables, 229
      MergedVariables, 264
   uniformDistLowerBnds