SAND2010-2184
Unlimited Release
December 2009
Updated April 29, 2013

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

Version 5.3.1 Reference Manual

Brian M. Adams, Mohamed S. Ebeida, Michael S. Eldred,
John D. Jakeman, Laura P. Swiler
Optimization and Uncertainty Quantification Department

William J. Bohnhoff
Radiation Transport Department

Keith R. Dalbey
Mission Analysis and Simulation Department

John P. Eddy
System Readiness and Sustainment Technologies Department

Kenneth T. Hu
Validation and Uncertainty Quantification Department

Dena M. Vigil
Multiphysics Simulation Technologies Department

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

Lara E. Bauman, Patricia D. Hough
Quantitative Modeling and Analysis Department

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

Abstract

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

This report serves as a reference manual for the commands specification for the Dakota software, providing input overviews, option descriptions, and example specifications.

Contents

1 Dakota Reference Manual **9**

1.1 Overview . . . 9

1.2 Input Specification Reference . . . 9

1.3 Additional Resources . . . 10

2 Introduction **11**

2.1 Overview . . . 11

2.2 NIDR Input Specification File . . . 11

2.3 Common Specification Mistakes . . . 13

2.4 Sample dakota.in Files . . . 13

2.5 Tabular descriptions . . . 18

3 Strategy Commands **19**

3.1 Strategy Description . . . 19

3.2 Strategy Specification . . . 20

3.3 Strategy Independent Controls . . . 21

3.4 Hybrid Minimization Commands . . . 22

3.5 Multistart Iteration Commands . . . 24

3.6 Pareto Set Optimization Commands . . . 24

3.7 Single Method Commands . . . 24

4 Method Commands **27**

4.1 Method Description . . . 27

4.2 Method Specification . . . 28

4.3 Method Independent Controls . . . 29

4.4 Optimization Methods . . . 34

4.5 Least Squares Methods . . . 67
4.6 Surrogate-Based Minimization Methods . . . 73
4.7 Uncertainty Quantification Methods . . . 78
4.8 Nondeterministic Calibration Methods . . . 112
4.9 Solution Verification Methods . . . 114
4.10 Design of Computer Experiments Methods . . . 115
4.11 Parameter Study Methods . . . 121

5 Model Commands **125**
5.1 Model Description . . . 125
5.2 Model Specification . . . 126
5.3 Model Independent Controls . . . 126
5.4 Single Model Controls . . . 127
5.5 Surrogate Model Controls . . . 127
5.6 Nested Model Controls . . . 136

6 Variables Commands **141**
6.1 Variables Description . . . 141
6.2 Variables Specification . . . 144
6.3 Variables Set Identifier . . . 144
6.4 Active Variables . . . 145
6.5 Variables Domain . . . 146
6.6 Design Variables . . . 146
6.7 Aleatory Uncertain Variables . . . 147
6.8 Epistemic Uncertain Variables . . . 163
6.9 State Variables . . . 165

7 Interface Commands **169**
7.1 Interface Description . . . 169
7.2 Interface Specification . . . 170
7.3 Interface Independent Controls . . . 170
7.4 Algebraic mappings . . . 172
7.5 Simulation interfaces . . . 173

8 Responses Commands **179**

8.1 Responses Description 179
8.2 Responses Specification 180
8.3 Responses Set Identifier 181
8.4 Response Labels 181
8.5 Function Specification 182
8.6 Gradient Specification 188
8.7 Hessian Specification 190

9 Bibliography **195**

10 File Documentation **201**

10.1 dakota.input.summary File Reference 201

Chapter 1

Dakota Reference Manual

Author:

Brian M. Adams, Lara E. Bauman, William J. Bohnhoff, Keith R. Dalbey, John P. Eddy, Mohamed S. Ebeida, Michael S. Eldred, Patricia D. Hough, Kenneth T. Hu, John D. Jakeman, Laura P. Swiler, Dena M. Vigil

1.1 Overview

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

The Reference Manual focuses on documentation of the various input commands for Dakota. It follows closely the structure of dakota.input.summary, the master input specification summary. For information on software structure, refer to the `Developers Manual`, and for a tour of Dakota features and capabilities, refer to the Users Manual [Adams et al., 2010].

1.2 Input Specification Reference

In Dakota, the *strategy* creates and manages *iterators* and *models*. A model contains a set of *variables*, an *interface*, and a set of *responses*, and the iterator operates on the model to map the variables into responses using the interface. In a Dakota input file, the user specifies these components through strategy, method, model, variables, interface, and responses keyword specifications. The Reference Manual closely follows this structure, with introductory material followed by detailed documentation of the strategy, method, model, variables, interface, and

responses keyword specifications:

Introduction

Strategy Commands

Method Commands

Model Commands

Variables Commands

Interface Commands

Responses Commands

1.3 Additional Resources

A bibliography for the Reference Manual is provided in:

Bibliography

Project web pages are maintained at http://dakota.sandia.gov/ including documentation at http://dakota.sandia.gov/documentation.html and a list of publications at http://dakota.sandia.gov/publications.html

Chapter 2

Introduction

2.1 Overview

In Dakota, a *strategy* creates and manages *iterators* and *models*. A model, generally speaking, contains a set of *variables*, an *interface*, and a set of *responses*. An iterator repeatedly operates on the model to map the variables into responses using the interface. Each of these six pieces (strategy, method, model, variables, interface, and responses) are separate specifications in the user's input file, and as a whole, determine the study to be performed during an execution of the Dakota software. A Dakota execution is limited to a single strategy, which may invoke multiple methods. Furthermore, each method may have its own model, consisting of (generally speaking) its own variables, interface, and set of responses. Thus, there may be multiple specifications of the method, model, variables, interface, and responses sections.

Dakota input is most commonly specified through a text file, whose syntax is governed by the New Input Deck Reader (NIDR) parsing system [Gay, 2008], in conjunction with the dakota.input.nspec file which describes allowable inputs. A more concise version of the input specification file, dakota.input.summary, offers a quick reference to keywords and data values from which a particular input file (e.g., `dakota.in`) may be derived. This automatically derived shortened form omits implementation details not needed in a quick reference.

This Reference Manual focuses on the details of allowable specifications in a file input to the Dakota program. Related details on the name and location of the `dakota` program, command line inputs, and execution syntax are provided in the Users Manual [Adams et al., 2010].

2.2 NIDR Input Specification File

Dakota input is governed by the NIDR input specification file. This file (dakota.input.nspec) is used by a code generator to create parsing system components that are compiled into the Dakota executable (refer to **Instructions for Modifying Dakota's Input Specification** for additional information). Therefore, dakota.input.nspec and its derived summary, dakota.input.summary, are the definitive source for input syntax, capability options, and optional and required capability sub-parameters for any given Dakota version. Beginning users may find dakota.input.summary overwhelming or confusing and will likely derive more benefit from adapting example input files to a particular problem. However, advanced users can master the many input specification possibilities by understanding the structure of the input specification file.

Refer to the dakota.input.summary file for current input specifications. From this file listing, it can be seen that the main structure of the strategy specification consists several required group specifications separated by logical OR's (indicated by |): either hybrid OR multi-start OR pareto set OR single method. The method keyword is the most lengthy specification; however, its structure is again relatively simple. The structure is a set of optional method-independent settings followed by a long list of possible methods appearing as required group specifications (containing a variety of method-dependent settings) separated by OR's. The model keyword reflects a structure of three required group specifications separated by OR's. Within the surrogate model type, the type of approximation must be specified with either a global OR multipoint OR local OR hierarchical required group specification. The structure of the variables keyword is that of optional group specifications for continuous and discrete design variables, a number of different uncertain variable distribution types (continuous and discrete, aleatory and epistemic), and continuous and discrete state variables. Each of these specifications can either appear or not appear as a group. Next, the interface keyword allows the specification of either algebraic mappings, simulation-based analysis driver mappings, or both. Within the analysis drivers specification, a system OR fork OR direct OR third-party (Matlab, Python, Scilab, etc.) group specification must be selected. Finally, within the responses keyword, the primary structure is the required specification of the function set (either optimization functions OR calibration functions OR generic response functions), followed by the required specification of the gradients (either none OR numerical OR analytic OR mixed) and the required specification of the Hessians (either none OR numerical OR quasi OR analytic OR mixed). Refer to Strategy Commands, Method Commands, Model Commands, Variables Commands, Interface Commands, and Responses Commands for detailed information on the keywords and their various optional and required specifications. And for additional details on NIDR specification logic and rules, refer to [Gay, 2008].

Some keywords, such as those providing bounds on variables, have an associated list of values. When the same value should be repeated several times in a row, you can use the notation `N*value` instead of repeating the value N times. For example, in Sample 2: Least Squares (Calibration) below,

```
lower_bounds    -2.0   -2.0
upper_bounds     2.0    2.0
```

could also be written

```
lower_bounds    2*-2.0
upper_bounds    2 * 2.0
```

(with optional spaces around the $*$). Another possible abbreviation is for sequences: L:S:U (with optional spaces around the :) is expanded to L L+S L+2$*$S ... U, and L:U (with no second colon) is treated as L:1:U. For example, in one of the test examples distributed with Dakota (test case 2 of `test/dakota_uq_textbook_sop_lhs.in`),

```
histogram_point = 2
  abscissas     = 50. 60. 70. 80. 90.
                  30. 40. 50. 60. 70.
  counts        = 10  20  30  20  10
                  10  20  30  20  10
```

could also be written

```
histogram_point = 2
  abscissas     = 50 : 10 : 90
                  30 : 10 : 70
  counts        = 10:10:30  20  10
                  10:10:30  20  10
```

Count and sequence abbreviations can be used together. For example

```
response_levels =
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

can be abbreviated

```
response_levels =
2*0.0:0.1:1.0
```

2.3 Common Specification Mistakes

Spelling mistakes and omission of required parameters are the most common errors. Some causes of errors are more obscure:

- Documentation of new capability sometimes lags its availability in source and executables, especially nightly releases. When parsing errors occur that the documentation cannot explain, reference to the particular input specification used in building the executable, which is installed alongside the executable, will often resolve the errors.

- If you want to compare results with those obtained using an earlier version of Dakota (prior to 4.1), your input file for the earlier version must use backslashes to indicate continuation lines for Dakota keywords. For example, rather than

  ```
  # Comment about the following "responses" keyword...
  responses,
          objective_functions = 1
          # Comment within keyword "responses"
          analytic_gradients
  # Another comment within keyword "responses"
          no_hessians
  ```

  you would need to write

  ```
  # Comment about the following "responses" keyword...
  responses,                                          \
          objective_functions = 1                     \
          # Comment within keyword "responses"        \
          analytic_gradients                          \
  # Another comment within keyword "responses"        \
          no_hessians
  ```

  with no white space (blanks or tabs) after the \ character.

In most cases, the NIDR system provides error messages that help the user isolate errors in Dakota input files.

2.4 Sample dakota.in Files

A Dakota input file is a collection of fields from the dakota.input.summary file that describe the problem to be solved by Dakota. Several examples follow.

2.4.1 Sample 1: Optimization

The following sample input file shows single-method optimization of the Textbook Example using DOT's modified method of feasible directions. A similar file is available as `Dakota/examples/users/textbook_opt_conmin.in`.

```
# Dakota Input File: textbook_opt_conmin.in
strategy
  single_method
  graphics
  tabular_graphics_data
    tabular_graphics_file = 'textbook_opt_conmin.dat'

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

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

interface
  direct
    analysis_driver =       'text_book'

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

2.4.2 Sample 2: Least Squares (Calibration)

The following sample input file shows a nonlinear least squares (calibration) solution of the Rosenbrock Example using the NL2SOL method. A similar file is available as `Dakota/examples/users/rosen_opt_nls.in`

```
# Dakota Input File: rosen_opt_nls.in
strategy
  graphics
  tabular_graphics_data
    tabular_graphics_file = 'rosen_opt_nls.dat'
  single_method

method
  max_iterations = 100
  convergence_tolerance = 1e-4
  nl2sol

model
  single
```

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

interface
  analysis_driver = 'rosenbrock'
    direct

responses
  calibration_terms = 2
  analytic_gradients
  no_hessians
```

2.4.3 Sample 3: Nondeterministic Analysis

The following sample input file shows Latin Hypercube Monte Carlo sampling using the Textbook Example. A similar file is available as `Dakota/test/dakota_uq_textbook_lhs.in`.

```
strategy,
        single_method

method,
        sampling,
          samples = 100 seed = 1
          complementary distribution
          response_levels = 3.6e+11 4.0e+11 4.4e+11
                            6.0e+04 6.5e+04 7.0e+04
                            3.5e+05 4.0e+05 4.5e+05
          sample_type lhs

variables,
        normal_uncertain = 2
          means             =  248.89, 593.33
          std_deviations    =   12.4,   29.7
          descriptors       =  'TF1n'  'TF2n'
        uniform_uncertain = 2
          lower_bounds      =  199.3,  474.63
          upper_bounds      =  298.5,  712.
          descriptors       =  'TF1u'  'TF2u'
        weibull_uncertain = 2
          alphas            =   12.,    30.
          betas             =  250.,   590.
          descriptors       =  'TF1w'  'TF2w'
        histogram_bin_uncertain = 2
          num_pairs   =  3         4
          abscissas   =  5  8 10  .1  .2  .3  .4
          counts      = 17 21  0  12  24  12   0
          descriptors = 'TF1h'  'TF2h'
        histogram_point_uncertain = 1
          num_pairs   = 2
          abscissas   = 3 4
          counts      = 1 1
          descriptors = 'TF3h'

interface,
        fork asynch evaluation_concurrency = 5
```

```
          analysis_driver = 'text_book'

responses,
        response_functions = 3
        no_gradients
        no_hessians
```

2.4.4 Sample 4: Parameter Study

The following sample input file shows a 1-D vector parameter study using the Textbook Example. It makes use of the default strategy and model specifications (`single_method` and `single`, respectively), so they can be omitted. A similar file is available in the test directory as `Dakota/examples/users/rosen_ps_vector.in`.

```
# Dakota Input File: rosen_ps_vector.in
strategy
  graphics
  tabular_graphics_data
    tabular_graphics_file = 'rosen_ps_vector.dat'
  single_method

method
  vector_parameter_study
    final_point = 1.1  1.3
    num_steps = 10

model
  single

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

interface
  analysis_driver = 'rosenbrock'
    direct

responses
  objective_functions = 1
  no_gradients
  no_hessians
```

2.4.5 Sample 5: Hybrid Strategy

The following sample input file shows a hybrid strategy using three methods. It employs a genetic algorithm, pattern search, and full Newton gradient-based optimization in succession to solve the Textbook Example. A similar file is available as `Dakota/examples/users/textbook_hybrid_strat.in`.

```
strategy
  graphics
  hybrid sequential
    method_list = 'PS' 'PS2' 'NLP'

method
  id_method = 'PS'
```

```
  model_pointer = 'M1'
  coliny_pattern_search stochastic
    seed = 1234
    initial_delta = 0.1
    threshold_delta = 1.e-4
    solution_accuracy = 1.e-10
    exploratory_moves basic_pattern
    #verbose output

method
  id_method = 'PS2'
  model_pointer = 'M1'
  max_function_evaluations = 10
  coliny_pattern_search stochastic
    seed = 1234
    initial_delta = 0.1
    threshold_delta = 1.e-4
    solution_accuracy = 1.e-10
    exploratory_moves basic_pattern
    #verbose output

method
  id_method = 'NLP'
  model_pointer = 'M2'
        optpp_newton
    gradient_tolerance = 1.e-12
    convergence_tolerance = 1.e-15
    #verbose output

model
  id_model = 'M1'
  single
    variables_pointer = 'V1'
    interface_pointer = 'I1'
    responses_pointer = 'R1'

model
  id_model = 'M2'
  single
    variables_pointer = 'V1'
    interface_pointer = 'I1'
    responses_pointer = 'R2'

variables
  id_variables = 'V1'
  continuous_design = 2
    initial_point    0.6    0.7
    upper_bounds     5.8    2.9
    lower_bounds     0.5   -2.9
    descriptors      'x1'   'x2'

interface
  id_interface = 'I1'
  direct
    analysis_driver=  'text_book'

responses
  id_responses = 'R1'
  objective_functions = 1
  no_gradients
  no_hessians
```

```
responses
  id_responses = 'R2'
  objective_functions = 1
  analytic_gradients
  analytic_hessians
```

Additional example input files, as well as the corresponding output and graphics, are provided in the Tutorial chapter of the Users Manual [Adams et al., 2010].

2.5 Tabular descriptions

In the following discussion of keyword specifications, tabular formats (Tables 4.1 through 9.11) are used to present a short description of the specification, the keyword used in the specification, the type of data associated with the keyword, the status of the specification (required, optional, required group, or optional group), and the default for an optional specification.

It can be difficult to capture in a simple tabular format the complex relationships that can occur when specifications are nested within multiple groupings. For example, in the model keyword, the `actual_model_pointer` specification is a required specification within the `multipoint` and `local` required group specifications, which are separated from each other and from other required group specifications (`global` and `hierarchical`) by logical OR's. The selection between the `global`, `multipoint`, `local`, or `hierarchical` required groups is contained within another required group specification (`surrogate`), which is separated from the `single` and `nested` required group specifications by logical OR's. Rather than unnecessarily proliferate the number of tables in attempting to capture all of these inter-relationships, a balance is sought, since some inter-relationships are more easily discussed in the associated text. The general structure of the following sections is to present the outermost specification groups first (e.g., `single`, `surrogate`, or `nested` in Table 6.1), followed by lower levels of group specifications (e.g., `global`, `multipoint`, `local`, or `hierarchical` surrogates in Table 6.3), followed by the components of each group (e.g., Tables 6.4 through 6.9) in succession.

Chapter 3

Strategy Commands

3.1 Strategy Description

The strategy section in a Dakota input file specifies the top level technique which will govern the management of iterators and models in the solution of the problem of interest. Four strategies currently exist: `hybrid`, `multi_start`, `pareto_set`, and `single_method`. These algorithms are implemented within the **Strategy** class hierarchy in the **CollaborativeHybridStrategy**, **EmbeddedHybridStrategy**, **SequentialHybridStrategy**, **ConcurrentStrategy**, and **SingleMethodStrategy** classes. For each of the strategies, a brief algorithm description is given below. Additional information on the algorithm logic is available in the Users Manual [Adams et al., 2010].

In a hybrid minimization strategy (`hybrid`), a set of methods synergistically seek an optimal design. The relationships among the methods are categorized as collaborative, embedded, or sequential. The goal in each case is to exploit the strengths of different optimization and nonlinear least squares algorithms through different stages of the minimization process. Global/local hybrids (e.g., genetic algorithms combined with nonlinear programming) are a common example in which the desire for identification of a global optimum is balanced with the need for efficient navigation to a local optimum.

In the multi-start iteration strategy (`multi_start`), a series of iterator runs are performed for different values of parameters in the model. A common use is for multi-start optimization (i.e., different local optimization runs from different starting points for the design variables), but the concept and the code are more general. An important feature is that these iterator runs may be performed concurrently.

In the pareto set optimization strategy (`pareto_set`), a series of optimization or least squares calibration runs are performed for different weightings applied to multiple objective functions. This set of optimal solutions defines a "Pareto set," which is useful for investigating design trade-offs between competing objectives. Again, these optimizations can be performed concurrently, similar to the multi-start strategy discussed above. The code is similar enough to the `multi_start` technique that both strategies are implemented in the same **ConcurrentStrategy** class.

Lastly, the `single_method` strategy is the default strategy in that it does not provide control over multiple iterators or multiple models. Rather, it provides the means for simple execution of a single iterator on a single model. It is most commonly used when specifying other strategy-independent controls, e.g., tabular output, where the existence of the `strategy` block mandates selection of a strategy.

Each of the strategy specifications identifies one or more method pointers (e.g., `method_list`, `method_pointer`) to identify the iterators that will be used in the strategy. These method pointers are strings that

correspond to the id_method identifier strings from the method specifications (see Method Independent Controls). These string identifiers (e.g., 'NLP1') should *not* be confused with method selections (e.g., dot_mmfd). Each of the method specifications identified in this manner has the responsibility for identifying corresponding model specifications (using model_pointer from Method Independent Controls), which in turn identify the variables, interface, and responses specifications (using variables_pointer, interface_pointer, and responses_pointer from Model Commands) that are used to build the model used by the iterator. If one of these specifications does not provide an optional pointer, then that component will be constructed using the last specification parsed. In addition to method pointers, a variety of graphics options (e.g., tabular_graphics_data), iterator concurrency controls (e.g., iterator_servers), and strategy data (e.g., starting_points) can be specified.

Specification of a strategy block in an input file is optional, with single_method being the default strategy. If no strategy is specified or if single_method is specified without its optional method_pointer specification, then the default behavior is to employ the last method, variables, interface, and responses specifications parsed. This default behavior is most appropriate if only one specification is present for method, variables, interface, and responses, since there is no ambiguity in this case.

Example specifications for each of the strategies follow. A hybrid example is:

```
strategy,
hybrid sequential
  method_list = 'GA', 'PS', 'NLP'
```

A multi_start example specification is:

```
strategy,
multi_start
  method_pointer = 'NLP1'
  random_starts = 10
```

A pareto_set example specification is:

```
strategy,
pareto_set
  method_pointer = 'NLP1'
          random_weight_sets = 10
```

And finally, a single_method example specification is:

```
strategy,
single_method
  method_pointer = 'NLP1'
```

3.2 Strategy Specification

The strategy specification has the following structure:

```
strategy,
<strategy independent controls>
<strategy selection>
  <strategy dependent controls>
```

where <strategy selection> is one of the following: hybrid, multi_start, pareto_set, or single_method.

The <strategy independent controls> are those controls which are valid for a variety of strategies. Unlike the Method Independent Controls, which can be abstractions with slightly different implementations from one method to the next, the implementations of each of the strategy independent controls are consistent for all strategies that use them. The <strategy dependent controls> are those controls which are only meaningful for a specific strategy. Referring to dakota.input.summary, the strategy independent controls are those controls defined externally from and prior to the strategy selection blocks. They are all optional. The strategy selection blocks are all required group specifications separated by logical OR's (hybrid OR multi_start OR pareto_set OR single_method). Thus, one and only one strategy selection must be provided. The strategy dependent controls are those controls defined within the strategy selection blocks. Defaults for strategy independent and strategy dependent controls are defined in **DataStrategy**. The following sections provide additional detail on the strategy independent controls followed by the strategy selections and their corresponding strategy dependent controls.

3.3 Strategy Independent Controls

The strategy independent controls are summarized in Table 4.1, including graphics, output data, and iterator scheduling controls. The graphics flag activates a 2D graphics window containing history plots for the variables and response functions in the study. This window is updated in an event loop with approximately a 2 second cycle time. The tabular_graphics_data flag activates file tabulation of the same variables and response function history data that gets passed to graphics windows with use of the graphics flag. The tabular_graphics_file specification optionally specifies a name to use for this file (dakota_tabular.dat is the default). Within the file, the variables and response functions appear as columns and each function evaluation provides a new table row. This capability is most useful for post-processing of Dakota results with third-party graphics tools such as MATLAB, Tecplot, etc. There is no dependence between the graphics flag and the tabular_graphics_data flag; they may be used independently or concurrently. Numeric output precision (as passed to the C++ control std::setprecision) can be controlled with output_precision with an upper limit of 16 enforced at run-time. When this control is not specified, most Dakota output will default to a precision of 10, though filesystem interfaces and pre-run output use higher precision for accuracy and better results reproducibility.

Experimental capability: Final results from a Dakota study can be output to dakota_results.txt by specifying results_output (optionally specifying an alternate file name with results_output_filename). The current experimental text file format is hierarchical and a precursor to planned output to structured text formats such as XML or YAML and binary formats such as HDF5. The contents, organization, and format of results files are all under active development and are subject to change.

The iterator_servers, iterator_self_scheduling, and iterator_static_scheduling specifications provide manual overrides for the number of concurrent iterator partitions and the scheduling policy for concurrent iterator jobs. These settings are normally determined automatically in the parallel configuration routines (see **ParallelLibrary**) but can be overridden with user inputs if desired. The graphics, tabular_graphics_data, and tabular_graphics_file specifications are valid for all strategies. However, the iterator_servers, iterator_self_scheduling, and iterator_static_scheduling overrides are only useful inputs for those strategies supporting concurrency in iterators, i.e., multi_start and pareto_set.

Description	Keyword	Associated Data	Status	Default
Graphics flag	graphics	none	Optional	no graphics
Tabulation of graphics data	tabular_graphics_data	none	Optional group	no data tabulation
File name for tabular graphics data	tabular_graphics_file	string	Optional	dakota_tabular.dat
Numeric output precision	output_precision	integer	Optional	10 (for most output)
Enable results output	results_output	none	Optional	no output DB
File name for results output	results_output_file	string	Optional	dakota_results.txt
Number of iterator servers	iterator_servers	integer	Optional	no override of auto configure
Self-scheduling of iterator jobs	iterator_self_scheduling	none	Optional	no override of auto configure
Static scheduling of iterator jobs	iterator_static_scheduling	none	Optional	no override of auto configure

Table 3.1: Specification detail for strategy independent controls

3.4 Hybrid Minimization Commands

The hybrid minimization strategy has `sequential`,`embedded`, and `collaborative` approaches (see the Users Manual [Adams et al., 2010] for more information on the algorithms employed). In the sequential approach, the best solutions are transferred from one method to the next through a specified sequence. In the embedded approach, a tightly-coupled hybrid is employed in which a subordinate local method provides periodic refinements to a top-level global method. And in the collaborative approach, multiple methods work together and share solutions while executing concurrently.

In the `sequential` approach, a list of method strings supplied with the `method_list` specification specifies the identity and sequence of iterators to be used. Any number of iterators may be specified. In the sequential approach, method switching is managed through the separate convergence controls of each method. The number of solutions transferred between methods is specified by the particular method through its `final_solutions` method control. For example, if one sets up a two-level strategy with a first method that generates multiple solutions such as a genetic algorithm, followed by a second method that is initialized only at a single point such as a gradient-based algorithm, it is possible to take the multiple solutions generated by the first method and create several instances of the second method, each one with a different initial starting point. The logic governing the transfer of multiple solutions between methods is as follows: if one solution is returned from method A, then one solution is transferred to method B. If multiple solutions are returned from method A, and method B can accept multiple solutions as input (for example, as a genetic algorithm population), then one instance of method B is initialized with multiple solutions. If multiple solutions are returned from method A but method B only can accept one initial starting point, then method B is run several times, each one with a separate starting point from the results of method A. Table 4.2 summarizes the sequential hybrid strategy inputs.

In the `embedded` approach, global and local method strings supplied with the `global_method_pointer`

Description	Keyword	Associated Data	Status	Default
Hybrid strategy	hybrid	none	Required group (1 of 4 selections)	N/A
Sequential hybrid	sequential	none	Required group (1 of 3 selections)	N/A
List of methods	method_list	list of strings	Required	N/A

Table 3.2: Specification detail for sequential hybrid strategies

and local_method_pointer specifications identify the two methods to be used. The local_search_probability setting is an optional specification for supplying the probability (between 0.0 and 1.0) of employing local search to improve estimates within the global search. Table 4.3 summarizes the embedded hybrid strategy inputs.

Description	Keyword	Associated Data	Status	Default
Hybrid strategy	hybrid	none	Required group (1 of 4 selections)	N/A
Embedded hybrid	embedded	none	Required group (1 of 3 selections)	N/A
Pointer to the global method specification	global_method_pointer	string	Required	N/A
Pointer to the local method specification	local_method_pointer	string	Required	N/A
Probability of executing local searches	local_search_probability	real	Optional	0.1

Table 3.3: Specification detail for embedded hybrid strategies

In the collaborative approach, a list of method strings supplied with the method_list specification specifies the pool of iterators to be used. Any number of iterators may be specified. The method collaboration logic follows that of either the Agent-Based Optimization or HOPSPACK codes and is currently under development. Table 4.4 summarizes the collaborative hybrid strategy inputs.

Description	Keyword	Associated Data	Status	Default
Hybrid strategy	hybrid	none	Required group (1 of 4 selections)	N/A
Collaborative hybrid	collaborative	none	Required group (1 of 3 selections)	N/A
List of methods	method_list	list of strings	Required	N/A

Table 3.4: Specification detail for collaborative hybrid strategies

3.5 Multistart Iteration Commands

The `multi_start` strategy must specify an iterator using `method_pointer`. This iterator is responsible for completing a series of iterative analyses from a set of different starting points. These starting points can be specified as follows: (1) using `random_starts`, for which the specified number of starting points are selected randomly within the variable bounds, (2) using `starting_points`, in which the starting values are provided in a list, or (3) using both `random_starts` and `starting_points`, for which the combined set of points will be used. In aggregate, at least one starting point must be specified. The most common example of a multi-start strategy is multi-start optimization, in which a series of optimizations are performed from different starting values for the design variables. This can be an effective approach for problems with multiple minima. Table 4.5 summarizes the multi-start strategy inputs.

Description	Keyword	Associated Data	Status	Default
Multi-start iteration strategy	`multi_start`	none	Required group (1 of 4 selections)	N/A
Method pointer	`method_pointer`	string	Required	N/A
Number of random starting points	`random_starts`	integer	Optional group	no random starting points
Seed for random starting points	`seed`	integer	Optional	system-generated seed
List of user-specified starting points	`starting_points`	list of reals	Optional	no user-specified starting points

Table 3.5: Specification detail for multi-start strategies

3.6 Pareto Set Optimization Commands

The `pareto_set` strategy must specify an optimization or least squares calibration method using `method_pointer`. This minimizer is responsible for computing a set of optimal solutions from a set of response weightings (multi-objective weights or least squares term weights). These weightings can be specified as follows: (1) using `random_weight_sets`, in which case weightings are selected randomly within [0,1] bounds, (2) using `weight_sets`, in which the weighting sets are specified in a list, or (3) using both `random_weight_sets` and `weight_sets`, for which the combined set of weights will be used. In aggregate, at least one set of weights must be specified. The set of optimal solutions is called the "pareto set," which can provide valuable design trade-off information when there are competing objectives. Table 4.6 summarizes the pareto set strategy inputs.

3.7 Single Method Commands

The single method strategy is the default if no strategy specification is included in a user input file. It may also be specified using the `single_method` keyword within a strategy specification. An optional `method_pointer` specification may be used to point to a particular method specification. If `method_pointer` is not used, then the last method specification parsed will be used as the iterator. Table 4.7 summarizes the single method strategy inputs.

Description	Keyword	Associated Data	Status	Default
Pareto set optimization strategy	`pareto_set`	none	Required group (1 of 4 selections)	N/A
Optimization method pointer	`method_-pointer`	string	Required	N/A
Number of random weighting sets	`random_-weight_sets`	integer	Optional	no random weighting sets
Seed for random weighting sets	`seed`	integer	Optional	system-generated seed
List of user-specified weighting sets	`weight_sets`	list of reals	Optional	no user-specified weighting sets

Table 3.6: Specification detail for pareto set strategies

Description	Keyword	Associated Data	Status	Default
Single method strategy	`single_-method`	string	Required group (1 of 4 selections)	N/A
Method pointer	`method_-pointer`	string	Optional	use of last method parsed

Table 3.7: Specification detail for single method strategies

Chapter 4

Method Commands

4.1 Method Description

The method section in a Dakota input file specifies the name and controls of an iterator. The terms "method" and "iterator" can be used interchangeably, although method often refers to an input specification whereas iterator usually refers to an object within the **Iterator** hierarchy. A method specification, then, is used to select an iterator from the iterator hierarchy, which includes optimization, uncertainty quantification, least squares, design of experiments, and parameter study iterators (see the Users Manual [Adams et al., 2010] for more information on these iterator branches). This iterator may be used alone or in combination with other iterators as dictated by the strategy specification (refer to Strategy Commands for strategy command syntax and to the Users Manual [Adams et al., 2010] for strategy algorithm descriptions).

Several examples follow. The first example shows a minimal specification for an optimization method.

```
method,
dot_sqp
```

This example uses all of the defaults for this method.

A more sophisticated example would be

```
method,
id_method = 'NLP1'
model_pointer = 'M1'
dot_sqp
  max_iterations = 50
  convergence_tolerance = 1e-4
  output verbose
```

This example demonstrates the use of identifiers and pointers (see Method Independent Controls) as well as some method independent and method dependent controls for the sequential quadratic programming (SQP) algorithm from the DOT library. The `max_iterations`, `convergence_tolerance`, and `output` settings are method independent controls, in that they are defined for a variety of methods (see DOT method independent controls for DOT usage of these controls).

The next example shows a specification for a least squares method.

```
method,
```

```
optpp_g_newton
  max_iterations = 10
  convergence_tolerance = 1.e-8
  search_method trust_region
  gradient_tolerance = 1.e-6
```

Some of the same method independent controls are present along with several method dependent controls (`search_method` and `gradient_tolerance`) which are only meaningful for OPT++ methods (see OPT++ method dependent controls).

The next example shows a specification for a nondeterministic method with several method dependent controls (refer to Nondeterministic sampling method).

```
method,
sampling
  samples = 100 seed = 12345
  sample_type lhs
  response_levels = 1000. 500.
```

The last example shows a specification for a parameter study method where, again, each of the controls are method dependent (refer to Vector parameter study).

```
method,
vector_parameter_study
  step_vector = 1. 1. 1.
  num_steps = 10
```

4.2 Method Specification

As alluded to in the examples above, the method specification has the following structure:

```
method,
<method independent controls>
<method selection>
  <method dependent controls>
```

where `<method selection>` is one of the following: `dot_frcg`, `dot_mmfd`, `dot_bfgs`, `dot_slp`, `dot_sqp`, `conmin_frcg`, `conmin_mfd`, `npsol_sqp`, `nlssol_sqp`, `nlpql_sqp`, `nl2sol`, `nonlinear_cg`, `optpp_cg`, `optpp_q_newton`, `optpp_fd_newton`, `optpp_g_newton`, `optpp_newton`, `optpp_pds`, `asynch_pattern_search`, `coliny_cobyla`, `coliny_direct`, `coliny_pattern_search`, `coliny_solis_wets`, `coliny_ea`, `moga`, `soga`, `ncsu_direct`, `dl_solver`, `surrogate_based_local`, `surrogate_based_global`, `efficient_global`, `polynomial_chaos`, `stoch_collocation`, `sampling`, `importance_sampling`, `adaptive_sampling`, `local_reliability`, `global_reliability`, `local_evidence`, `global_evidence`, `local_interval_est`, `global_interval_est`, `bayes_calibration`, `dace`, `fsu_quasi_mc`, `fsu_cvt`, `psuade_moat`, `vector_parameter_study`, `list_parameter_study`, `centered_parameter_study`, or `multidim_parameter_study`.

The `<method independent controls>` are those controls which are valid for a variety of methods. In some cases, these controls are abstractions which may have slightly different implementations from one method to the next. The `<method dependent controls>` are those controls which are only meaningful for a specific method or library. Referring to dakota.input.summary, the method independent controls are those controls defined externally from and prior to the method selection blocks. They are all optional. The method selection blocks are

all required group specifications separated by logical OR's. The method dependent controls are those controls defined within the method selection blocks. Defaults for method independent and method dependent controls are defined in **DataMethod**. The following sections provide additional detail on the method independent controls followed by the method selections and their corresponding method dependent controls.

4.3 Method Independent Controls

The method independent controls include a method identifier string, a model type specification with pointers to variables, interface, and responses specifications, a speculative gradient selection, an output verbosity control, maximum iteration and function evaluation limits, constraint and convergence tolerance specifications, a scaling selection, and a set of linear inequality and equality constraint specifications. While each of these controls is not valid for every method, the controls are valid for enough methods that it was reasonable to pull them out of the method dependent blocks and consolidate the specifications.

The method identifier string is supplied with `id_method` and is used to provide a unique identifier string for use with strategy specifications (refer to Strategy Description). It is appropriate to omit a method identifier string if only one method is included in the input file and `single_method` is the selected strategy (all other strategies require one or more method pointers), since the single method to use is unambiguous in this case.

The model pointer string is specified with `model_pointer` and is used to identify the model used to perform function evaluations for the method. If a model pointer string is specified and no corresponding id is available, Dakota will exit with an error message. If no model pointer string is specified, then the last model specification parsed will be used. If no model pointer string is specified and no model specification is provided by the user, then a default model specification is used (similar to the default strategy specification, see Strategy Description). This default model specification is of type `single` with no `variables_pointer`, `interface_pointer`, or `responses_pointer` (see Single Model Controls). It is appropriate to omit a model specification whenever the relationships are unambiguous due to the presence of single variables, interface, and responses specifications.

When performing gradient-based optimization in parallel, `speculative` gradients can be selected to address the load imbalance that can occur between gradient evaluation and line search phases. In a typical gradient-based optimization, the line search phase consists primarily of evaluating the objective function and any constraints at a trial point, and then testing the trial point for a sufficient decrease in the objective function value and/or constraint violation. If a sufficient decrease is not observed, then one or more additional trial points may be attempted sequentially. However, if the trial point is accepted then the line search phase is complete and the gradient evaluation phase begins. By speculating that the gradient information associated with a given line search trial point will be used later, additional coarse grained parallelism can be introduced by computing the gradient information (either by finite difference or analytically) in parallel, at the same time as the line search phase trial-point function values. This balances the total amount of computation to be performed at each design point and allows for efficient utilization of multiple processors. While the total amount of work performed will generally increase (since some speculative gradients will not be used when a trial point is rejected in the line search phase), the run time will usually decrease (since gradient evaluations needed at the start of each new optimization cycle were already performed in parallel during the line search phase). Refer to [Byrd et al., 1998] for additional details. The speculative specification is implemented for the gradient-based optimizers in the DOT, CONMIN, and OPT++ libraries, and it can be used with dakota numerical or analytic gradient selections in the responses specification (refer to Gradient Specification for information on these specifications). It should not be selected with vendor numerical gradients since vendor internal finite difference algorithms have not been modified for this purpose. In full-Newton approaches, the Hessian is also computed speculatively. NPSOL and NLSSOL do not support speculative gradients, as their gradient-based line search in user-supplied gradient mode (dakota numerical or analytic gradients) is a superior approach for load-balanced parallel execution.

Output verbosity control is specified with `output` followed by `silent`, `quiet`, `verbose` or `debug`. If there is no user specification for output verbosity, then the default setting is `normal`. This gives a total of five output levels to manage the volume of data that is returned to the user during the course of a study, ranging from full run annotation plus internal debug diagnostics (`debug`) to the bare minimum of output containing little more than the total number of simulations performed and the final solution (`silent`). Output verbosity is observed within the **Iterator** (algorithm verbosity), **Model** (synchronize/fd_gradients verbosity), **Interface** (map/synch verbosity), **Approximation** (global data fit coefficient reporting),and **AnalysisCode** (file operation reporting) class hierarchies; however, not all of these software components observe the full granularity of verbosity settings. Specific mappings are as follows:

- `output silent` (i.e., really quiet): silent iterators, silent model, silent interface, quiet approximation, quiet file operations
- `output quiet`: quiet iterators, quiet model, quiet interface, quiet approximation, quiet file operations
- `output normal`: normal iterators, normal model, normal interface, quiet approximation, quiet file operations
- `output verbose`: verbose iterators, normal model, verbose interface, verbose approximation, verbose file operations
- `output debug` (i.e., really verbose): debug iterators, normal model, debug interface, verbose approximation, verbose file operations

Note that iterators and interfaces utilize the full granularity in verbosity, whereas models, approximations, and file operations do not. With respect to iterator verbosity, different iterators implement this control in slightly different ways (as described below in the method independent controls descriptions for each iterator), however the meaning is consistent. For models, interfaces, approximations, and file operations, `quiet` suppresses parameter and response set reporting and `silent` further suppresses function evaluation headers and scheduling output. Similarly, `verbose` adds file management, approximation evaluation, and global approximation coefficient details, and `debug` further adds diagnostics from nonblocking schedulers.

The `constraint_tolerance` specification determines the maximum allowable value of infeasibility that any constraint in an optimization problem may possess and still be considered to be satisfied. It is specified as a positive real value. If a constraint function is greater than this value then it is considered to be violated by the optimization algorithm. This specification gives some control over how tightly the constraints will be satisfied at convergence of the algorithm. However, if the value is set too small the algorithm may terminate with one or more constraints being violated. This specification is currently meaningful for the NPSOL, NLSSOL, DOT and CONMIN constrained optimizers (refer to DOT method independent controls and NPSOL method independent controls).

The `convergence_tolerance` specification provides a real value for controlling the termination of iteration. In most cases, it is a relative convergence tolerance for the objective function; i.e., if the change in the objective function between successive iterations divided by the previous objective function is less than the amount specified by convergence_tolerance, then this convergence criterion is satisfied on the current iteration. Since no progress may be made on one iteration followed by significant progress on a subsequent iteration, some libraries require that the convergence tolerance be satisfied on two or more consecutive iterations prior to termination of iteration. This control is used with optimization and least squares iterators (DOT, CONMIN, NPSOL, NLSSOL, OPT++, and SCOLIB) and is not used within the uncertainty quantification, design of experiments, or parameter study iterator branches. Refer to DOT method independent controls, NPSOL method independent controls, OPT++ method independent controls, and SCOLIB method independent controls for specific interpretations of the `convergence_tolerance` specification.

The `max_iterations` and `max_function_evaluations` controls provide integer limits for the maximum number of iterations and maximum number of function evaluations, respectively. The difference between an iteration and a function evaluation is that a function evaluation involves a single parameter to response mapping through an interface, whereas an iteration involves a complete cycle of computation within the iterator. Thus, an iteration generally involves multiple function evaluations (e.g., an iteration contains descent direction and line search computations in gradient-based optimization, population and multiple offset evaluations in nongradient-based optimization, etc.). The `max_function_evaluations` control is not currently used within the uncertainty quantification, design of experiments, and parameter study iterator branches, and in the case of gradient-based methods, does not currently capture function evaluations that occur as part of the `method_source dakota` finite difference routine (since these additional evaluations are intentionally isolated from the iterators).

The `final_solutions` controls the number of final solutions returned by the iterator as the best solutions. For most optimizers, this is one, but some optimizers can produce multiple solutions (e.g. genetic algorithms). In the case of analyzers such as sampling methods, if one specifies 100 samples (for example) but also specifies `final_solutions` = 5, the five best solutions (in order of lowest response function value) are returned. When using a `hybrid` strategy, the number of final solutions dictates how many solutions are passed from one method to another.

Continuous design variable, function, and constraint scaling can be turned on for optimizers and least squares minimizers by providing the `scaling` keyword. Discrete variable scaling is not supported. When scaling is enabled, variables, functions, gradients, Hessians, etc., are transformed such that the optimizer iterates in scaled variable space, whereas evaluations of the computational model as specified in the interface are performed on the original problem scale. Therefore using scaling does not require rewriting the interface to the simulation code. The user may specify no, one, or a vector of scaling type strings through each of the `scale_types` (see Variables Commands); `objective_function_scale_types`, `calibration_term_scale_types`, `nonlinear_inequality_scale_types`, `nonlinear_equality_scale_types` (see Function Specification); `linear_inequality_scale_types`, and `linear_equality_scale_types` (see Method Independent Controls below) specifications. Valid options for types include `'none'` (default), `'value'`, `'auto'`, or `'log'`, for no, characteristic value, automatic, or logarithmic scaling, respectively, although not all types are valid for scaling all entities (see the references for details). If a single string is specified using any of these keywords it will apply to each component of the relevant vector, e.g., `scale_types = 'value'` will enable characteristic value scaling for each continuous design variable. The user may specify no, one, or a vector of nonzero characteristic scale values through each of the `scales` (see Variables Commands); `objective_function_scales`, `calibration_term_scales`, `nonlinear_inequality_scales`, `nonlinear_equality_scales` (see Function Specification); `linear_inequality_scales`, and `linear_equality_scales` (see Method Independent Controls below) specifications. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'` and `'log'`. If a single value is specified using any of these keywords it will apply to each component of the relevant vector, e.g., `scales = 3.0` will apply a characteristic scaling value of `3.0` to each continuous design variable. When the `scaling` keyword is omitted, all `_scale_types` and `*_scales` specifications are ignored in the method, variables, and responses sections.

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

- None (`'none'`): no scaling performed (`*_scales` ignored) on this component
- Characteristic value (`'value'`): the corresponding quantity is scaled by the (required) characteristic value

provided in the `*_scales` specification. If the scale value is negative, the sense of inequalities are changed accordingly.

- Automatic (`'auto'`): First, any characteristic values from the optional `*_scales` specification are applied. Then, automatic scaling will be attempted according to the following scheme:
  - two-sided bounds scaled into the interval [0,1];
  - one-sided bound or targets are scaled by the characteristic value, moving the bound or target to 1 and changing the sense of inequalities where necessary;
  - no bounds or targets: no automatic scaling possible, therefore no scaling for this component

  Automatic scaling is not available for objective functions nor calibration terms since they lack bound constraints. Futher, when automatically scaled, linear constraints are scaled by characteristic values only, not affinely scaled into [0,1].

- Logarithmic (`'log'`): First, any characteristic values from the optional `*_scales` specification are applied. Then, logarithm base 10 scaling is applied. Logarithmic scaling is not available for linear constraints. Further, when continuous design variables are log scaled, linear constraints are not allowed.

Table 5.1 provides the specification detail for the method independent controls involving identifiers, pointers, output verbosity, speculative gradients, and scaling, and Table 5.2 provides the specification detail for the method independent controls involving convergence tolerances and iteration limits.

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Method set identifier	`id_method`	string	Optional	strategy use of last method parsed
Model pointer	`model_-pointer`	string	Optional	method use of last model parsed (or use of default model if none parsed)
Speculative gradients and Hessians	`speculative`	none	Optional	no speculation
Output verbosity	`output`	`silent` | `quiet` | `normal` | `verbose` | `debug`	Optional	`normal`
Final solutions	`final_-solutions`	integer	Optional	1
Scaling flag	`scaling`	none	Optional	no scaling

Table 4.1: Specification detail for the method independent controls: identifiers, pointers, output verbosity, speculative gradients, and scaling

Linear inequality constraints can be supplied with the `linear_inequality_constraint_matrix`, `linear_inequality_lower_bounds`, and `linear_inequality_upper_bounds` specifications, and linear equality constraints can be supplied with the `linear_equality_constraint_matrix` and `linear_equality_targets` specifications. In the inequality case, the constraint matrix provides coefficients for the variables and the lower and upper bounds provide constraint limits for the following two-sided

Description	Keyword	Associated Data	Status	Default
Maximum iterations	`max_-iterations`	integer	Optional	100 (exceptions: `fsu_cvt`, `local_-reliability`: 25; `global_-{reliability, interval_est, evidence}` / `efficient_-global`: 25∗n)
Maximum function evaluations	`max_-function_-evaluations`	integer	Optional	1000
Constraint tolerance	`constraint_-tolerance`	real	Optional	Library default
Convergence tolerance	`convergence_-tolerance`	real	Optional	1.e-4

Table 4.2: Specification detail for the method independent controls: convergence tolerances and iteration limits

formulation:

$$a_l \leq Ax \leq a_u$$

As with nonlinear inequality constraints (see Objective and constraint functions (optimization data set)), the default linear inequality constraint bounds are selected so that one-sided inequalities of the form

$$Ax \leq 0.0$$

result when there are no user bounds specifications (this provides backwards compatibility with previous Dakota versions). In a user bounds specification, any upper bound values greater than `+bigRealBoundSize` (1.e+30, as defined in **Minimizer**) are treated as +infinity and any lower bound values less than `-bigRealBoundSize` are treated as -infinity. This feature is commonly used to drop one of the bounds in order to specify a 1-sided constraint (just as the default lower bounds drop out since `-DBL_MAX` < `-bigRealBoundSize`). In the equality case, the constraint matrix again provides coefficients for the variables and the targets provide the equality constraint right hand sides:

$$Ax = a_t$$

and the defaults for the equality constraint targets enforce a value of `0.` for each constraint

$$Ax = 0.0$$

Currently, APPS, CONMIN, DOT, JEGA, NLSSOL, NLPQL, NPSOL, OPT++, and SCOLIB all support specialized handling of linear constraints (either directly through the algorithm itself or indirectly through the Dakota wrapper). Linear constraints need not be computed by the user's interface on every function evaluation; rather the coefficients, bounds, and targets of the linear constraints can be provided at start up, allowing the optimizers to track the linear constraints internally. It is important to recognize that linear constraints are those constraints that are linear in the *design* variables, e.g.:

$$0.0 \leq 3x_1 - 4x_2 + 2x_3 \leq 15.0$$

$$x_1 + x_2 + x_3 \geq 2.0$$
$$x_1 + x_2 - x_3 = 1.0$$

which is not to be confused with something like

$$s(X) - s_{fail} \leq 0.0$$

where the constraint is linear in a response quantity, but may be a nonlinear implicit function of the design variables. For the three linear constraints above, the specification would appear as:

```
linear_inequality_constraint_matrix =  3.0 -4.0  2.0
                                       1.0  1.0  1.0
linear_inequality_lower_bounds =       0.0  2.0
linear_inequality_upper_bounds =      15.0  1.e+50
linear_equality_constraint_matrix =    1.0  1.0 -1.0
linear_equality_targets =              1.0
```

where the `1.e+50` is a dummy upper bound value which defines a 1-sided inequality since it is greater than `bigRealBoundSize`. The constraint matrix specifications list the coefficients of the first constraint followed by the coefficients of the second constraint, and so on. They are divided into individual constraints based on the number of design variables, and can be broken onto multiple lines for readability as shown above.

The `linear_inequality_scale_types` and `linear_equality_scale_types` specifications provide strings specifying the scaling type for each linear inequality or equality constraint, respectively, in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `linear_*_scale_types` may be selected from `'none'`, `'value'`, or `'auto'` to select no, characteristic value, or automatic scaling, respectively. If a single string is specified it will apply to each constraint component. Each entry in `linear_inequality_scales` or `linear_equality_scales` may be a user-specified nonzero characteristic value to be used in scaling each constraint. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'`. If a single real value is specified it will apply to all components of the constraint. Scaling for linear constraints is applied *after* any continuous variable scaling. For example, for variable scaling on continuous design variables x:

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

we have the following system for linear inequality constraints

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

and user-specified or automatically computed scaling multipliers are appplied to this final transformed system, which accounts for continuous design variable scaling. When automatic scaling is in use for linear constraints they are linearly scaled by a computed characteristic value, but not affinely to [0,1].

Table 5.3 provides the specification detail for the method independent controls involving linear constraints.

4.4 Optimization Methods

The Dakota project started as an toolbox for optimization methods, and has accumulated a broad variety of gradient-based and nongradient-based optimizers from the DOT, NPSOL, NLPQL, CONMIN, OPT++, APPS, SCOLIB, NCSU, and JEGA packages. These capabilities are described below.

Description	Keyword	Associated Data	Status	Default
Linear inequality coefficient matrix	`linear_inequality_constraint_matrix`	list of reals	Optional	no linear inequality constraints
Linear inequality lower bounds	`linear_inequality_lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Linear inequality upper bounds	`linear_inequality_upper_bounds`	list of reals	Optional	vector values = `0.`
Linear inequality scaling types	`linear_inequality_scale_types`	list of strings	Optional	vector values = `'none'`
Linear inequality scales	`linear_inequality_scales`	list of reals	Optional	vector values = `1.` (no scaling)
Linear equality coefficient matrix	`linear_equality_constraint_matrix`	list of reals	Optional	no linear equality constraints
Linear equality targets	`linear_equality_targets`	list of reals	Optional	vector values = `0.`
Linear equality scaling types	`linear_equality_scale_types`	list of strings	Optional	vector values = `'none'`
Linear equality scales	`linear_equality_scales`	list of reals	Optional	vector values = `1.` (no scaling)

Table 4.3: Specification detail for the method independent controls: linear inequality and equality constraints

4.4.1 DOT Methods

The DOT library [Vanderplaats Research and Development, 1995] contains nonlinear programming optimizers, specifically the Broyden-Fletcher-Goldfarb-Shanno (Dakota's `dot_bfgs` method) and Fletcher-Reeves conjugate gradient (Dakota's `dot_frcg` method) methods for unconstrained optimization, and the modified method of feasible directions (Dakota's `dot_mmfd` method), sequential linear programming (Dakota's `dot_slp` method), and sequential quadratic programming (Dakota's `dot_sqp` method) methods for constrained optimization. Dakota provides access to the DOT library through the **DOTOptimizer** class.

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

4.4.1.1 DOT method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during a DOT optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. This convergence criterion must be satisfied for two consecutive iterations before DOT will terminate. The `constraint_tolerance` specification defines how tightly constraint functions are to be satisfied at convergence. The default value for DOT constrained optimizers is 0.003. Extremely small values for constraint_tolerance may not be attainable. The output verbosity specification controls the amount of information generated by DOT: the `silent` and `quiet` settings result in header information, final results, and objective function, constraint, and parameter information on each iteration; whereas the `verbose` and `debug` settings add additional information on gradients, search direction, one-dimensional search results, and parameter scaling factors. DOT contains no parallel algorithms which can directly take advantage of concurrent evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, then the finite difference function evaluations can be performed concurrently (using any of the parallel modes described in the Users Manual [Adams et al., 2010]). In addition, if `speculative` is specified, then gradients (`dakota numerical` or `analytic` gradients) will be computed on each line search evaluation in order to balance the load and lower the total run time in parallel optimization studies. Lastly, specialized handling of linear constraints is supported with DOT; linear constraint coefficients, bounds, and targets can be provided to DOT at start-up and tracked internally. Specification detail for these method independent controls is provided in Tables 5.1 through 5.3.

4.4.1.2 DOT method dependent controls

DOT does not currently support any method dependent controls.

4.4.2 NPSOL Method

The NPSOL library [Gill et al., 1986] contains a sequential quadratic programming (SQP) implementation (the `npsol_sqp` method). SQP is a nonlinear programming optimizer for constrained minimization. Dakota provides access to the NPSOL library through the **NPSOLOptimizer** class.

4.4.2.1 NPSOL method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major SQP iterations and the number of function evaluations that can be performed during an NPSOL optimization. The `convergence_tolerance` control defines NPSOL's internal optimality tolerance which is used in evaluating if an iterate satisfies the first-order Kuhn-Tucker conditions for a minimum. The magnitude of `convergence_tolerance` approximately specifies the number of significant digits of accuracy desired in the final objective function (e.g., `convergence_tolerance = 1.e-6` will result in approximately six digits of accuracy in the final objective function). The `constraint_tolerance` control defines how tightly the constraint functions are satisfied at convergence. The default value is dependent upon the machine precision of the platform in use, but is typically on the order of `1.e-8` for double precision computations. Extremely small values for `constraint_tolerance` may not be attainable. The `output` verbosity setting controls the amount of information generated at each major SQP iteration: the `silent` and `quiet` settings result in only one line of diagnostic output for each major iteration and print the final optimization solution, whereas the `verbose` and `debug` settings add additional information on the objective function, constraints, and variables at each major iteration.

NPSOL is not a parallel algorithm and cannot directly take advantage of concurrent evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, then the finite difference function evaluations can be performed concurrently (using any of the parallel modes described in the Users Manual [Adams et al., 2010]). An important related observation is the fact that NPSOL uses two different line searches depending on how gradients are computed. For either `analytic_gradients` or `numerical_gradients` with `method_source dakota`, NPSOL is placed in user-supplied gradient mode (NPSOL's "Derivative Level" is set to 3) and it uses a gradient-based line search (the assumption is that user-supplied gradients are inexpensive). On the other hand, if `numerical_gradients` are selected with `method_source vendor`, then NPSOL is computing finite differences internally and it will use a value-based line search (the assumption is that finite differencing on each line search evaluation is too expensive). The ramifications of this are: (1) performance will vary between `method_source dakota` and `method_source vendor` for `numerical_gradients`, and (2) gradient speculation is unnecessary when performing optimization in parallel since the gradient-based line search in user-supplied gradient mode is already load balanced for parallel execution. Therefore, a `speculative` specification will be ignored by NPSOL, and optimization with numerical gradients should select `method_source dakota` for load balanced parallel operation and `method_source vendor` for efficient serial operation.

Lastly, NPSOL supports specialized handling of linear inequality and equality constraints. By specifying the coefficients and bounds of the linear inequality constraints and the coefficients and targets of the linear equality constraints, this information can be provided to NPSOL at initialization and tracked internally, removing the need for the user to provide the values of the linear constraints on every function evaluation. Refer to Method Independent Controls for additional information and to Tables 5.1 through 5.3 for method independent control specification detail.

4.4.2.2 NPSOL method dependent controls

NPSOL's method dependent controls are `verify_level`, `function_precision`, and `linesearch_tolerance`. The `verify_level` control instructs NPSOL to perform finite difference verifications on user-supplied gradient components. The `function_precision` control provides NPSOL an estimate of the accuracy to which the problem functions can be computed. This is used to prevent NPSOL from trying to distinguish between function values that differ by less than the inherent error in the calculation. And the `linesearch_tolerance` setting controls the accuracy of the line search. The smaller the value (between 0 and 1), the more accurately NPSOL will attempt to compute a precise minimum along the search direction. Table 5.4 provides the specification detail for the NPSOL SQP method and its method dependent controls.

Description	Keyword	Associated Data	Status	Default
Gradient verification level	`verify_level`	integer	Optional	-1 (no gradient verification)
Function precision	`function_-precision`	real	Optional	1.e-10
Line search tolerance	`linesearch_-tolerance`	real	Optional	0.9 (inaccurate line search)

Table 4.4: Specification detail for the NPSOL SQP method

4.4.3 NLPQL Methods

The NLPQL library is a commercially-licensed library containing a sequential quadratic programming (SQP) optimizer, specified as Dakota's `nlpql_sqp` method, for constrained optimization. The particular implementation used is NLPQLP [Schittkowski, 2004], a variant with distributed and non-monotone line search. Dakota provides access to the NLPQL library through the **NLPQLPOptimizer** class.

4.4.3.1 NLPQL method independent controls

The method independent controls for maximum iterations and output verbosity are mapped to NLPQL controls MAXIT and IPRINT, respectively. The maximum number of function evaluations is enforced within the **NLPQLPOptimizer** class.

4.4.3.2 NLPQL method dependent controls

NLPQL does not currently support any method dependent controls.

4.4.4 CONMIN Methods

The CONMIN library [Vanderplaats, 1973] is a public domain library of nonlinear programming optimizers, specifically the Fletcher-Reeves conjugate gradient (Dakota's `conmin_frcg` method) method for unconstrained optimization, and the method of feasible directions (Dakota's `conmin_mfd` method) for constrained optimization. As CONMIN was a predecessor to the DOT commercial library, the algorithm controls are very similar. Dakota provides access to the CONMIN library through the **CONMINOptimizer** class.

4.4.4.1 CONMIN method independent controls

The interpretations of the method independent controls for CONMIN are essentially identical to those for DOT. Therefore, the discussion in DOT method independent controls is relevant for CONMIN.

4.4.4.2 CONMIN method dependent controls

CONMIN does not currently support any method dependent controls.

4.4.5 OPT++ Methods

The OPT++ library [Meza et al., 2007] contains primarily gradient-based nonlinear programming optimizers for unconstrained, bound-constrained, and nonlinearly constrained minimization: Polak-Ribiere conjugate gradient (Dakota's `optpp_cg` method), quasi-Newton (Dakota's `optpp_q_newton` method), finite difference Newton (Dakota's `optpp_fd_newton` method), and full Newton (Dakota's `optpp_newton` method). The conjugate gradient method is strictly unconstrained, and each of the Newton-based methods are automatically bound to the appropriate OPT++ algorithm based on the user constraint specification (unconstrained, bound-constrained, or generally-constrained). In the generally-constrained case, the Newton methods use a nonlinear interior-point approach to manage the constraints. The library also contains a direct search algorithm, PDS (parallel direct search, Dakota's `optpp_pds` method), which supports bound constraints. Dakota provides access to the OPT++ library through the **SNLLOptimizer** class, where "SNLL" denotes Sandia National Laboratories - Livermore.

4.4.5.1 OPT++ method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during an OPT++ optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. The `output` verbosity specification controls the amount of information generated from OPT++ executions: the `debug` setting turns on OPT++'s internal debug mode and also generates additional debugging information from Dakota's **SNLLOptimizer** wrapper class. OPT++'s gradient-based methods are not parallel algorithms and cannot directly take advantage of concurrent function evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, a parallel Dakota configuration can utilize concurrent evaluations for the finite difference gradient computations. OPT++'s nongradient-based PDS method can directly exploit asynchronous evaluations; however, this capability has not yet been implemented in the **SNLLOptimizer** class.

The `speculative` specification enables speculative computation of gradient and/or Hessian information, where applicable, for parallel optimization studies. By speculating that the derivative information at the current point will be used later, the complete data set (all available gradient/Hessian information) can be computed on every function evaluation. While some of these computations will be wasted, the positive effects are a consistent parallel load balance and usually shorter wall clock time. The `speculative` specification is applicable only when parallelism in the gradient calculations can be exploited by Dakota (it will be ignored for `vendor numerical` gradients).

Lastly, linear constraint specifications are supported by each of the Newton methods (`optpp_newton`, `optpp_q_newton`, `optpp_fd_newton`, and `optpp_g_newton`); whereas `optpp_cg` must be unconstrained and `optpp_pds` can be, at most, bound-constrained. Specification detail for the method independent controls is provided in Tables 5.1 through 5.3.

4.4.5.2 OPT++ method dependent controls

OPT++'s method dependent controls are `max_step`, `gradient_tolerance`, `search_method`, `merit_function`, `steplength_to_boundary`, `centering_parameter`, and `search_scheme_size`. The `max_step` control specifies the maximum step that can be taken when computing a change in the current design point (e.g., limiting the Newton step computed from current gradient and Hessian information). It is equivalent to a move limit or a maximum trust region size. The `gradient_tolerance` control defines the threshold value on the L2 norm of the objective function gradient that indicates convergence to an unconstrained minimum (no active constraints). The `gradient_tolerance` control is defined for all gradient-based optimizers.

max_step and gradient_tolerance are the only method dependent controls for the OPT++ conjugate gradient method. Table 5.5 covers this specification.

Description	Keyword	Associated Data	Status	Default
OPT++ conjugate gradient method	optpp_cg	none	Required	N/A
Maximum step size	max_step	real	Optional	1000.
Gradient tolerance	gradient_-tolerance	real	Optional	1.e-4

Table 4.5: Specification detail for the OPT++ conjugate gradient method

The search_method control is defined for all Newton-based optimizers and is used to select between trust_region, gradient_based_line_search, and value_based_line_search methods. The gradient_based_line_search option uses the line search method proposed by [More and Thuente, 1994]. This option satisfies sufficient decrease and curvature conditions; whereas, value_base_line_search only satisfies the sufficient decrease condition. At each line search iteration, the gradient_based_line_search method computes the function and gradient at the trial point. Consequently, given expensive function evaluations, the value_based_line_search method is preferred to the gradient_based_line_search method. Each of these Newton methods additionally supports the tr_pds selection for unconstrained problems. This option performs a robust trust region search using pattern search techniques. Use of a line search is the default for bound-constrained and generally-constrained problems, and use of a trust_region search method is the default for unconstrained problems.

The merit_function, steplength_to_boundary, and centering_parameter selections are additional specifications that are defined for the solution of generally-constrained problems with nonlinear interior-point algorithms. A merit_function is a function in constrained optimization that attempts to provide joint progress toward reducing the objective function and satisfying the constraints. Valid options are el_bakry, argaez_tapia, or van_shanno, where user input is not case sensitive in this case. Details for these selections are as follows:

- The "el_bakry" merit function is the L2-norm of the first order optimality conditions for the nonlinear programming problem. The cost per linesearch iteration is n+1 function evaluations. For more information, see [El-Bakry et al., 1996].

- The "argaez_tapia" merit function can be classified as a modified augmented Lagrangian function. The augmented Lagrangian is modified by adding to its penalty term a potential reduction function to handle the perturbed complementarity condition. The cost per linesearch iteration is one function evaluation. For more information, see [Tapia and Argaez].

- The "van_shanno" merit function can be classified as a penalty function for the logarithmic barrier formulation of the nonlinear programming problem. The cost per linesearch iteration is one function evaluation. For more information see [Vanderbei and Shanno, 1999].

If the function evaluation is expensive or noisy, set the merit_function to "argaez_tapia" or "van_shanno".

The steplength_to_boundary specification is a parameter (between 0 and 1) that controls how close to the boundary of the feasible region the algorithm is allowed to move. A value of 1 means that the algorithm is allowed to take steps that may reach the boundary of the feasible region. If the user wishes to maintain strict feasibility

of the design parameters this value should be less than 1. Default values are .8, .99995, and .95 for the `el_bakry`, `argaez_tapia`, and `van_shanno` merit functions, respectively. The `centering_parameter` specification is a parameter (between 0 and 1) that controls how closely the algorithm should follow the "central path". See [Wright] for the definition of central path. The larger the value, the more closely the algorithm follows the central path, which results in small steps. A value of 0 indicates that the algorithm will take a pure Newton step. Default values are .2, .2, and .1 for the `el_bakry`, `argaez_tapia`, and `van_shanno` merit functions, respectively.

Table 5.6 provides the details for the Newton-based methods.

Description	Keyword	Associated Data	Status	Default
OPT++ Newton-based methods	`optpp_q_newton` | `optpp_fd_newton` | `optpp_newton`	none	Required group	N/A
Search method	`value_based_line_search` | `gradient_based_line_search` | `trust_region` | `tr_pds`	none	Optional group	`trust_region` (unconstrained), `value_based_line_search` (bound/general constraints)
Maximum step size	`max_step`	real	Optional	1000.
Gradient tolerance	`gradient_tolerance`	real	Optional	1.e-4
Merit function	`merit_function`	`argaez_tapia`, `el_bakry`, or `van_shanno`	Optional	`argaez_tapia`
Steplength to boundary	`steplength_to_boundary`	real	Optional	Merit function dependent: 0.8 (el_bakry), 0.99995 (argaez_tapia), 0.95 (van_shanno)
Centering parameter	`centering_parameter`	real	Optional	Merit function dependent: 0.2 (el_bakry), 0.2 (argaez_tapia), 0.1 (van_shanno)

Table 4.6: Specification detail for OPT++ Newton-based optimization methods

The `search_scheme_size` is defined for the PDS method to specify the number of points to be used in the direct search template. PDS does not support parallelism at this time due to current limitations in the OPT++ interface. Table 5.7 provides the detail for the parallel direct search method.

Description	Keyword	Associated Data	Status	Default
OPT++ parallel direct search method	optpp_pds	none	Required group	N/A
Search scheme size	search_-scheme_size	integer	Optional	32

Table 4.7: Specification detail for the OPT++ PDS method

4.4.6 Asynchronous Parallel Pattern Search (APPS)

The asynchronous parallel pattern search (APPS) algorithm [Gray and Kolda, 2006] is a fully asynchronous pattern search technique in that the search along each offset direction continues without waiting for searches along other directions to finish. It is now made available in Dakota through the HOPSPACK software [Plantenga, 2009]. It can handle unconstrained problems as well as those with bound constraints, linear constraints, and general nonlinear constraints. HOPSPACK is available to the public under the GNU LGPL and the source code is included with Dakota. HOPSPACK-specific software documentation is available from https://software.sandia.gov/trac/hopspack.

4.4.6.1 APPS method independent controls

The only method independent controls that are currently mapped to APPS are max_function_-evaluations, constraint_tolerance, and the output verbosity control. We note that while APPS treats the constraint tolerance separately for linear and nonlinear constraints, we apply the same value to both if the user specifies constraint_tolerance. The APPS internal "display" level is mapped to the Dakota debug, verbose, normal, quiet, and silent settings as follows:

- Dakota "debug": display final solution, all input parameters, variable and constraint info, trial points, search directions, and execution details
- Dakota "verbose": display final solution, all input parameters, variable and constraint info, and trial points
- Dakota "normal": display final solution, all input parameters, variable and constraint summaries, and new best points
- Dakota "quiet": display final solution and all input parameters
- Dakota "silent": display final solution

APPS exploits parallelism through the use of Dakota's concurrent function evaluations. The variant of the algorithm that is currently exposed, however, limits the amount of concurrency that can be exploited. In particular, APPS can leverage an evaluation concurrency level of at most twice the number of variables. More options that allow for greater evaluation concurrency may be exposed in future releases.

4.4.6.2 APPS method dependent controls

The APPS method is invoked using a asynch_pattern_search group specification. Some of the method dependent controls are similar to the SCOLIB controls for coliny_pattern_search described in Pattern Search. In particular, APPS supports the following step length control parameters:

- `initial_delta`: the initial step length, must be positive
- `threshold_delta`: step length used to determine convergence, must be greater than or equal to 4.4e-16
- `contraction_factor`: amount by which step length is rescaled after unsuccesful iterates, must be strictly between 0 and 1

If `initial_delta` is supplied by the user, it will be applied in an absolute sense in all coordinate directions. APPS documentation advocates choosing `initial_delta` to be the approximate distance from the initial point to the solution. If this is unknown, it is advisable to err on the side of choosing an `initial_delta` that is too large or to not specify it. In the latter case, APPS will take a full step to the boundary in each direction. Relative application of `initial_delta` is not available unless the user scales the problem accordingly.

When the solution to the optimization problem is known, the user may specify a value for `solution_target` as a termination criteria. APPS will terminate when the function value falls below `solution_target`.

Currently, APPS only supports coordinate bases with a total of *2n* function evaluations in the pattern, and these patterns may only contract. The `synchronization` specification can be used to specify the use of either `blocking` or `nonblocking` schedulers for APPS. The `blocking` option causes APPS to behave as a synchronous algorithm. The `nonblocking` option is not available when Dakota is used in message-passing mode.

APPS solves nonlinearly constrained problems by solving a sequence of linearly constrained merit function-base subproblems. There are several exact and smoothed exact penalty functions that can be specified with the `merit_function` control. The options are as follows:

- `merit_max`: based on ℓ_∞ norm
- `merit_max_smooth`: based on smoothed ℓ_∞ norm
- `merit1`: based on ℓ_1 norm
- `merit1_smooth`: based on smoothed ℓ_1 norm
- `merit2`: based on ℓ_2 norm
- `merit2_smooth`: based on smoothed ℓ_2 norm
- `merit2_squared`: based on ℓ_2^2 norm

The user can also specify the following:

- `constraint_penalty`: the initial multiplier for the penalty function, must be nonnegative
- `smoothing_parameter`: initial smoothing value for smoothed penalty functions, must be between 0 and 1 (inclusive)

Table 5.8 summarizes the APPS specification.

Description	Keyword	Associated Data	Status	Default
APPS method	asynch_-pattern_-search	none	Required group	N/A
Initial offset value	initial_-delta	real	Optional	1.0
Threshold for offset values	threshold_-delta	real	Optional	0.01
Pattern contraction factor	contraction_-factor	real	Optional	0.5
Solution target	solution_-target	real	Optional	not used
Evaluation synchronization	synchronization	blocking| nonblocking	Optional	nonblocking
Merit function	merit_-function	merit_max| merit_max_-smooth| merit1| merit1_-smooth| merit2| merit2_-smooth| merit2_-squared	Optional	merit2_-squared
Constraint penalty	constraint_-penalty	real	Optional	1.0
Smoothing factor	smoothing_-factor	real	Optional	0.0

Table 4.8: Specification detail for the APPS method

4.4.7 SCOLIB Methods

SCOLIB (formerly known as COLINY) is a collection of nongradient-based optimizers that support the Common Optimization Library INterface (COLIN). SCOLIB optimizers currently include `coliny_cobyla`, `coliny_direct`, `coliny_ea`, `coliny_pattern_search` and `coliny_solis_wets`. (Yes, the input spec still has "coliny" prepended to the method name.) Additional SCOLIB information is available from https://software.sandia.gov/trac/acro.

SCOLIB solvers now support bound constraints and general nonlinear constraints. Supported nonlinear constraints include both equality and two-sided inequality constraints. SCOLIB solvers do not yet support linear constraints. Most SCOLIB optimizers treat constraints with a simple penalty scheme that adds `constraint_penalty` times the sum of squares of the constraint violations to the objective function. Specific exceptions to this method for handling constraint violations are noted below. (The default value of `constraint_penalty` is 1000.0, except for methods that dynamically adapt their constraint penalty, for which the default value is 1.0.)

4.4.7.1 SCOLIB method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during a SCOLIB optimization, respectively. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. The `output` verbosity specification controls the amount of information generated by SCOLIB: the `silent`, `quiet`, and `normal` settings correspond to minimal reporting from SCOLIB, whereas the `verbose` setting corresponds to a higher level of information, and `debug` outputs method initialization and a variety of internal SCOLIB diagnostics. The majority of SCOLIB's methods perform independent function evaluations that can directly take advantage of Dakota's parallel capabilities. Only `coliny_solis_wets`, `coliny_cobyla`, and certain configurations of `coliny_pattern_search` are inherently serial (see Pattern Search). The parallel methods automatically utilize parallel logic when the Dakota configuration supports parallelism. Lastly, neither `speculative` gradients nor linear constraints are currently supported with SCOLIB. Specification detail for method independent controls is provided in Tables 5.1 through 5.3.

Some SCOLIB methods exploit parallelism through the use of Dakota's concurrent function evaluations. The nature of the algorithms, however, limits the amount of concurrency that can be exploited. The maximum amount of evaluation concurrency that can be leveraged by the various methods is as follows:

- COBYLA: one
- DIRECT: twice the number of variables
- Evolutionary Algorithms: size of the population
- Pattern Search: size of the search pattern
- Solis-Wets: one

4.4.7.2 SCOLIB method dependent controls

All SCOLIB methods support the `show_misc_options` optional specification which results in a dump of all the allowable method inputs. Note that the information provided by this command refers to optimizer parameters that are internal to SCOLIB, and which may differ from corresponding parameters used by the Dakota interface.

The `misc_options` optional specification provides a means for inputing additional settings supported by the SCOLIB methods but which are not currently mapped through the Dakota input specification. Care must be taken in using this specification; they should only be employed by users familiar with the full range of parameter specifications available directly from SCOLIB and understand any differences that exist between those specifications and the ones available through Dakota.

Each of the SCOLIB methods supports the `solution_target` control, which defines a convergence criterion in which the optimizer will terminate if it finds an objective function value lower than the specified target. Specification detail for method dependent controls for all SCOLIB methods is provided in Table 5.9.

Description	Keyword	Associated Data	Status	Default
Show miscellaneous options	`show_misc_-options`	none	Optional	no dump of specification options
Specify miscellaneous options	`misc_options`	list of strings	Optional	no miscellaneous options specified
Desired solution target	`solution_-target`	real	Optional	`-DBL_MAX`

Table 4.9: Specification detail for SCOLIB method dependent controls

Each SCOLIB method supplements the settings of Table 5.9 with controls which are specific to its particular class of method.

4.4.7.3 COBYLA

The Constrained Optimization BY Linear Approximations (COBYLA) algorithm is an extension to the Nelder-Mead simplex algorithm for handling general linear/nonlinear constraints and is invoked using the `coliny_-cobyla` group specification. The COBYLA algorithm employs linear approximations to the objective and constraint functions, the approximations being formed by linear interpolation at N+1 points in the space of the variables. We regard these interpolation points as vertices of a simplex. The step length parameter controls the size of the simplex and it is reduced automatically from `initial_delta` to `threshold_delta`. One advantage that COBYLA has over many of its competitors is that it treats each constraint individually when calculating a change to the variables, instead of lumping the constraints together into a single penalty function.

COBYLA currently only supports termination based on the `max_function_evaluations` and `solution_target` specifications. The search performed by COBYLA is currently not parallelized.

Table 5.10 summarizes the COBYLA specification.

Description	Keyword	Associated Data	Status	Default
COBYLA method	`coliny_-cobyla`	none	Required group	N/A
Initial offset value	`initial_-delta`	real	Optional	1.0
Threshold for offset values	`threshold_-delta`	real	Optional	0.0001

Table 4.10: Specification detail for the COBYLA method

4.4.7.4 DIRECT

The DIviding RECTangles (DIRECT) optimization algorithm is a derivative free global optimization method that balances local search in promising regions of the design space with global search in unexplored regions. As shown in Figure 5.1, DIRECT adaptively subdivides the space of feasible design points so as to guarantee that iterates are generated in the neighborhood of a global minimum in finitely many iterations.

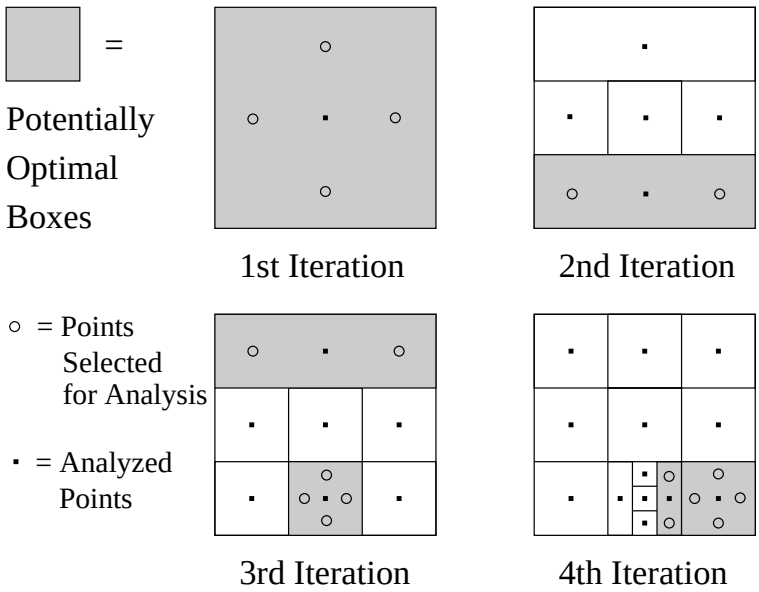

Figure 4.1: Design space partitioning with DIRECT

In practice, DIRECT has proven an effective heuristic for engineering design applications, for which it is able to quickly identify candidate solutions that can be further refined with fast local optimizers.

DIRECT uses the `solution_target`, `constraint_penalty` and `show_misc_options` specifications that are described in SCOLIB method dependent controls. Note, however, that DIRECT uses a fixed penalty value for constraint violations (i.e. it is not dynamically adapted as is done in `coliny_pattern_search`).

The `division` specification determines how DIRECT subdivides each subregion of the search space. If `division` is set to `major_dimension`, then the dimension representing the longest edge of the subregion is subdivided (this is the default). If `division` is set to `all_dimensions`, then all dimensions are simultaneously subdivided.

Each subregion considered by DIRECT has a **size**, which corresponds to the longest diagonal of the subregion. The `global_balance_parameter` controls how much global search is performed by only allowing a subregion to be subdivided if the size of the subregion divided by the size of the largest subregion is at least `global_balance_parameter`. Intuitively, this forces large subregions to be subdivided before the smallest subregions are refined. The `local_balance_parameter` provides a tolerance for estimating whether the smallest subregion can provide a sufficient decrease to be worth subdividing; the default value is a small value that is suitable for most applications.

DIRECT can be terminated with the standard `max_function_evaluations` and `solution_target` specifications. Additionally, the `max_boxsize_limit` specification terminates DIRECT if the size of the

largest subregion falls below this threshold, and the `min_boxsize_limit` specification terminates DIRECT if the size of the smallest subregion falls below this threshold. In practice, this latter specification is likely to be more effective at limiting DIRECT's search.

Table 5.11 summarizes the DIRECT specification.

Description	Keyword	Associated Data	Status	Default
DIRECT method	`coliny_-direct`	none	Required group	N/A
Box subdivision approach	`division`	`major_-dimension` | `all_-dimensions`	Optional group	`major_-dimension`
Global search balancing parameter	`global_-balance_-parameter`	real	Optional	0.0
Local search balancing parameter	`local_-balance_-parameter`	real	Optional	1.e-8
Maximum boxsize limit	`max_-boxsize_-limit`	real	Optional	0.0
Minimum boxsize limit	`min_-boxsize_-limit`	real	Optional	0.0001
Constraint penalty	`constraint_-penalty`	real	Optional	1000.0

Table 4.11: Specification detail for the DIRECT method

4.4.7.5 Evolutionary Algorithms

Dakota currently provides several variants of evolutionary algorithms, invoked through the `coliny_ea` group specification.

The basic steps of an evolutionary algorithm are depicted in Figure 5.2.

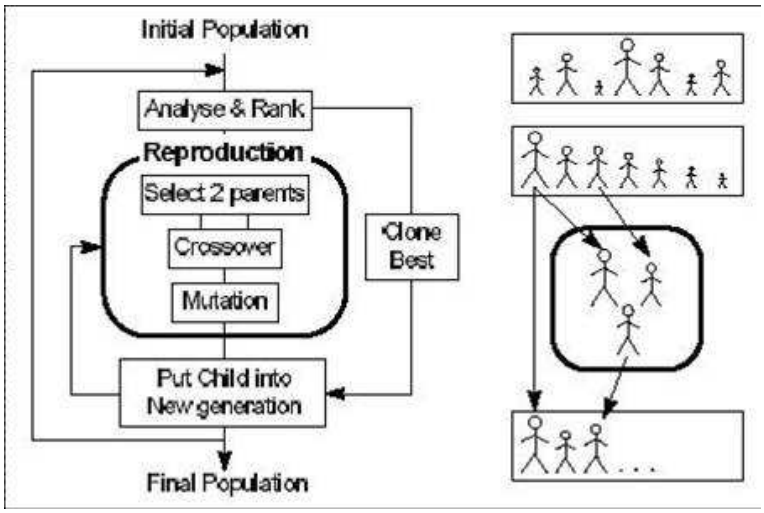

Figure 4.2: Depiction of evolutionary algorithm

They can be enumerated as follows:

1. Select an initial population randomly and perform function evaluations on these individuals
2. Perform selection for parents based on relative fitness
3. Apply crossover and mutation to generate `new_solutions_generated` new individuals from the selected parents
   - Apply crossover with a fixed probability from two selected parents
   - If crossover is applied, apply mutation to the newly generated individual with a fixed probability
   - If crossover is not applied, apply mutation with a fixed probability to a single selected parent
4. Perform function evaluations on the new individuals
5. Perform replacement to determine the new population
6. Return to step 2 and continue the algorithm until convergence criteria are satisfied or iteration limits are exceeded

Table 5.12 provides the specification detail for the controls for seeding the method, initializing a population, and for selecting and replacing population members.

The random `seed` control provides a mechanism for making a stochastic optimization repeatable. That is, the use of the same random seed in identical studies will generate identical results. The `population_size` control specifies how many individuals will comprise the EA's population.

The `initialization_type` defines the type of initialization for the population of the EA. There are three types: `simple_random`, `unique_random`, and `flat_file`. `simple_random` creates initial solutions with random variable values according to a uniform random number distribution. It gives no consideration to any previously generated designs. The number of designs is specified by the `population_size`. `unique_random` is the same as `simple_random`, except that when a new solution is generated, it is checked against

Description	Keyword	Associated Data	Status	Default
EA selection	coliny_ea	none	Required group	N/A
Random seed	seed	integer	Optional	randomly generated seed
Number of population members	population_-size	integer	Optional	50
Initialization type	initialization_-type	simple_-random | unique_-random | flat_file	Required	unique_-random
Fitness type	fitness_type	linear_rank | merit_-function	Optional	linear_rank
Replacement type	replacement_-type	random | chc | elitist	Optional group	elitist = 1
Random replacement type	random	integer	Required	N/A
CHC replacement type	chc	integer	Required	N/A
Elitist replacement type	elitist	integer	Required	N/A
New solutions generated	new_-solutions_-generated	integer	Optional	population_-size - replacement_-size

Table 4.12: Specification detail for the SCOLIB EA method dependent controls: seed, initialization, selection, and replacement

the rest of the solutions. If it duplicates any of them, it is rejected. `flat_file` allows the initial population to be read from a flat file. If `flat_file` is specified, a file name must be given.

The `fitness_type` controls how strongly differences in "fitness" (i.e., the objective function) are weighted in the process of selecting "parents" for crossover:

- the `linear_rank` setting uses a linear scaling of probability of selection based on the rank order of each individual's objective function within the population

- the `merit_function` setting uses a proportional scaling of probability of selection based on the relative value of each individual's objective function within the population

The `replacement_type` controls how current populations and newly generated individuals are combined to create a new population. Each of the `replacement_type` selections accepts an integer value, which is referred to below and in Table 5.12 as the `replacement_size`:

- The `random` setting creates a new population using (a) `replacement_size` randomly selected individuals from the current population, and (b) `population_size` - `replacement_size` individuals randomly selected from among the newly generated individuals (the number of which is optionally specified using `new_solutions_generated`) that are created for each generation (using the selection, crossover, and mutation procedures).

- The `chc` setting creates a new population using (a) the `replacement_size` best individuals from the *combination* of the current population and the newly generated individuals, and (b) `population_size` - `replacement_size` individuals randomly selected from among the remaining individuals in this combined pool. The `chc` setting is the preferred selection for many engineering problems.

- The `elitist` (default) setting creates a new population using (a) the `replacement_size` best individuals from the current population, (b) and `population_size` - `replacement_size` individuals randomly selected from the newly generated individuals. It is possible in this case to lose a good solution from the newly generated individuals if it is not randomly selected for replacement; however, the default `new_solutions_generated` value is set such that the entire set of newly generated individuals will be selected for replacement.

Note that `new_solutions_generated` is not recognized by Dakota as a valid keyword unless `replacement_type` has been specified.

Table 5.13 show the controls in the EA method associated with crossover and mutation.

The `crossover_type` controls what approach is employed for combining parent genetic information to create offspring, and the `crossover_rate` specifies the probability of a crossover operation being performed to generate a new offspring. The SCOLIB EA method supports three forms of crossover, `two_point`, `blend`, and `uniform`, which generate a new individual through combinations of two parent individuals. Two-point crossover divides each parent into three regions, where offspring are created from the combination of the middle region from one parent and the end regions from the other parent. Since the SCOLIB EA does not utilize bit representations of variable values, the crossover points only occur on coordinate boundaries, never within the bits of a particular coordinate. Uniform crossover creates offspring through random combination of coordinates from the two parents. Blend crossover generates a new individual randomly along the multidimensional vector connecting the two parents.

Description	Keyword	Associated Data	Status	Default
Crossover type	crossover_type	two_point | blend | uniform	Optional group	two_point
Crossover rate	crossover_rate	real	Optional	0.8
Mutation type	mutation_type	replace_uniform | offset_normal | offset_cauchy | offset_uniform	Optional group	offset_normal
Mutation scale	mutation_scale	real	Optional	0.1
Mutation range	mutation_range	integer	Optional	1
Mutation dimension ratio	dimension_ratio	real	Optional	1.0
Mutation rate	mutation_rate	real	Optional	1.0
Non-adaptive mutation flag	non_adaptive	none	Optional	Adaptive mutation

Table 4.13: Specification detail for the SCOLIB EA method: crossover and mutation

The `mutation_type` controls what approach is employed in randomly modifying continuous design variables within the EA population. Each of the mutation methods generates coordinate-wise changes to individuals, usually by adding a random variable to a given coordinate value (an "offset" mutation), but also by replacing a given coordinate value with a random variable (a "replace" mutation). Discrete design variables are always mutated using the `offset_uniform` method. The `mutation_rate` controls the probability of mutation being performed on an individual, both for new individuals generated by crossover (if crossover occurs) and for individuals from the existing population. It is the fraction of trial points that are mutated in a given iteration and therefore must be specified to be between 0 and 1. When mutation is performed, all dimensions of each individual are mutated. The `mutation_scale` specifies a scale factor which scales continuous mutation offsets; this is a fraction of the total range of each dimension, so `mutation_scale` is a relative value between 0 and 1. The `mutation_range` is used to control `offset_uniform` mutation used for discrete parameters. The replacement discrete value is the original value plus or minus an integer value up to `mutation_range`. The `offset_normal`, `offset_cauchy`, and `offset_uniform` mutation types are "offset" mutations in that they add a 0-mean random variable with a normal, cauchy, or uniform distribution, respectively, to the existing coordinate value. These offsets are limited in magnitude by `mutation_scale`. The `replace_uniform` mutation type is not limited by `mutation_scale`; rather it generates a replacement value for a coordinate using a uniformly distributed value over the total range for that coordinate.

Note that `mutation_scale` and `mutation_range` are not recognized by Dakota as valid keywords unless `mutation_type` has been specified and the type is an "offset" mutations.

The SCOLIB EA method uses self-adaptive mutation, which modifies the mutation scale dynamically. This mechanism is borrowed from EAs like evolution strategies. The `non_adaptive` flag can be used to deactivate the self-adaptation, which may facilitate a more global search.

Note that `non_adaptive` is not recognized by Dakota as a valid keyword unless `mutation_type` has been specified.

4.4.7.6 Pattern Search

Pattern search techniques are nongradient-based optimization methods which use a set of offsets from the current iterate to locate improved points in the design space. The SCOLIB pattern search technique is invoked using a `coliny_pattern_search` group specification, which includes a variety of specification components.

Traditional pattern search methods search with a fixed pattern of search directions to try to find improvements to the current iterate. The SCOLIB pattern search methods generalize this simple algorithmic strategy to enable control of how the search pattern is adapted, as well as how each search pattern is evaluated. The `stochastic` and `synchronization` specifications denote how the the trial points are evaluated. The `stochastic` specification indicates that the trial points are considered in a random order. For parallel pattern search, `synchronization` dictates whether the evaluations are scheduled using a `blocking` scheduler or a `nonblocking` scheduler (i.e., **Model::synchronize()** or **Model::synchronize_nowait()**, respectively). In the `blocking` case, all points in the pattern are evaluated (in parallel), and if the best of these trial points is an improving point, then it becomes the next iterate. These runs are reproducible, assuming use of the same seed in the `stochastic` case. In the `nonblocking` case, all points in the pattern may not be evaluated, since the first improving point found becomes the next iterate. Since the algorithm steps will be subject to parallel timing variabilities, these runs will not generally be repeatable. The `synchronization` specification has similar connotations for sequential pattern search. If `blocking` is specified, then each sequential iteration terminates after all trial points have been considered, and if `nonblocking` is specified, then each sequential iteration terminates after the first improving trial point is evaluated. In this release, both `blocking` and `nonblocking` specifications result in blocking behavior (except in the case where `exporatory_moves` below is set to `adaptive_pattern`). Nonblocking behavior will be re-enabled after some underlying technical issues have been resolved.

The particular form of the search pattern is controlled by the `pattern_basis` specification. If `pattern_basis` is `coordinate` basis, then the pattern search uses a plus and minus offset in each coordinate direction, for a total of *2n* function evaluations in the pattern. This case is depicted in Figure 5.3 for three coordinate dimensions.

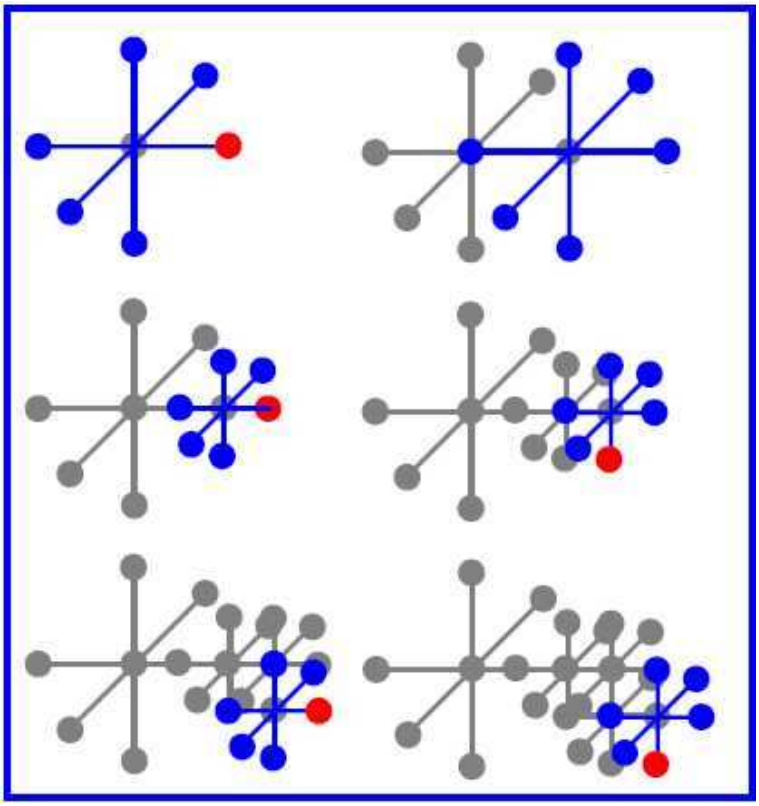

Figure 4.3: Depiction of coordinate pattern search algorithm

If `pattern_basis` is `simplex`, then pattern search uses a minimal positive basis simplex for the parameter space, for a total of *n+1* function evaluations in the pattern. Note that the `simplex` pattern basis can be used for unbounded problems only. The `total_pattern_size` specification can be used to augment the basic `coordinate` and `simplex` patterns with additional function evaluations, and is particularly useful for parallel load balancing. For example, if some function evaluations in the pattern are dropped due to duplication or bound constraint interaction, then the `total_pattern_size` specification instructs the algorithm to generate new offsets to bring the total number of evaluations up to this consistent total.

The `exploratory_moves` specification controls how the search pattern is adapted. (The search pattern can be adapted after an improving trial point is found, or after all trial points in a search pattern have been found to be unimproving points.) The following exploratory moves selections are supported by SCOLIB:

- The `basic_pattern` case is the simple pattern search approach, which uses the same pattern in each iteration.
- The `multi_step` case examines each trial step in the pattern in turn. If a successful step is found, the

pattern search continues examining trial steps about this new point. In this manner, the effects of multiple successful steps are cumulative within a single iteration. This option does not support any parallelism and will result in a serial pattern search.

- The `adaptive_pattern` case invokes a pattern search technique that adaptively rescales the different search directions to maximize the number of redundant function evaluations. See [Hart et al., 2001] for details of this method. In preliminary experiments, this method had more robust performance than the standard `basic_pattern` case in serial tests. This option supports a limited degree of parallelism. After successful iterations (where the step length is not contracted), a parallel search will be performed. After unsuccessful iterations (where the step length is contracted), only a single evaluation is performed.

The `initial_delta` and `threshold_delta` specifications provide the initial offset size and the threshold size at which to terminate the algorithm. For any dimension that has both upper and lower bounds, this step length will be internally rescaled to provide search steps of length `initial_delta` $*$ range $*$ 0.1. This rescaling does not occur for other dimensions, so search steps in those directions have length `initial_delta`. Note that the factor of 0.1 in the rescaling could result in an undesirably small initial step. This can be offset by providing a large `initial_delta`.

In general, pattern search methods can expand and contract their step lengths. SCOLIB pattern search methods contract the step length by the value `contraction_factor`, and they expand the step length by the value (1/contraction_factor). The `expand_after_success` control specifies how many successful objective function improvements must occur with a specific step length prior to expansion of the step length, whereas the `no_expansion` flag instructs the algorithm to forgo pattern expansion altogether.

Finally, constraint infeasibility can be managed in a somewhat more sophisticated manner than the simple weighted penalty function. If the `constant_penalty` specification is used, then the simple weighted penalty scheme described above is used. Otherwise, the constraint penalty is adapted to the value `constraint_penalty/L`, where L is the the smallest step length used so far.

Table 5.14 and Table 5.15 provide the specification detail for the SCOLIB pattern search method and its method dependent controls.

4.4.7.7 Solis-Wets

Dakota's implementation of SCOLIB also contains the Solis-Wets algorithm. The Solis-Wets method is a simple greedy local search heuristic for continuous parameter spaces. Solis-Wets generates trial points using a multivariate normal distribution, and unsuccessful trial points are reflected about the current point to find a descent direction. This algorithm is inherently serial and will not utilize any parallelism. Table 5.16 provides the specification detail for this method and its method dependent controls.

These specifications have the same meaning as corresponding specifications for `coliny_pattern_search`. In particular, `coliny_solis_wets` supports dynamic rescaling of the step length, and dynamic rescaling of the constraint penalty. The only new specification is `contract_after_failure`, which specifies the number of unsuccessful cycles which must occur with a specific delta prior to contraction of the delta.

4.4.8 NCSU Methods

North Carolina State University (NCSU) has an implementation of the DIRECT algorithm (DIviding RECTangles algorithm that is outlined in the SCOLIB method section above). This version is documented in [Gablonsky,

Description	Keyword	Associated Data	Status	Default
SCOLIB pattern search method	coliny_-pattern_-search	none	Required group	N/A
Stochastic pattern search	stochastic	none	Optional group	N/A
Random seed for stochastic pattern search	seed	integer	Optional	randomly generated seed
Initial offset value	initial_-delta	real	Optional	0.1∗range
Threshold for offset values	threshold_-delta	real	Optional	0.00001
Constraint penalty	constraint_-penalty	real	Optional	1.0
Control of dynamic penalty	constant_-penalty	none	Optional	algorithm dynamically adapts the constraint penalty

Table 4.14: Specification detail for the SCOLIB pattern search method: randomization, delta, and constraint controls

Description	Keyword	Associated Data	Status	Default
Pattern basis selection	pattern_-basis	coordinate | simplex	Optional	coordinate
Total number of points in pattern	total_-pattern_size	integer	Optional	no augmentation of basic pattern
No expansion flag	no_expansion	none	Optional	algorithm may expand pattern size
Number of consecutive improvements before expansion	expand_-after_-success	integer	Optional	5
Pattern contraction factor	contraction_-factor	real	Optional	0.5
Evaluation synchronization	synchronization	blocking | nonblocking	Optional	nonblocking
Exploratory moves selection	exploratory_-moves	basic_-pattern | multi_step | adaptive_-pattern	Optional	basic_-pattern

Table 4.15: Specification detail for the SCOLIB pattern search method: pattern controls

Description	Keyword	Associated Data	Status	Default
SCOLIB Solis-Wets method	coliny_solis_wets	none	Required group	N/A
Random seed for stochastic pattern search	seed	integer	Optional	randomly generated seed
Initial offset value	initial_delta	real	Optional	0.1∗range
Threshold for offset values	threshold_delta	real	Optional	0.000001
No expansion flag	no_expansion	none	Optional	algorithm may expand pattern size
Number of consecutive improvements before expansion	expand_after_success	integer	Optional	5
Number of consecutive failures before contraction	contract_after_failure	integer	Optional	4∗number of variables
Pattern contraction factor	contraction_factor	real	Optional	0.5
Constraint penalty	constraint_penalty	real	Optional	1.0
Control of dynamic penalty	constant_penalty	none	Optional	algorithm dynamically adapts the constraint penalty

Table 4.16: Specification detail for the SCOLIB Solis-Wets method

2001.] We have found that the NCSU DIRECT implementation works better and is more robust for some problems than `coliny_direct`. Currently, we maintain both versions of DIRECT in Dakota; in the future, we may deprecate one. The NCSU DIRECT method is selected with `ncsu_direct`. We have tried to maintain consistency between the keywords in SCOLIB and NCSU implementation of DIRECT, but the algorithms have different parameters, so the keywords sometimes have slightly different meaning.

4.4.8.1 NCSU method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of iterations and the number of function evaluations that can be performed during an NCSU DIRECT optimization. This methods will always strictly respect the number of iterations, but may slightly exceed the number of function evaluations, as it will always explore all sub-rectangles at the current level.

4.4.8.2 NCSU method dependent controls

There are four specification controls affecting NCSU DIRECT: `solution_target`, `convergence_tolerance`, `min_boxsize_limit`, and `volume_boxsize_limit`. The solution target specifies a goal toward which the optimizer should track. When `solution_target` is specified, `convergence_tolerance` specifies a percent error on the optimization. This is used for test problems, when the true global minimum is known (call it `solution_target` := fglobal). Then, the optimization terminates when 100(f_min-fglobal)/max(1,abs(fglobal) < convergence_tolerance. The default for fglobal is -1.0e100 and the default for convergence tolerance is as given above.

`min_boxsize_limit` is a setting that terminates the optimization when the measure of a hyperrectangle S with f(c(S)) = fmin is less than min_boxsize_limit. `volume_boxsize_limit` is a setting that terminates the optimization when the volume of a hyperrectangle S with f(c(S)) = fmin is less than volume_boxsize_limit percent of the original hyperrectangle. Basically, volume_boxsize_limit stops the optimization when the volume of the particular rectangle which has fmin is less than a certain percentage of the whole volume. `min_boxsize_limit` uses an arbitrary measure to stop the optimization. The keywords for NCSU DIRECT are described in Table 5.17 below.

Description	Keyword	Associated Data	Status	Default
Solution Target	`solution_target`	real	Optional	0
Min boxsize limit	`min_boxsize_limit`	real in [0,1]	Optional	1.0e-4
Volume boxsize limit	`volume_boxsize_limit`	real in [0,1]	Optional	1.0e-6

Table 4.17: Specification detail for the NCSU DIRECT method

4.4.9 JEGA Methods

The JEGA library [Eddy and Lewis, 2001] contains two global optimization methods. The first is a Multi-objective Genetic Algorithm (MOGA) which performs Pareto optimization. The second is a Single-objective Genetic Algorithm (SOGA) which performs optimization on a single objective function. Both methods support general

constraints and a mixture of real and discrete variables. The JEGA library was written by John Eddy, currently a member of the technical staff in the System Readiness and Sustainment Technologies department at Sandia National Laboratories in Albuquerque. These algorithms are accessed as `moga` and `soga` within Dakota. Dakota provides access to the JEGA library through the **JEGAOptimizer** class.

4.4.9.1 JEGA method independent controls

JEGA utilizes the `max_iterations` and `max_function_evaluations` method independent controls to provide integer limits for the maximum number of generations and function evaluations, respectively. Note that currently, the Dakota default for `max_iterations` is 100 and for `max_function_evaluations` is 1000. These are the default settings that will be used to "stop" the JEGA algorithms, unless some specific convergence criteria are set (see Tables 5.20 and 5.21 below).

Beginning with v2.0, JEGA also utilizes the `output` method independent control to vary the amount of information presented to the user during execution.

4.4.9.2 JEGA method dependent controls

The JEGA library currently provides two types of genetic algorithms (GAs): a multi-objective genetic algorithm (`moga`), and a single- objective genetic algorithm (`soga`). Both of these GAs can take real-valued inputs, integer-valued inputs, or a mixture of real and integer-valued inputs. "Real-valued" and "integer-valued" refer to the use of continuous or discrete variable domains, respectively (the response data are real-valued in all cases).

The basic steps of the genetic algorithm are as follows:

1. Initialize the population (by randomly generating population members with or without duplicates allowed, or by flat-file initialization)
2. Evaluate the initial population members (calculate the values of the objective function(s) and constraints for each population member)
3. Perform crossover (several crossover types are available)
4. Perform mutation (several mutation types are available)
5. Evaluate the new population members.
6. Assess the fitness of each member in the population. There are a number of ways to evaluate the fitness of the members of the population. Choice of fitness assessor operators is strongly related to the type of replacement algorithm being used and can have a profound effect on the solutions selected for the next generation. For example, if using `MOGA`, the available assessors are the `layer_rank` and `domination_count` fitness assessors. If using either of these, it is strongly recommended that you use the `replacement_type` called the `below_limit` selector as well (although the roulette wheel selectors can also be used). The functionality of the domination_count selector of JEGA v1.0 can now be achieved using the `domination_count` fitness assessor and `below_limit` replacement selector together. If using `SOGA`, there are a number of possible combinations of fitness assessors and selectors.
7. Replace the population with members selected to continue in the next generation. The pool of potential members is the current population and the current set of offspring. The `replacement_type` of `roulette_wheel` or `unique_roulette_wheel` may be used either with MOGA or SOGA problems however they are not recommended for use with MOGA. Given that the only two fitness assessors for

MOGA are the `layer_rank` and `domination_count`, the recommended selector is the `below_limit` selector. The `below_limit` replacement will only keep designs that are dominated by fewer than a limiting number of other designs. The `replacement_type` of `favor_feasible` is specific to a SOGA. This replacement operator will always prefer a more feasible design to a less feasible one. Beyond that, it favors solutions based on an assigned fitness value which must have been installed by the weighted sum only fitness assessor (see the discussion below).

8. Apply niche pressure to the population. This step is specific to the MOGA and is new as of JEGA v2.0. Technically, the step is carried out during runs of the SOGA but only the `null_niching` operator is available for use with SOGA. In MOGA, the `radial` or `distance` operators can be used. The purpose of niching is to encourage differentiation along the Pareto frontier and thus a more even and uniform sampling. The radial nicher takes information input from the user to compute a minimum allowable distance between designs in the performance space and acts as a secondary selection operator whereby it enforces this minimum distance. The distance nicher requires that solutions must be separated from other solutions by a minimum distance in each dimension (vs. Euclidean distance for the radial niching). After niching is complete, all designs in the population will be at least the minimum distance from one another in all directions.

9. Test for convergence. There are two aspects to convergence that must be considered. The first is stopping criteria. A stopping criteria dictates some sort of limit on the algorithm that is independent of its performance. Examples of stopping criteria available for use with JEGA are the `max_iterations` and `max_function_evaluations` inputs. All JEGA convergers respect these stopping criteria in addition to anything else that they do.

   The second aspect to convergence involves repeated assessment of the algorithms progress in solving the problem. In JEGA v1.0, the SOGA fitness tracker convergers (`best_fitness_tracker` and `average_fitness_tracker`) performed this function by asserting that the fitness values (either best or average) of the population continue to improve. There was no such operator for the MOGA. As of JEGA v2.0, the same fitness tracker convergers exist for use with SOGA and there is now a converger available for use with the MOGA. The MOGA converger (`metric_tracker`) operates by tracking various changes in the non-dominated frontier from generation to generation. When the changes occurring over a user specified number of generations fall below a user specified threshold, the algorithm stops.

10. Perform post processing. This step is new as of JEGA v2.1. The purpose of this operation is to perform any needed data manipulations on the final solution deemed necessary. Currently the `distance_postprocessor` is the only one other than the `null_postprocessor`. The `distance_postprocessor` is specifically for use with the MOGA and reduces the final solution set size such that a minimum distance in each direction exists between any two designs.

There are many controls which can be used for both MOGA and SOGA methods. These include among others the random seed, initialization types, crossover and mutation types, and some replacement types. These are described in Tables 5.18 and 5.19 below.

The `seed` control defines the starting seed for the random number generator. The algorithm uses random numbers heavily but a specification of a random seed will cause the algorithm to run identically from one trial to the next so long as all other input specifications remain the same. New as of JEGA v2.0 is the introduction of the `log_file` specification. JEGA now uses a logging library to output messages and status to the user. JEGA can be configured at build time to log to both the console window and a text file, one or the other, or neither. The `log_file` input is a string name of a file into which to log. If the build was configured without file logging in JEGA, this input is ignored. If file logging is enabled and no `log_file` is specified, the default file name of JEGAGlobal.log is used. Also new to JEGA v2.0 is the introduction of the `print_each_pop` specification. It serves as a flag and

if supplied, the population at each generation will be printed to a file named "population<GEN#>.dat" where <GEN#> is the number of the current generation.

The `initialization_type` defines the type of initialization for the GA. There are three types: `simple_random`, `unique_random`, and `flat_file`. `simple_random` creates initial solutions with random variable values according to a uniform random number distribution. It gives no consideration to any previously generated designs. The number of designs is specified by the `population_size`. `unique_random` is the same as `simple_random`, except that when a new solution is generated, it is checked against the rest of the solutions. If it duplicates any of them, it is rejected. `flat_file` allows the initial population to be read from a flat file. If `flat_file` is specified, a file name must be given. Variables can be delimited in the flat file in any way you see fit with a few exceptions. The delimiter must be the same on any given line of input with the exception of leading and trailing whitespace. So a line could look like: 1.1, 2.2 ,3.3 for example but could not look like: 1.1, 2.2 3.3. The delimiter can vary from line to line within the file which can be useful if data from multiple sources is pasted into the same input file. The delimiter can be any string that does not contain any of the characters .+-dDeE or any of the digits 0-9. The input will be read until the end of the file. The algorithm will discard any configurations for which it was unable to retrieve at least the number of design variables. The objective and constraint entries are not required but if ALL are present, they will be recorded and the design will be tagged as evaluated so that evaluators may choose not to re-evaluate them. Setting the size for this initializer has the effect of requiring a minimum number of designs to create. If this minimum number has not been created once the files are all read, the rest are created using the `unique_random` initializer and then the `simple_random` initializer if necessary.

Note that the `population_size` only sets the size of the initial population. The population size may vary in the JEGA methods according to the type of operators chosen for a particular optimization run.

There are many crossover types available. `multi_point_binary` crossover requires an integer number, N, of crossover points. This crossover type performs a bit switching crossover at N crossover points in the binary encoded genome of two designs. Thus, crossover may occur at any point along a solution chromosome (in the middle of a gene representing a design variable, for example). `multi_point_parameterized_binary` crossover is similar in that it performs a bit switching crossover routine at N crossover points. However, this crossover type performs crossover on each design variable individually. So the individual chromosomes are crossed at N locations. `multi_point_real` crossover performs a variable switching crossover routing at N crossover points in the real real valued genome of two designs. In this scheme, crossover only occurs between design variables (chromosomes). Note that the standard solution chromosome representation in the JEGA algorithm is real encoded and can handle integer or real design variables. For any crossover types that use a binary representation, real variables are converted to long integers by multiplying the real number by 10^6 and then truncating. Note that this assumes a precision of only six decimal places. Discrete variables are represented as integers (indices within a list of possible values) within the algorithm and thus require no special treatment by the binary operators.

The final crossover type is `shuffle_random`. This crossover type performs crossover by choosing design variables at random from a specified number of parents enough times that the requested number of children are produced. For example, consider the case of 3 parents producing 2 children. This operator would go through and for each design variable, select one of the parents as the donor for the child. So it creates a random shuffle of the parent design variable values. The relative numbers of children and parents are controllable to allow for as much mixing as desired. The more parents involved, the less likely that the children will wind up exact duplicates of the parents.

All crossover types take a `crossover_rate`. The crossover rate is used to calculate the number of crossover operations that take place. The number of crossovers is equal to the rate $*$ population_size.

There are five mutation types allowed. `replace_uniform` introduces random variation by first randomly choosing a design variable of a randomly selected design and reassigning it to a random valid value for that variable. No consideration of the current value is given when determining the new value. All mutation types have

a `mutation_rate`. The number of mutations for the replace_uniform mutator is the product of the mutation_-rate and the population_size.

The `bit_random` mutator introduces random variation by first converting a randomly chosen variable of a randomly chosen design into a binary string. It then flips a randomly chosen bit in the string from a 1 to a 0 or visa versa. In this mutation scheme, the resulting value has more probability of being similar to the original value. The number of mutations performed is the product of the mutation_rate, the number of design variables, and the population size.

The offset mutators all act by adding an "offset" random amount to a variable value. The random amount has a mean of zero in all cases. The `offset_normal` mutator introduces random variation by adding a Gaussian random amount to a variable value. The random amount has a standard deviation dependent on the `mutation_-scale`. The `mutation_scale` is a fraction in the range [0, 1] and is meant to help control the amount of variation that takes place when a variable is mutated. `mutation_scale` is multiplied by the range of the variable being mutated to serve as standard deviation. `offset_cauchy` is similar to `offset_normal`, except that a Cauchy random variable is added to the variable being mutated. The `mutation_scale` also defines the standard deviation for this mutator. Finally, `offset_uniform` adds a uniform random amount to the variable value. For the `offset_uniform` mutator, the `mutation_scale` is interpreted as a fraction of the total range of the variable. The range of possible deviation amounts is +/- 1/2 $*$ (`mutation_scale` $*$ variable range). The number of mutations for all offset mutators is defined as the product of `mutation_rate` and `population_-size`.

As of JEGA v2.0, all replacement types are common to both MOGA and SOGA. They include the `roulette_-wheel`, `unique_roulette_wheel`, `elitist`, and `below_limit` selectors. In roulette_wheel replacement, each design is conceptually allotted a portion of a wheel proportional to its fitness relative to the fitnesses of the other Designs. Then, portions of the wheel are chosen at random and the design occupying those portions are duplicated into the next population. Those Designs allotted larger portions of the wheel are more likely to be selected (potentially many times). `unique_roulette_wheel` replacement is the same as `roulette_-wheel` replacement, with the exception that a design may only be selected once. The `below_limit` selector attempts to keep all designs for which the negated fitness is below a certain limit. The values are negated to keep with the convention that higher fitness is better. The inputs to the `below_limit` selector are the limit as a real value, and a `shrinkage_percentage` as a real value. The `shrinkage_percentage` defines the minimum amount of selections that will take place if enough designs are available. It is interpreted as a percentage of the population size that must go on to the subsequent generation. To enforce this, `below_limit` makes all the selections it would make anyway and if that is not enough, it takes the remaining that it needs from the best of what is left (effectively raising its limit as far as it must to get the minimum number of selections). It continues until it has made enough selections. The `shrinkage_percentage` is designed to prevent extreme decreases in the population size at any given generation, and thus prevent a big loss of genetic diversity in a very short time. Without a shrinkage limit, a small group of "super" designs may appear and quickly cull the population down to a size on the order of the limiting value. In this case, all the diversity of the population is lost and it is expensive to re-diversify and spread the population. The `elitist` selector simply chooses the required number of designs taking the most fit. For example, if 100 selections are requested, then the top 100 designs as ranked by fitness will be selected and the remaining will be discarded.

4.4.9.3 Multi-objective Evolutionary Algorithms

The specification for controls specific to Multi-objective Evolutionary algorithms are described here. These controls will be appropriate to use if the user has specified `moga` as the method.

The initialization, crossover, and mutation controls were all described in the preceding section. There are no MOGA specific aspects to these controls. The `fitness_type` for a MOGA may be `domination_count` or

Description	Keyword	Associated Data	Status	Default
GA Method	moga | soga	none	Required group	N/A
Random seed	seed	integer	Optional	Time based seed
Log file	log_file	string	Optional	JEGAGlobal.log
Number of population members	population_-size	integer	Optional	50
Population output	print_each_-pop	none	Optional	No printing
Output verbosity	output	silent | quiet | verbose | debug	Optional	normal
Initialization type	initialization_-type	simple_-random | unique_-random | flat_file	Optimal	unique_random
Mutation type	mutation_-type	replace_-uniform | bit_random | offset_-cauchy | offset_-uniform | offset_-normal	Optional group	replace_uniform
Mutation scale	mutation_-scale	real	Optional	0.15
Mutation rate	mutation_-rate	real	Optional	0.08
Below limit selection	below_limit	real	Optional	6
Shrinkage percentage in below limit selection	shrinkage_-percentage	real	Optional	0.9

Table 4.18: Specification detail for JEGA method dependent controls: seed, output, initialization, mutation, and replacement

Description	Keyword	Associated Data	Status	Default
Crossover type	crossover_type	multi_point_binary | multi_point_parameterized_binary | multi_point_real | shuffle_random	Optional group	shuffle_random
Multi point binary crossover	multi_point_binary	integer	Required	N/A
Multi point parameterized binary crossover	multi_point_parameterized_binary	integer	Required	N/A
Multi point real crossover	multi_point_real	integer	Required	N/A
Random shuffle crossover	shuffle_random	num_parents, num_offspring	Required	N/A
Number of parents in random shuffle crossover	num_parents	integer	optional	2
Number of offspring in random shuffle crossover	num_offspring	integer	optional	2
Crossover rate	crossover_rate	real	optional (applies to all crossover types)	0.8

Table 4.19: Specification detail for JEGA method dependent controls: crossover

layer_rank. Both have been specifically designed to avoid problems with aggregating and scaling objective function values and transforming them into a single objective. Instead, the domination_count fitness assessor works by ordering population members by the negative of the number of designs that dominate them. The values are negated in keeping with the convention that higher fitness is better. The layer_rank fitness assessor works by assigning all non-dominated designs a layer of 0, then from what remains, assigning all the non-dominated a layer of 1, and so on until all designs have been assigned a layer. Again, the values are negated for the higher-is-better fitness convention. Use of the below_limit selector with the domination_count fitness assessor has the effect of keeping all designs that are dominated by fewer then a limiting number of other designs subject to the shrinkage limit. Using it with the layer_rank fitness assessor has the effect of keeping all those designs whose layer is below a certain threshold again subject to the shrinkage limit.

New as of JEGA v2.0 is the introduction of niche pressure operators. These operators are meant primarily for use with the moga. The job of a niche pressure operator is to encourage diversity along the Pareto frontier as the algorithm runs. This is typically accomplished by discouraging clustering of design points in the performance space. In JEGA, the application of niche pressure occurs as a secondary selection operation. The nicher is given a chance to perform a pre-selection operation prior to the operation of the selection (replacement) operator, and is then called to perform niching on the set of designs that were selected by the selection operator.

Currently, the only niche pressure operators available are the radial nicher, the distance nicher, and the max_designs nicher. The radial niche pressure applicator works by enforcing a minimum Euclidean distance between designs in the performance space at each generation. The algorithm proceeds by starting at the (or one of the) extreme designs along objective dimension 0 and marching through the population removing all designs that are too close to the current design. One exception to the rule is that the algorithm will never remove an extreme design which is defined as a design that is maximal or minimal in all but 1 objective dimension (for a classical 2 objective problem, the extreme designs are those at the tips of the non-dominated frontier). The distance nicher enforces a minimimum distance in each dimension.

The designs that are removed by the nicher are not discarded. They are buffered and re-inserted into the population during the next pre-selection operation. This way, the selector is still the only operator that discards designs and the algorithm will not waste time "re-filling" gaps created by the nicher.

The radial nicher requires as input a vector of fractions with length equal to the number of objectives. The elements of the vector are interpreted as percentages of the non-dominated range for each objective defining a minimum distance to all other designs. All values should be in the range (0, 1). The minimum allowable distance between any two designs in the performance space is the Euclidian (simple square-root-sum-of-squares calculation) distance defined by these percentages. The distance nicher has a similar input vector requirement, only the distance is the minimum distance in each dimension.

The max_designs niche pressure applicator is designed to choose a limited number of solutions to remain in the population. That number is specified by num_designs. It does so in order to balance the tendency for populations to grow very large and thus consuming too many computer resources. It operates by ranking designs according to their fitness standing and a computed count of how many other designs are too close to them. Too close is a function of the supplied niche_vector, which specifies the minimum distance between any two points in the performance space along each dimension individually. Once the designs are all ranked, the top c\ num_-designs designs are kept in the population and the remaining ones are bufferred or discarded. Note that like other niching operators, this one will not discard an extreme design.

Also new as of JEGA v2.0 is the introduction of the MOGA specific metric_tracker converger. This converger is conceptually similar to the best and average fitness tracker convergers in that it tracks the progress of the population over a certain number of generations and stops when the progress falls below a certain threshold. The implementation is quite different however. The metric_tracker converger tracks 3 metrics specific to the non-dominated frontier from generation to generation. All 3 of these metrics are computed as percent changes between the generations. In order to compute these metrics, the converger stores a duplicate of the non-dominated

frontier at each generation for comparison to the non-dominated frontier of the next generation.

The first metric is one that indicates how the expanse of the frontier is changing. The expanse along a given objective is defined by the range of values existing within the non-dominated set. The expansion metric is computed by tracking the extremes of the non-dominated frontier from one generation to the next. Any movement of the extreme values is noticed and the maximum percentage movement is computed as:

```
Em = max over j of abs((range(j, i) - range(j, i-1)) / range(j, i-1))  j=1,nof
```

where Em is the max expansion metric, j is the objective function index, i is the current generation number, and nof is the total number of objectives. The range is the difference between the largest value along an objective and the smallest when considering only non-dominated designs.

The second metric monitors changes in the density of the non-dominated set. The density metric is computed as the number of non-dominated points divided by the hypervolume of the non-dominated region of space. Therefore, changes in the density can be caused by changes in the number of non-dominated points or by changes in size of the non-dominated space or both. The size of the non-dominated space is computed as:

```
Vps(i) = product over j of range(j, i)   j=1,nof
```

where Vps(i) is the hypervolume of the non-dominated space at generation i and all other terms have the same meanings as above.

The density of the a given non-dominated space is then:

```
Dps(i) = Pct(i) / Vps(i)
```

where Pct(i) is the number of points on the non-dominated frontier at generation i.

The percentage increase in density of the frontier is then calculated as

```
Cd = abs((Dps(i) - Dps(i-1)) / Dps(i-1))
```

where Cd is the change in density metric.

The final metric is one that monitors the "goodness" of the non-dominated frontier. This metric is computed by considering each design in the previous population and determining if it is dominated by any designs in the current population. All that are determined to be dominated are counted. The metric is the ratio of the number that are dominated to the total number that exist in the previous population.

As mentioned above, each of these metrics is a percentage. The tracker records the largest of these three at each generation. Once the recorded percentage is below the supplied percent change for the supplied number of generations consecutively, the algorithm is converged.

The specification for convergence in a moga can either be `metric_tracker` or can be omitted all together. If omitted, no convergence algorithm will be used and the algorithm will rely on stopping criteria only. If `metric_tracker` is specified, then a `percent_change` and `num_generations` must be supplied as with the other metric tracker convergers (average and best fitness trackers). The `percent_change` is the threshold beneath which convergence is attained whereby it is compared to the metric value computed as described above. The `num_generations` is the number of generations over which the metric value should be tracked. Convergence will be attained if the recorded metric is below `percent_change` for `num_generations` consecutive generations.

The MOGA specific controls are described in Table 5.20 below. Note that MOGA and SOGA create additional output files during execution. "finaldata.dat" is a file that holds the final set of Pareto optimal solutions after any post-processing is complete. "discards.dat" holds solutions that were discarded from the population during the

course of evolution. It can often be useful to plot objective function values from these files to visually see the Pareto front and ensure that finaldata.dat solutions dominate discards.dat solutions. The solutions are written to these output files in the format "Input1...InputN..Output1...OutputM". If MOGA is used in a hybrid optimization strategy (which requires one optimal solution from each individual optimization method to be passed to the subsequent optimization method as its starting point), the solution in the Pareto set closest to the "utopia" point is given as the best solution. This solution is also reported in the Dakota output. This "best" solution in the Pareto set has minimum distance from the utopia point. The utopia point is defined as the point of extreme (best) values for each objective function. For example, if the Pareto front is bounded by (1,100) and (90,2), then (1,2) is the utopia point. There will be a point in the Pareto set that has minimum L2-norm distance to this point, for example (10,10) may be such a point. In SOGA, the solution that minimizes the single objective function is returned as the best solution. If moga is used in a strategy which may require passing multiple solutions to the next level (such as the `surrogate_based_global` method or `hybrid` strategy), the `orthogonal_distance` postprocessor type may be used to specify the distances between each solution value to winnow down the solutions in the full Pareto front to a subset which will be passed to the next iteration.

4.4.9.4 Single-objective Evolutionary Algorithms

The specification for controls specific to Single-objective Evolutionary algorithms are described here. These controls will be appropriate to use if the user has specified `soga` as the method.

The initialization, crossover, and mutation controls were all described above. There are no SOGA specific aspects to these controls. The `replacement_type` for a SOGA may be `roulette_wheel`, `unique_roulette_wheel`, `elitist`, or `favor_feasible`. The `favor_feasible` replacement type first considers feasibility as a selection criteria. If that does not produce a "winner" then it moves on to considering fitness value. Because of this, any fitness assessor used with the `favor_feasible` selector must only account objectives in the creation of fitness. Therefore, there is such a fitness assessor and it's use is enforced when the \ favor_feasible selector is chosen. In that case, and if the output level is set high enough, a message will be presented indicating that the `weighted_sum_only` fitness assessor will be used. As of JEGA v2.0 and beyond, the fitness assessment operator must be specified with SOGA although the `merit_function` is currently the only one (note that the `weighted_sum_only` assessor exists but cannot be selected). The roulette wheel selectors no longer assume a fitness function. The `merit_function` fitness assessor uses an exterior penalty function formulation to penalize infeasible designs. The specification allows the input of a `constraint_penalty` which is the multiplier to use on the constraint violations.

The SOGA controls allow two additional convergence types. The `convergence_type` called `average_fitness_tracker` keeps track of the average fitness in a population. If this average fitness does not change more than `percent_change` over some number of generations, `num_generations`, then the solution is reported as converged and the algorithm terminates. The `best_fitness_tracker` works in a similar manner, only it tracks the best fitness in the population. Convergence occurs after `num_generations` has passed and there has been less than `percent_change` in the best fitness value. The percent change can be as low as 0% in which case there must be no change at all over the number of generations. Both also respect the stopping criteria.

The SOGA specific controls are described in Table 5.21 below.

4.5 Least Squares Methods

Dakota's least squares branch currently contains three methods for solving nonlinear least squares problems: NL2SOL, a trust-region method that adaptively chooses between two Hessian approximations (Gauss-Newton and Gauss-Newton plus a quasi-Newton approximation to the rest of the Hessian), NLSSOL, a sequential quadratic

Description	Keyword	Associated Data	Status	Default
Fitness type	fitness_type	layer_rank | domination_count	Required group	domination_count
Niche pressure type	niching_type	radial | distance | max_designs	Optional group	No niche pressure
Niching distance	radial | distance | max_designs	list of real	Optional	0.01 for all objectives
Number designs to keep for max_designs nicher	num_designs	integer	Optional	100
Replacement type	replacement_type	below_limit | roulette_wheel | unique_roulette_wheel | elitist	Required group	below_limit
Convergence type	metric_tracker	none	Optional group	metric_tracker
Percent change limit for metric_tracker converger	percent_change	real	Optional	0.1
Number generations for metric_tracker converger	num_generations	integer	Optional	10
Post_processor type	postprocessor_type	orthogonal_distance	Optional	No post-processing of solutions
Post_processor distance	orthogonal_distance	list of real	Optional	0.01 for all objectives

Table 4.20: Specification detail for MOGA method controls

Description	Keyword	Associated Data	Status	Default
Fitness type	fitness_type	merit_function	Optional group	merit_function
Constraint penalty in merit function	constraint_penalty	real	Optional	1.0
Replacement type	replacement_type	favor_feasible | unique_roulette_wheel | roulette_wheel | elitist	Required group	elitist
Convergence type	convergence_type	best_fitness_tracker | average_fitness_tracker	Optional	average_fitness_tracker
Number of generations (for convergence test)	num_generations	integer	Optional	10
Percent change in fitness	percent_change	real	Optional	0.1

Table 4.21: Specification detail for SOGA method controls

programming (SQP) approach that is from the same algorithm family as NPSOL, and Gauss-Newton, which supplies the Gauss-Newton Hessian approximation to the full-Newton optimizers from OPT++.

The important difference of these algorithms from general-purpose optimization methods is that the response set is defined by calibration terms (e.g. separate terms for each residual), rather than an objective function. Thus, a finer granularity of data is used by least squares solvers as compared to that used by optimizers. This allows the exploitation of the special structure provided by a sum of squares objective function. Refer to Calibration terms and constraint functions (least squares data set) for additional information on the least squares response data set.

4.5.1 NL2SOL Method

NL2SOL is available as `nl2sol` and addresses unconstrained and bound-constrained problems. It uses a trust-region method (and thus can be viewed as a generalization of the Levenberg-Marquardt algorithm) and adaptively chooses between two Hessian approximations, the Gauss-Newton approximation alone and the Gauss-Newton approximation plus a quasi-Newton approximation to the rest of the Hessian. Even on small-residual problems, the latter Hessian approximation can be useful when the starting guess is far from the solution. On problems that are not over-parameterized (i.e., that do not involve more optimization variables than the data support), NL2SOL usually exhibits fast convergence.

NL2SOL has a variety of internal controls as described in AT&T Bell Labs CS TR 153 (`http://cm.bell-labs.com/cm/cs/cstr/153.ps.gz`). A number of existing Dakota controls (method independent controls and responses controls) are mapped into these NL2SOL internal controls. In particular, Dakota's `convergence_tolerance`, `max_iterations`, `max_function_evaluations`, and `fd_gradient_step_size` are mapped directly into NL2SOL's `rfctol`, `mxiter`, `mxfcal`, and `dltfdj` controls, respectively. In addition, Dakota's `fd_hessian_step_size` is mapped into both `delta0` and `dltfdc`, and Dakota's `output` verbosity is mapped into NL2SOL's `auxprt` and `outlev` (for `normal/verbose/debug output, NL2SOL` prints initial guess, final solution, solution statistics, nondefault values, and changes to the active bound constraint set on every iteration; for `quiet output`, NL2SOL prints only the initial guess and final solution; and for `silent output`, NL2SOL output is suppressed).

Several NL2SOL convergence tolerances are adjusted in response to `function_precision`, which gives the relative precision to which responses are computed. These tolerances may also be specified explicitly: `convergence_tolerance` (NL2SOL's `rfctol`, as mentioned previously) is the relative-function convergence tolerance (on the accuracy desired in the sum-of-squares function); `x_conv_tol` (NL2SOL's `xctol`) is the X-convergence tolerance (scaled relative accuracy of the solution variables); `absolute_conv_tol` (NL2SOL's `afctol`) is the absolute function convergence tolerance (stop when half the sum of squares is less than `absolute_conv_tol`, which is mainly of interest on zero-residual test problems); `singular_conv_tol` (NL2SOL's `sctol`) is the singular convergence tolerance, which works in conjunction with `singular_radius` (NL2SOL's `lmaxs`) to test for underdetermined least-squares problems (stop when the relative reduction yet possible in the sum of squares appears less then `singular_conv_tol` for steps of scaled length at most `singular_radius`); `false_conv_tol` (NL2SOL's `xftol`) is the false-convergence tolerance (stop with a suspicion of discontinuity when a more favorable stopping test is not satisfied and a step of scaled length at most `false_conv_tol` is not accepted). Finally, the `initial_trust_radius` specification (NL2SOL's `lmax0`) specifies the initial trust region radius for the algorithm.

The internal NL2SOL defaults can be obtained for many of these controls by specifying the value -1. For both the `singular_radius` and the `initial_trust_radius`, this results in the internal use of steps of length 1. For other controls, the internal defaults are often functions of machine epsilon (as limited by `function_precision`). Refer to CS TR 153 for additional details on these formulations.

Whether and how NL2SOL computes and prints a final covariance matrix and regression diagnostics is affected

by several keywords. `covariance` (NL2SOL's `covreq`) specifies the desired covariance approximation:

- 0 = default = none
- 1 or -1 ==> $\sigma^2 H^{-1} J^T J H^{-1}$
- 2 or -2 ==> $\sigma^2 H^{-1}$
- 3 or -3 ==> $\sigma^2 (J^T J)^{-1}$
- Negative values ==> estimate the final Hessian H by finite differences of function values only (using `fd_hessian_step_size`)
- Positive values ==> differences of gradients (using `fd_hessian_step_size`)

When `regression_diagnostics` (NL2SOL's `rdreq`) is specified and a positive-definite final Hessian approximation H is computed, NL2SOL computes and prints a regression diagnostic vector RD such that if omitting the i-th observation would cause alpha times the change in the solution that omitting the j-th observation would cause, then RD[i] = |alpha| RD[j]. The finite-difference step-size tolerance affecting H is `fd_hessian_step_size` (NL2SOL's `delta0` and `dltfdc`, as mentioned previously).

Table 5.22 provides the specification detail for the NL2SOL method dependent controls.

4.5.2 NLSSOL Method

NLSSOL is available as `nlssol_sqp` and supports unconstrained, bound-constrained, and generally-constrained problems. It exploits the structure of a least squares objective function through the periodic use of Gauss-Newton Hessian approximations to accelerate the SQP algorithm. Dakota provides access to the NLSSOL library through the **NLSSOLLeastSq** class. The method independent and method dependent controls are identical to those of NPSOL as described in NPSOL method independent controls and NPSOL method dependent controls.

4.5.3 Gauss-Newton Method

The Gauss-Newton algorithm is available as `optpp_g_newton` and supports unconstrained, bound-constrained, and generally-constrained problems. The code for the Gauss-Newton approximation (objective function value, gradient, and approximate Hessian defined from residual function values and gradients) is provided outside of OPT++ within **SNLLLeastSq::nlf2_evaluator_gn()**. When interfaced with the unconstrained, bound-constrained, and nonlinear interior point full-Newton optimizers from the OPT++ library, it provides a Gauss-Newton least squares capability which -- on zero-residual test problems -- can exhibit quadratic convergence rates near the solution. (Real problems almost never have zero residuals, i.e., perfect fits.)

Mappings for the method independent and dependent controls are the same as for the OPT++ optimization methods and are as described in OPT++ method independent controls and OPT++ method dependent controls. In particular, since OPT++ full-Newton optimizers provide the foundation for Gauss-Newton, the specifications from Table 5.6 are also applicable for `optpp_g_newton`.

Description	Keyword	Associated Data	Status	Default
Relative precision in least squares terms	function_-precision	real	Optional	1e-10
Absolute function convergence tolerance	absolute_-conv_tol	real	Optional	-1. (use NL2SOL internal default)
Convergence tolerance for change in parameter vector	x_conv_tol	real	Optional	-1. (use NL2SOL internal default)
Singular convergence tolerance	singular_-conv_tol	real	Optional	-1. (use NL2SOL internal default)
Step limit for sctol	singular_-radius	real	Optional	-1. (use NL2SOL internal default of 1)
False convergence tolerance	false_conv_-tol	real	Optional	-1. (use NL2SOL internal default)
Initial trust region radius	initial_-trust_radius	real	Optional	-1. (use NL2SOL internal default of 1)
Covariance post-processing	covariance	integer	Optional	0 (no covariance)
Regression diagnostics post-processing	regression_-diagnostics	none	Optional	no regression diagnostics

Table 4.22: Specification detail for NL2SOL method dependent controls.

4.6 Surrogate-Based Minimization Methods

In surrogate-based optimization (SBO) and surrogate-based nonlinear least squares (SBNLS), minimization occurs using a set of one or more approximations, defined from a surrogate model, that are built and periodically updated using data from a "truth" model. The surrogate model can be a global data fit (e.g., regression or interpolation of data generated from a design of computer experiments), a multipoint approximation, a local Taylor Series expansion, or a model hierarchy approximation (e.g., a low-fidelity simulation model), whereas the truth model involves a high-fidelity simulation model. The goals of surrogate-based methods are to reduce the total number of truth model simulations and, in the case of global data fit surrogates, to smooth noisy data with an easily navigated analytic function.

4.6.1 Surrogate-Based Local Method

In the surrogate-based local (SBL) method, a trust region approach is used to manage the minimization process to maintain acceptable accuracy between the surrogate model and the truth model (by limiting the range over which the surrogate model is trusted). The process involves a sequence of minimizations performed on the surrogate model and bounded by the trust region. At the end of each approximate minimization, the candidate optimum point is validated using the truth model. If sufficient decrease has been obtained in the truth model, the trust region is re-centered around the candidate optimum point and the trust region will either shrink, expand, or remain the same size depending on the accuracy with which the surrogate model predicted the truth model decrease. If sufficient decrease has not been attained, the trust region center is not updated and the entire trust region shrinks by a user-specified factor. The cycle then repeats with the construction of a new surrogate model, a minimization, and another test for sufficient decrease in the truth model. This cycle continues until convergence is attained.

The `surrogate_based_local` method must specify an optimization or least squares sub-method either by pointer using `approx_method_pointer` (e.g., 'NLP1') or by name using `approx_method_name` (e.g., 'npsol_sqp'). The former identifies a full sub-method specification for the sub-problem minimizer (which allows non-default minimizer settings), whereas the latter supports a streamlined specification (that employs default minimizer settings). For both cases, the `surrogate_based_local` method specification is responsible for using its `model_pointer` (see Method Independent Controls) to select a `surrogate` model (see Surrogate Model Controls). Any `model_pointer` identified in an approximate sub-method specification is ignored.

In addition to the method independent controls for `max_iterations` and `convergence_tolerance` described in Table 5.1, SBL algorithm controls include `soft_convergence_limit` (a soft convergence control for the SBL iterations which limits the number of consecutive iterations with improvement less than the convergence tolerance) and `truth_surrogate_bypass` (a flag for bypassing all lower level surrogates when performing truth verifications on a top level surrogate). Table 5.23 summarizes these SBL inputs.

The `trust_region` optional group specification can be used to specify the initial size of the trust region (using `initial_size`) relative to the total variable bounds, the minimum size of the trust region (using `minimum_size`), the contraction factor for the trust region size (using `contraction_factor`) used when the surrogate model is performing poorly, and the expansion factor for the trust region size (using `expansion_factor`) used when the the surrogate model is performing well. Two additional commands are the trust region size contraction threshold (using `contract_threshold`) and the trust region size expansion threshold (using `expand_threshold`). These two commands are related to what is called the trust region ratio, which is the actual decrease in the truth model divided by the predicted decrease in the truth model in the current trust region. The command `contract_threshold` sets the minimum acceptable value for the trust region ratio, i.e., values below this threshold cause the trust region to shrink for the next SBL iteration. The command `expand_threshold` determines the trust region value above which the trust region will expand for the next SBL iteration. Table 5.24 summarizes these trust region inputs.

Description	Keyword	Associated Data	Status	Default
Surrogate-based local method	surrogate_-based_local	none	Required group	N/A
Approximate sub-problem minimization method pointer	approx_-method_-pointer	string	Required (1 of 2 selections)	N/A
Approximate sub-problem minimization method name	approx_-method_name	string	Required (1 of 2 selections)	N/A
Soft convergence limit for SBL iterations	soft_-convergence_-limit	integer	Optional	5
Flag for bypassing lower level surrogates in truth verifications	truth_-surrogate_-bypass	none	Optional	no bypass

Table 4.23: Specification detail for surrogate-based local minimization method

Description	Keyword	Associated Data	Status	Default
Trust region group specification	trust_region	none	Optional group	N/A
Trust region initial size (relative to bounds)	initial_size	real	Optional	0.4
Trust region minimum size	minimum_size	real	Optional	1.e-6
Shrink trust region if trust region ratio is below this value	contract_-threshold	real	Optional	0.25
Expand trust region if trust region ratio is above this value	expand_-threshold	real	Optional	0.75
Trust region contraction factor	contraction_-factor	real	Optional	0.25
Trust region expansion factor	expansion_-factor	real	Optional	2.0

Table 4.24: Specification detail for trust region controls in surrogate-based local methods

For SBL problems with nonlinear constraints, a number of algorithm formulations exist as described in [Eldred and Dunlavy, 2006] and as summarized in the Advanced Examples section of the Models chapter of the Users Manual [Adams et al., 2010]. First, the "primary" functions (that is, the objective functions or calibration terms) in the approximate subproblem can be selected to be surrogates of the original primary functions (`original_primary`), a single objective function (`single_objective`) formed from the primary function surrogates, or either an augmented Lagrangian merit function (`augmented_lagrangian_objective`) or a Lagrangian merit function (`lagrangian_objective`) formed from the primary and secondary function surrogates. The former option may imply the use of a nonlinear least squares method, a multiobjective optimization method, or a single objective optimization method to solve the approximate subproblem, depending on the definition of the primary functions. The latter three options all imply the use of a single objective optimization method regardless of primary function definition. Second, the surrogate constraints in the approximate subproblem can be selected to be surrogates of the original constraints (`original_constraints`) or linearized approximations to the surrogate constraints (`linearized_constraints`), or constraints can be omitted from the subproblem (`no_constraints`). Following optimization of the approximate subproblem, the candidate iterate is evaluated using a merit function, which can be selected to be a simple penalty function with penalty ramped by SBL iteration number (`penalty_merit`), an adaptive penalty function where the penalty ramping may be accelerated in order to avoid rejecting good iterates which decrease the constraint violation (`adaptive_penalty_merit`), a Lagrangian merit function which employs first-order Lagrange multiplier updates (`lagrangian_merit`), or an augmented Lagrangian merit function which employs both a penalty parameter and zeroth-order Lagrange multiplier updates (`augmented_lagrangian_merit`). When an augmented Lagrangian is selected for either the subproblem objective or the merit function (or both), updating of penalties and multipliers follows the approach described in [Conn et al., 2000]. Following calculation of the merit function for the new iterate, the iterate is accepted or rejected and the trust region size is adjusted for the next SBL iteration. Iterate acceptance is governed either by a trust region ratio (`tr_ratio`) formed from the merit function values or by a filter method (`filter`); however, trust region resizing logic is currently based only on the trust region ratio. For infeasible iterates, constraint relaxation can be used for balancing constraint satisfaction and progress made toward an optimum. The command `constraint_relax` followed by a method name specifies the type of relaxation to be used. Currently, `homotopy` [Perez et al., 2004] is the only available method for constraint relaxation, and this method is dependent on the presence of the NPSOL library within the Dakota executable. Table 5.25 summarizes these constraint management inputs.

4.6.2 Surrogate-Based Global Method

The `surrogate_based_global` method differs from the `surrogate_based_local` method in a few ways. First, `surrogate_based_global` is not a trust region method. Rather, `surrogate_based_global` works in an iterative scheme where optimization is performed on a global surrogate using the same bounds during each iteration. In one iteration, the optimal solutions of the surrogate model are found, and then a selected set of these optimal surrogate solutions are passed to the next iteration. At the next iteration, these surrogate points are evaluated with the "truth" model, and then these points are added back to the set of points upon which the next surrogate is constructed. In this way, the optimization acts on a more accurate surrogate during each iteration, presumably driving to optimality quickly. This approach has no guarantee of convergence. It was originally designed for MOGA (a multi-objective genetic algorithm). Since genetic algorithms often need thousands or tens of thousands of points to produce optimal or near-optimal solutions, the use of surrogates can be helpful for reducing the truth model evaluations. Instead of creating one set of surrogates for the individual objectives and running the optimization algorithm on the surrogate once, the idea is to select points along the (surrogate) Pareto frontier, which can be used to supplement the existing points. In this way, one does not need to use many points initially to get a very accurate surrogate. The surrogate becomes more accurate as the iterations progress. Note that the user has the option of appending the optimal points from the surrogate model to the current set of truth points or using the optimal points from the surrogate model to replace the optimal set of points from

Description	Keyword	Associated Data	Status	Default
Approximate subproblem formulation	approx_-subproblem	original_-primary| single_-objective| augmented_-lagrangian_-objective| lagrangian_-objective original_-constraints| linearized_-constraints| no_-constraints	Optional group	original_-primary original_-constraints
SBL merit function	merit_-function	penalty_-merit| adaptive_-penalty_-merit| lagrangian_-merit| augmented_-lagrangian_-merit	Optional group	augmented_-lagrangian_-merit
SBL iterate acceptance logic	acceptance_-logic	tr_ratio| filter	Optional group	filter
SBL constraint relaxation method for infeasible iterates	constraint_-relax	homotopy	Optional group	no relaxation

Table 4.25: Specification detail for constraint management in surrogate-based local methods

the previous iteration. Although appending to the set is the default behavior, at this time we strongly recommend using the option `replace_points` because it appears to be more accurate and robust. Finally, the number of best solutions that will be passed from one iteration to another is governed by the iterator control `final_solutions`. If this is not specified, the surrogate-based global method will take all of the solutions available (e.g. all of the solutions in the Pareto front).

As for the `surrogate_based_local` method, the `surrogate_based_global` specification must identify a sub-method using either `approx_method_pointer` or `approx_method_name` and must identify a surrogate model (see Surrogate Model Controls) using its `model_pointer` (see Method Independent Controls). The only other algorithm control at this time is the method independent control for `max_iterations` described in Table 5.1. Table 5.26 summarizes the method dependent surrogate based global inputs.

Description	Keyword	Associated Data	Status	Default
Surrogate-based global method	`surrogate_based_global`	none	Required group	N/A
Approximate sub-problem minimization method pointer	`approx_method_pointer`	string	Required (1 of 2 selections)	N/A
Approximate sub-problem minimization method name	`approx_method_name`	string	Required (1 of 2 selections)	N/A
Replace points used in surrogate construction with best points from previous iteration	`replace_points`	none	Optional	Points appended, not replaced

Table 4.26: Specification detail for the surrogate-based global method

We have two cautionary notes before using the surrogate-based global method:

- One might first try a single minimization method coupled with a surrogate model prior to using the surrogate-based global method. This is essentially equivalent to setting `max_iterations` to 1 and will allow one to get a sense of what surrogate types are the most accurate to use for the problem. (Also note that one can specify that surrogates be built for all primary functions and constraints or for only a subset of these functions and constraints. This allows one to use a "truth" model directly for some of the response functions, perhaps due to them being much less expensive than other functions. This is outlined in Surrogate Model Controls.)

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

4.6.3 Efficient Global Method

The Efficient Global Optimization (EGO) method was first developed by Jones, Schonlau, and Welch [Jones et al., 1998]. In EGO, a stochastic response surface approximation for the objective function is developed based on

some sample points from the "true" simulation. The particular response surface used is a Gaussian process (GP). The GP allows one to calculate the prediction at a new input location as well as the uncertainty associated with that prediction. The key idea in EGO is to maximize the Expected Improvement Function (EIF). The EIF is used to select the location at which a new training point should be added to the Gaussian process model by maximizing the amount of improvement in the objective function that can be expected by adding that point. A point could be expected to produce an improvement in the objective function if its predicted value is better than the current best solution, or if the uncertainty in its prediction is such that the probability of it producing a better solution is high. Because the uncertainty is higher in regions of the design space with few observations, this provides a balance between exploiting areas of the design space that predict good solutions, and exploring areas where more information is needed. EGO trades off this "exploitation vs. exploration." The general procedure for these EGO-type methods is:

- Build an initial Gaussian process model of the objective function
- Find the point that maximizes the EIF. If the EIF value at this point is sufficiently small, stop.
- Evaluate the objective function at the point where the EIF is maximized. Update the Gaussian process model using this new point. Return to the previous step.

Note that several major differences exist between our implementation and that of [Jones et al., 1998]. First, rather than using a branch and bound method to find the point which maximizes the EIF, we use the DIRECT global optimization method (see DIRECT and NCSU Methods). Second, we support both global optimization and global nonlinear least squares as well as general nonlinear constraints through abstraction and subproblem recasting within the **SurrBasedMinimizer** and **EffGlobalMinimizer** classes.

The efficient global method is in prototype form. Currently, we do not expose any specification controls for the underlying Gaussian process model used or for the optimization of the expected improvement function (which is currently performed by the NCSU DIRECT algorithm using its internal defaults). The efficient global algorithm is specified by the keyword `efficient_global` along with an optional `seed` specification, as shown in in Table 5.27 below. By default, EGO uses the Surfpack GP (Kriging) model, but the Dakota implementation may be selected instead. If `use_derivatives` is specified the GP model will be built using available derivative data (Surfpack GP only).

Description	Keyword	Associated Data	Status	Default
Efficient global method	`efficient_global`	none	Required group	N/A
Random seed	`seed`	integer	Optional	Time based seed: nonrepeatable
GP selection	`gaussian_process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_derivatives`	none	Optional	Use function values only

Table 4.27: Specification detail for the efficient global method

4.7 Uncertainty Quantification Methods

Dakota provides a variety of methods for propagating uncertainty. Aleatory uncertainty refers to inherent variability, irreducible uncertainty, or randomness, and is addressed with the probabilistic methods described in Aleatory

Uncertainty Quantification Methods. Epistemic uncertainty refers to subjective uncertainty, reducible uncertainty, model form uncertainty, or uncertainty due to lack of knowledge, and is addressed using the non-probabilistic approaches described in Epistemic Uncertainty Quantification Methods. In general, we refer to both classes of uncertainty quantification methods in Dakota as nondeterministic methods. In the descriptions below, we described the issues and specification controls that are common to both aleatory and epistemic uncertainty quantification.

Dakota's nondeterministic methods make use of a few method independent controls. `max_iterations` and `convergence_tolerance` are used by iterated local and global reliability, stochastic expansions undergoing automated refinement, and optimization-based epistemic methods. `output` level controls are also employed by several methods:

- optional output of probability density functions (PDFs), for settings of `normal` or higher (sampling methods, including numerical statistics for stochastic expansion methods)
- optional output of sample evaluations performed on approximations, for settings of `verbose` or higher (stochastic expansion methods)
- optional tabular output of numerical integration points and weights, for settings of `verbose` or higher (stochastic expansion methods)
- optional output of local sensitivities computed from global approximations, for settings of `normal` or higher (stochastic expansion methods)
- optional output of statistical quantities of interest (QOI) for iterations prior to convergence of an automated refinement process (stochastic expansion methods with a `debug` setting)

Each of the uncertainty quantification techniques is standardized on support for `response_levels`, `probability_levels`, `reliability_levels`, and `gen_reliability_levels` specifications along with their optional `num_response_levels`, `num_probability_levels`, `num_reliability_levels` and `num_gen_reliability_levels` keys, except for a few exceptions where certain level mappings cannot be supported: `global_reliability`, `importance`, `local_evidence`, and `global_evidence` do not support mappings involving `reliability_levels`, and `local_interval_est` and `global_interval_est` do not support any level mappings. The keys define the distribution of the levels among the different response functions. For example, the following specification

```
num_response_levels = 2 4 3
response_levels = 1. 2. .1 .2 .3 .4 10. 20. 30.
```

would assign the first two response levels (1., 2.) to response function 1, the next four response levels (.1, .2, .3, .4) to response function 2, and the final three response levels (10., 20., 30.) to response function 3. If the `num_response_levels` key were omitted from this example, then the response levels would be evenly distributed among the response functions (three levels each in this case).

The `response_levels` specification provides the target response values for generating probabilities, reliabilities, or generalized reliabilities (forward mapping). The selection among these possible results for the forward mapping is performed with the `compute` keyword followed by either `probabilities`, `reliabilities`, or `gen_reliabilities`. For example, specifying a `response_level` of 52.3 followed with `compute probabilities` will result in the calculation that the (uncertain) output value is less than or equal to 52.3, given the uncertain distributions on the inputs. Conversely, the `probability_levels`, `reliability_levels`, and `gen_reliability_levels` specifications provide target levels for which response values will be computed (inverse mapping). For example, specifying a `probability_level` of 0.95 will result in the calculation of a response value which corresponds to the 95th percentile of the output distribution. Specifications of `response_levels`, `probability_levels`, `reliability_levels`, and `gen_reliability_levels` may be combined within the calculations for each response function. The mapping results (probabilities,

reliabilities, or generalized reliabilities for the forward mapping and response values for the inverse mapping) define the final statistics of the nondeterministic analysis that can be accessed for use at a higher level (via the primary and secondary mapping matrices for nested models; see Nested Model Controls).

Sets of response-probability pairs computed with the forward/inverse mappings define either a cumulative distribution function (CDF) or a complementary cumulative distribution function (CCDF) for each response function. In the case of evidence-based epistemic methods, this is generalized to define either cumulative belief and plausibility functions (CBF and CPF) or complementary cumulative belief and plausibility functions (CCBF and CCPF) for each response function, where a forward mapping involves computing the belief and plausibility probability level for a specified response level and an inverse mapping involves computing the belief and plausibility response level for either a specified probability level or a specified generalized reliability level (two results for each level mapping in the evidence-based epistemic case, instead of the one result for each level mapping in the aleatory case). The selection of a CDF/CBF/CPF or CCDF/CCBF/CCPF can be performed with the `distribution` keyword followed by either `cumulative` for the CDF/CBF/CPF option or `complementary` for the CCDF/CCBF/CCPF option. This selection also defines the sign of the reliability or generalized reliability indices. Table 5.28 provides the specification detail for the forward/inverse mappings used by each of the nondeterministic analysis methods.

Different nondeterministic methods have differing support for uncertain variable distributions. Tables 5.29, 5.30, and 5.31 summarize the uncertain variables that are available for use by the different methods, where a "-" indicates that the distribution is not supported by the method, a "U" means the uncertain input variables of this type must be uncorrelated, a "C" denotes that correlations are supported involving uncertain input variables of this type, and an "A" means the appropriate variables must be specified as active in the variables specification block. For example, if one wants to support sampling or a stochastic expansion method over both continuous uncertain and continuous state variables, the specification `active all` must be listed in the variables specification block. Additional notes include:

- we have four variants for stochastic expansions (SE), listed as Wiener, Askey, Extended, and Piecewise which draw from different sets of basis polynomials. The term stochastic expansion indicates polynomial chaos and stochastic collocation collectively, although the Piecewise option is only currently supported for stochastic collocation. Refer to Polynomial chaos expansion method and Stochastic collocation method for additional information on these three options.

- methods supporting the epistemic interval distributions have differing approaches: `sampling` and the `lhs` option of `global_interval_est` model the interval basic probability assignments (BPAs) as continuous histogram bin distributions for purposes of generating samples; `local_interval_est` and the `ego` option of `global_interval_est` ignore the BPA details and models these variables as simple bounded regions defined by the cell extremes; and `local_evidence` and `global_evidence` model the interval specifications as true BPAs.

4.7.1 Aleatory Uncertainty Quantification Methods

Aleatory uncertainty is also known as inherent variability, irreducible uncertainty, or randomness. An example of aleatory uncertainty is the distribution of height in a population, as it is characterized by the availability of sufficient data to accurately model the form of the variation. For this reason, aleatory uncertainty is typically modeled using probabilistic approaches through the specification of probability distributions to represent the uncertain input variables and the propagation of this uncertainty using probability theory. The probabilistic approaches supported in Dakota include sampling, local and global reliability, polynomial chaos, and stochastic collocation, which may be used to propagate random variables described by Normal Distribution, Lognormal

Description	Keyword	Associated Data	Status	Default
Distribution type	distribution	cumulative | complementary	Optional group	cumulative (CDF)
Response levels	response_levels	list of reals	Optional group	No CDF/CCDF probabilities/reliabilities to compute
Number of response levels	num_response_levels	list of integers	Optional	response_levels evenly distributed among response functions
Target statistics for response levels	compute	probabilities | reliabilities | gen_reliabilities	Optional	probabilities
Probability levels	probability_levels	list of reals	Optional group	No CDF/CCDF response levels to compute
Number of probability levels	num_probability_levels	list of integers	Optional	probability_levels evenly distributed among response functions
Reliability levels	reliability_levels	list of reals	Optional group	No CDF/CCDF response levels to compute
Number of reliability levels	num_reliability_levels	list of integers	Optional	reliability_levels evenly distributed among response functions
Generalized reliability levels	gen_reliability_levels	list of reals	Optional group	No CDF/CCDF response levels to compute
Number of generalized reliability levels	num_gen_reliability_levels	list of integers	Optional	gen_reliability_levels evenly distributed among response functions

Table 4.28: Specification detail for forward/inverse level mappings

Distribution Type	Sampling	Local Reliability	Global Reliability	Wiener SE	Askey SE	Extended SE	Piecewise SE	Local Interval	Global Interval	Local Evidence	Global Evidence
Normal	C	C	C	C	C	C	-	-	-	-	-
Bounded Normal	C	U	U	U	U	U	U	-	-	-	-
Lognormal	C	C	C	C	C	U	-	-	-	-	-
Bounded Lognormal	C	U	U	U	U	U	U	-	-	-	-
Uniform	C	C	C	C	U	U	U	-	-	-	-
Loguniform	C	U	U	U	U	U	U	-	-	-	-
Triangular	C	U	U	U	U	U	U	-	-	-	-
Exponential	C	C	C	C	U	U	-	-	-	-	-
Beta	C	U	U	U	U	U	U	-	-	-	-
Gamma	C	C	C	C	U	U	-	-	-	-	-
Gumbel	C	C	C	C	C	U	-	-	-	-	-
Frechet	C	C	C	C	C	U	-	-	-	-	-
Weibull	C	C	C	C	C	U	-	-	-	-	-
Continuous Histogram Bin	C	U	U	U	U	U	U	-	-	-	-

Table 4.29: Summary of Distribution Types supported by Nondeterministic Methods, Part I (Continuous Aleatory Types)

Distribution Type	Sampling	Local Reliability	Global Reliability	Wiener SE	Askey SE	Extended SE	Piecewise SE	Local Interval	Global Interval	Local Evidence	Global Evidence
Poisson	C	-	-	-	-	-	-	-	-	-	-
Binomial	C	-	-	-	-	-	-	-	-	-	-
Negative Binomial	C	-	-	-	-	-	-	-	-	-	-
Geometric	C	-	-	-	-	-	-	-	-	-	-
Hypergeometric	C	-	-	-	-	-	-	-	-	-	-
Discrete Histogram Point	C	-	-	-	-	-	-	-	-	-	-

Table 4.30: Summary of Distribution Types supported by Nondeterministic Methods, Part II (Discrete Aleatory Types)

Distribution Type	Sampling	Local Reliability	Global Reliability	Wiener SE	Askey SE	Extended SE	Piecewise SE	Local Interval	Global Interval	Local Evidence	Global Evidence
Interval	U	-	U,A	U,A	U,A	U,A	U,A	U	U	U	U
Continuous Design	U,A	-	U,A	U,A	U,A	U,A	U,A	-	-	-	-
Discrete Design Range, Int Set, Real Set	U,A	-	-	-	-	-	-	-	-	-	-
Continuous State	U,A	-	U,A	U,A	U,A	U,A	U,A	-	-	-	-
Discrete State Range, Int Set, Real Set	U,A	-	-	-	-	-	-	-	-	-	-

Table 4.31: Summary of Distribution Types supported by Nondeterministic Methods, Part III (Epistemic, Design, and State Types)

Distribution, Uniform Distribution, Loguniform Distribution, Triangular Distribution, Exponential Distribution, Beta Distribution, Gamma Distribution, Gumbel Distribution, Frechet Distribution, Weibull Distribution, Histogram Bin Distribution, Poisson Distribution, Binomial Distribution, Negative Binomial Distribution, Geometric Distribution, Hypergeometric Distribution, and/or Histogram Point Distribution.

4.7.1.1 Nondeterministic sampling method

The nondeterministic sampling method is selected using the `sampling` specification. This method draws samples from the specified uncertain variable probability distributions and propagates them through the model to obtain statistics on the output response functions of interest. Dakota provides access to nondeterministic sampling methods through the combination of the **NonDSampling** base class and the **NonDLHSSampling** derived class.

CDF/CCDF probabilities are calculated for specified response levels using a simple binning approach. Response levels are calculated for specified CDF/CCDF probabilities and generalized reliabilities by indexing into a sorted samples array (the response levels computed are not interpolated and will correspond to one of the sampled values). CDF/CCDF reliabilities are calculated for specified response levels by computing the number of sample standard deviations separating the sample mean from the response level. Response levels are calculated for specified CDF/CCDF reliabilities by projecting out the prescribed number of sample standard deviations from the sample mean.

The `seed` integer specification specifies the seed for the random number generator which is used to make sampling studies repeatable, and `rng` specifies which random number generator is used. The `fixed_seed` flag is relevant if multiple sampling sets will be generated during the course of a strategy (e.g., surrogate-based optimization, optimization under uncertainty). Specifying this flag results in the reuse of the same seed value for each of these multiple sampling sets, which can be important for reducing variability in the sampling results. However, this behavior is not the default as the repetition of the same sampling pattern can result in a modeling weakness that an optimizer could potentially exploit (resulting in actual reliabilities that are lower than the estimated reliabilities). In either case (`fixed_seed` or not), the study is repeatable if the user specifies a `seed` and the study is random is the user omits a `seed` specification.

The number of samples to be evaluated is selected with the `samples` integer specification. The algorithm used to generate the samples can be specified using `sample_type` followed by either `random`, for pure random Monte Carlo sampling, or `lhs`, for Latin Hypercube sampling.

If the user wants to increment a particular set of samples with more samples to get better estimates of mean, variance, and percentiles, one can select `incremental_random` or `incremental_lhs` as the `sample_type`. Note that a preliminary sample of size N must have already been performed, and a `dakota.rst` restart file must be available from this original sample. For example, say a user performs an initial study using `lhs` as the `sample_type`, and generates 50 samples. If the user creates a new input file where `samples` is now specified to be 100, the `sample_type` is defined to be `incremental_lhs` or `incremental_random`, and `previous_samples` is specified to be 50, the user will get 50 new LHS samples which maintain both the correlation and stratification of the original LHS sample. The N new samples will be combined with the N original samples to generate a combined sample of size 2N. The syntax for running the second sample set is: `dakota -i input2.in -r dakota.rst`, where `input2.in` is the file which specifies incremental sampling. Note that the number of samples in the second set MUST currently be 2 times the number of previous samples, although incremental sampling based on any power of two may be supported in future releases.

The nondeterministic sampling method also supports sampling over different types of variables, depending on what is specified as `active` in the variables block of the input specification. Normally, `sampling` generates samples only for the uncertain variables, and treats any design or state variables as constants. However, if `active all` is specified in the variables block, sampling will be performed over all variables, including uncertain, design,

and state. In this case, the sampling algorithm will treat any continuous design or continuous state variables as parameters with uniform probability distributions between their upper and lower bounds. Samples are then generated over all of the continuous variables (design, uncertain, and state) in the variables specification. This is similar to the behavior of the design of experiments methods described in Design of Computer Experiments Methods, since they will also generate samples over all continuous design, uncertain, and state variables in the variables specification. However, the design of experiments methods will treat all variables as being uniformly distributed between their upper and lower bounds, whereas the `sampling` method will sample the uncertain variables within their specified probability distributions. If further granularity of sampling is necessary for uncertain variables, one can specify `active epistemic` or `active aleatory` to specify sampling over only epistemic uncertain or only aleatory uncertain variables, respectively. In the case where one wants to generate samples only over state variables, one would specify `active state` in the variables specification block.

Finally, the nondeterministic sampling method supports two types of sensitivity analysis. In this context of sampling, we take sensitivity analysis to be global, not local as when calculating derivatives of output variables with respect to input variables. Our definition is similar to that of [Saltelli et al., 2004]: "The study of how uncertainty in the output of a model can be apportioned to different sources of uncertainty in the model input." As a default, Dakota provides correlation analyses when running LHS. Correlation tables are printed with the simple, partial, and rank correlations between inputs and outputs. These can be useful to get a quick sense of how correlated the inputs are to each other, and how correlated various outputs are to inputs. In addition, we have the capability to calculate sensitivity indices through variance based decomposition using the keyword `variance_based_decomp`. Variance based decomposition is a way of using sets of samples to understand how the variance of the output behaves, with respect to each input variable. A larger value of the sensitivity index, S_i, means that the uncertainty in the input variable i has a larger effect on the variance of the output. More details on the calculations and interpretation of the sensitivity indices can be found in [Saltelli et al., 2004] and [Weirs et al., 2010]. Note that `variance_based_decomp` is extremely computationally intensive since replicated sets of sample values are evaluated. If the user specified a number of samples, N, and a number of nondeterministic variables, M, variance-based decomposition requires the evaluation of N$*$(M+2) samples. To obtain sensitivity indices that are reasonably accurate, we recommend that N, the number of samples, be at least one hundred and preferably several hundred or thousands. Because of the computational cost, `variance_based_decomp` is turned off as a default. Table 5.32 provides details of the nondeterministic sampling specifications beyond those of Table 5.28.

4.7.1.2 Local reliability methods

Local reliability methods are selected using the `local_reliability` specification and are implemented within the **NonDLocalReliability** class. These methods compute approximate response function distribution statistics based on specified uncertain variable probability distributions. Each of the local reliability methods can compute forward and inverse mappings involving response, probability, reliability, and generalized reliability levels.

The Mean Value method (MV, also known as MVFOSM in [Haldar and Mahadevan, 2000]) is the simplest, least-expensive method in that it estimates the response means, response standard deviations, and all CDF/CCDF forward/inverse mappings from a single evaluation of response functions and gradients at the uncertain variable means. This approximation can have acceptable accuracy when the response functions are nearly linear and their distributions are approximately Gaussian, but can have poor accuracy in other situations.

All other reliability methods perform an internal nonlinear optimization to compute a most probable point (MPP) of failure. A sign convention and the distance of the MPP from the origin in the transformed standard normal space ("u-space") define the reliability index, as explained in the section on Reliability Methods in the Uncertainty Quantification chapter of the Users Manual [Adams et al., 2010]. The reliability can then be converted to a probability using either first- or second-order integration, may then be refined using importance sampling, and

Description	Keyword	Associated Data	Status	Default
Nondeterministic sampling method	`sampling`	none	Required group	N/A
Sampling type	`sample_type`	`random` | `lhs` | `incremental_random` | `incremental_lhs`	Optional group	`lhs`
Number of samples	`samples`	integer	Optional	minimum required
Previous samples for incremental approaches	`previous_samples`	integer	Optional	0 (no previous_samples)
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple runs
Random number generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
Variance based decomposition (VBD)	`variance_based_decomp`	none	Optional	No VBD analysis
VBD tolerance for omitting small indices	`drop_tolerance`	none	Optional	All VBD indices displayed

Table 4.32: Specification detail for nondeterministic sampling method

finally may be converted to a generalized reliability index. The forward reliability analysis algorithm of computing reliabilities/probabilities for specified response levels is called the Reliability Index Approach (RIA), and the inverse reliability analysis algorithm of computing response levels for specified probability levels is called the Performance Measure Approach (PMA). The different RIA/PMA algorithm options are specified using the `mpp_search` specification which selects among different limit state approximations that can be used to reduce computational expense during the MPP searches. The `x_taylor_mean` MPP search option performs a single Taylor series approximation in the space of the original uncertain variables ("x-space") centered at the uncertain variable means, searches for the MPP for each response/probability level using this approximation, and performs a validation response evaluation at each predicted MPP. This option is commonly known as the Advanced Mean Value (AMV) method. The `u_taylor_mean` option is identical to the `x_taylor_mean` option, except that the approximation is performed in u-space. The `x_taylor_mpp` approach starts with an x-space Taylor series at the uncertain variable means, but iteratively updates the Taylor series approximation at each MPP prediction until the MPP converges. This option is commonly known as the AMV+ method. The `u_taylor_mpp` option is identical to the `x_taylor_mpp` option, except that all approximations are performed in u-space. The order of the Taylor-series approximation is determined by the corresponding `responses` specification and may be first or second-order. If second-order (methods named AMV^2 and AMV^2+ in [Eldred and Bichon, 2006]), the series may employ analytic, finite difference, or quasi Hessians (BFGS or SR1). The `x_two_point` MPP search option uses an x-space Taylor series approximation at the uncertain variable means for the initial MPP prediction, then utilizes the Two-point Adaptive Nonlinear Approximation (TANA) outlined in [Xu and Grandhi, 1998] for all subsequent MPP predictions. The `u_two_point` approach is identical to `x_two_point`, but all the approximations are performed in u-space. The `x_taylor_mpp` and `u_taylor_mpp`, `x_two_point` and `u_two_point` approaches utilize the `max_iterations` and `convergence_tolerance` method independent controls to control the convergence of the MPP iterations (the maximum number of MPP iterations per level is limited by `max_iterations`, and the MPP iterations are considered converged when $\| \mathbf{u}^{(k+1)} - \mathbf{u}^{(k)} \|_2 <$ `convergence_tolerance`). And, finally, the `no_approx` option performs the MPP search on the original response functions without the use of any approximations. The optimization algorithm used to perform these MPP searches can be selected to be either sequential quadratic programming (uses the `npsol_sqp` optimizer) or nonlinear interior point (uses the `optpp_q_newton` optimizer) algorithms using the `sqp` or `nip` keywords.

In addition to the MPP search specifications, one may select among different integration approaches for computing probabilities at the MPP by using the `integration` keyword followed by either `first_order` or `second_order`. Second-order integration employs the formulation of [Hohenbichler and Rackwitz, 1988] (the approach of [Breitung, 1984] and the correction of [Hong 1999] are also implemented, but are not active). Combining the `no_approx` option of the MPP search with first- and second-order integrations results in the traditional first- and second-order reliability methods (FORM and SORM). These integration approximations may be subsequently refined using importance sampling. The `refinement` specification allows the seletion of basic importance sampling (`import`), adaptive importance sampling (`adapt_import`), or multimodal adaptive importance sampling (`mm_adapt_import`), along with the specification of number of samples (`samples`) and random seed (`seed`). Additional details on these methods are available in [Eldred et al., 2004b] and [Eldred and Bichon, 2006] and in the Uncertainty Quantification Capabilities chapter of the Users Manual [Adams et al., 2010].

Table 5.33 provides details of the local reliability method specifications beyond those of Table 5.28.

4.7.1.3 Global reliability methods

Global reliability methods are selected using the `global_reliability` specification and are implemented within the **NonDGlobalReliability** class. These methods do not support forward/inverse mappings involving `reliability_levels`, since they never form a reliability index based on distance in u-space. Rather they use a Gaussian process model to form an approximation to the limit state (based either in x-space via the `x_`–

Description	Keyword	Associated Data	Status	Default
Reliability method	local_-reliability	none	Required group	N/A
MPP search type	mpp_search	x_taylor_-mean | u_taylor_-mean | x_taylor_mpp | u_taylor_mpp | x_two_point | u_two_point | no_approx	Optional group	No MPP search (MV method)
MPP search algorithm	sqp, nip	none	Optional	NPSOL's SQP algorithm
Integration method	integration	first_order | second_order	Optional group	First-order integration
Refinement method	refinement	import | adapt_import | mm_adapt_-import	Optional group	No refinement
Refinement samples	samples	integer	Optional	0
Refinement seed	seed	integer	Optional group	randomly generated seed

Table 4.33: Specification detail for local reliability methods

gaussian_process specification or in u-space via the u_gaussian_process specification), followed by probability estimation based on multimodal adaptive importance sampling (see [Bichon et al., 2007]) and [Bichon et al., 2008]). These probability estimates may then be transformed into generalized reliability levels if desired. At this time, inverse reliability analysis (mapping probability or generalized reliability levels into response levels) is not implemented. The Gaussian process model approximation to the limit state is formed over the aleatory uncertain variables by default, but may be extended to also capture the effect of design, epistemic uncertain, and state variables. If this is desired, one must use the appropriate controls to specify the active variables in the variables specification block. By default, the Surfpack GP (Kriging) model is used, but the Dakota implementation may be selected instead. If use_derivatives is specified the GP model will be built using available derivative data (Surfpack GP only).

Table 5.34 provides details of the global reliability method specifications beyond those of Table 5.28.

Description	Keyword	Associated Data	Status	Default
Global reliability method	global_reliability	none	Required group	N/A
Approximation type	x_gaussian_process | u_gaussian_process	none	Required	N/A
GP selection	gaussian_process	surfpack | dakota	Optional	Surfpack Gaussian process
Derivative usage	use_derivatives	none	Optional	Use function values only
Random seed for initial GP construction	seed	integer	Optional	Time based seed: nonrepeatable

Table 4.34: Specification detail for global reliability methods

4.7.1.4 Importance sampling methods

Importance sampling is a method that allows one to estimate statistical quantities such as failure probabilities (e.g. the probability that a response quantity will exceed a threshold or fall below a threshold value) in a way that is more efficient than Monte Carlo sampling. The core idea in importance sampling is that one generates samples that preferentially samples important regions in the space (e.g. in or near the failure region or user-defined region of interest), and then appropriately weights the samples to obtain an unbiased estimate of the failure probability [Srinivasan, 2002]. In importance sampling, the samples are generated from a density which is called the importance density: it is not the original probability density of the input distributions. The importance density should be centered near the failure region of interest. For black-box simulations such as those commonly interfaced with Dakota, it is difficult to specify the importance density a priori: the user often does not know where the failure region lies, especially in a high-dimensional space.[Swiler and West, 2010]. We have developed two importance sampling approaches which do not rely on the user explicitly specifying an importance density.

The first method is based on ideas in reliability modeling Local reliability methods. An initial Latin Hypercube sampling is performed to generate an initial set of samples. These initial samples are augmented with samples from an importance density as follows: The variables are transformed to standard normal space. In the transformed space, the importance density is a set of normal densities centered around points which are in the failure region. Note that this is similar in spirit to the reliability methods, in which importance sampling is centered around a Most Probable Point (MPP). In the case of the LHS samples, the importance sampling density will simply

by a mixture of normal distributions centered around points in the failure region. The options for importance sampling are as follows: `import` centers a sampling density at one of the initial LHS samples identified in the failure region. It then generates the importance samples, weights them by their probability of occurence given the original density, and calculates the required probability (CDF or CCDF level). `adapt_import` is the same as `import` but is performed iteratively until the failure probability estimate converges. `mm_adapt_import` starts with all of the samples located in the failure region to build a multimodal sampling density. First, it uses a small number of samples around each of the initial samples in the failure region. Note that these samples are allocated to the different points based on their relative probabilities of occurrence: more probable points get more samples. This early part of the approach is done to search for "representative" points. Once these are located, the multimodal sampling density is set and then `mm_adapt_import` proceeds similarly to `adapt_import` (sample until convergence).

Table 5.35 provides details of the `importance_sampling` method.

Description	Keyword	Associated Data	Status	Default
Importance sampling method	`importance_sampling`	none	Required group	N/A
Type of importance sampling performed	`import` | `adapt_import` | `mm_adapt_import`	none	Required	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Random number generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
File name for previously generated points to be imported as the basis for the initial GP	`points_file`	string	annotated|freeform	N/A

Table 4.35: Specification detail for the LHS-based importance sampling method

The second importance sampling method in Dakota, `gpais`, is the one we recommend, at least for problems that have a relatively small number of input variables (e.g. less than 10-20). This method, Gaussian Process Adaptive Importance Sampling, is outlined in the paper [Dalbey and Swiler, 2012]. This method starts with an initial set of LHS samples and adds samples one at a time, with the goal of adaptively improving the estimate of the ideal importance density during the process. The approach uses a mixture of component densities. An itterative process is used to construct the sequence of improving component densities. At each iteration, a Gaussian process (GP) surrogate is used to help identify areas in the space where failure is likely to occur. The GPs are not used to directly calculate the failure probability; they are only used to approximate the importance density. Thus, the Gaussian process adaptive importance sampling algorithm overcomes limitations involving using a potentially inaccurate surrogate model directly in importance sampling calculations.

Table 5.36 provides details of the `gpais` method for Gaussian process adaptive importance sampling.

Description	Keyword	Associated Data	Status	Default
Gaussian process adaptive importance sampling method	gpais	none	Required group	N/A
Number of initial LHS samples	samples	integer	Required	N/A
Number of samples on the emulator to generate a new true sample each iteration	emulator_-samples	integer	Optional	10,000
Random seed	seed	integer	Optional	randomly generated seed
Random number generator	rng	mt19937 | rnum2	Optional	Mersenne twister (mt19937)
File name for previously generated points to be imported as the basis for the initial GP	points_file	string	annotated|freeform	N/A

Table 4.36: Specification detail for Gaussian process adaptive importance sampling method

4.7.1.5 Adaptive sampling methods

The goal in performing adaptive sampling is to construct a surrogate model that can be used as an accurate predictor to some expensive simulation, thus it is to one's advantage to build a surrogate that minimizes the error over the entire domain of interest using as little data as possible from the expensive simulation. The adaptive part alludes to the fact that the surrogate will be refined by focusing samples of the expensive simulation on particular areas of interest rather than rely on random selection or standard space-filling techniques.

At a high-level, the adaptive sampling pipeline is a four-step process:

1. Evaluate the expensive simulation (referred to as the true model) at initial sample point
2. Fit/refit a surrogate model
3. Create a candidate set and score based on information from surrogate
4. Select a candidate point to evaluate the true model and Repeat 2-4

In terms of the Dakota implementation, the adaptive sampling method currently uses Latin Hypercube sampling (LHS) to generate the initial points in Step 1 above. For Step 2, we use a Gaussian process model. The user can specify the `fitness_metric` used to select the next point (or points) to evaluate and add to the set. The fitness metrics used for scoring candidate points include: `predicted_variance`, `distance`, and `gradient`. The predicted variance metric uses the predicted variance of the Gaussian process surrogate as the score of a candidate point. Thus, the adaptively chosen points will be in areas of highest uncertainty according to the Gaussian process model. The distance metric calculates the Euclidean distance in domain space between the candidate and its

nearest neighbor in the set of points already evaluated on the true model. Therefore, the most undersampled area of the domain will always be selected. Note that this is a space-filling metric. The gradient metric calculates the score as the absolute value of the difference in range space (the outputs) of the two points. The output values used are predicted from the surrogate model. This method attempts to evenly fill the range space of the surrogate.

At each iteration (e.g. each loop of Steps 2-4 above), a Latin Hypercube sample is generated (a new one, different from the initial sample) and the surrogate model is evaluated at this points. These are the candidate points that are then evaluated according to the fitness metric. The number of candidates used in practice should be high enough to fill most of the input domain: we recommend at least hundreds of points for a low-dimensional problem. All of the candidates (samples on the emulator) are given a score and then the highest-scoring candidate is selected to be evaluated on the true model.

The adaptive sampling method also can generate batches of points to add at a time. With batch or multi-point selection, the true model can be evaluated in parallel and thus increase throughput before refitting our surrogate model. This proposes a new challenge as the problem of choosing a single point and choosing multiple points off a surrogate are fundamentally different. Selecting the `n` best scoring candidates is more than likely to generate a set of points clustered in one area which will not be conducive to adapting the surrogate.

We have implemented several strategies for batch selection of points. These are described in the User's manual and are the subject of active research. The number of points to add in each batch is specified with `batch_size`. Briefly, the `batch_selection` strategies include:

1. `naive`: This strategy will select the `n` highest scoring candidates regardless of their position. This tends to group an entire round of points in the same area.
2. `distance_penalty` In this strategy, the highest scoring candidate is selected and then all remaining candidates are re-scored with a distance penalization factor added in to the score.
3. `topology` In this strategy we look at the topology of the scoring function and select the `n` highest maxima in the topology. To determine local maxima, we construct the approximate Morse-Smale complex. This strategy does require the user to have the Morse-Smale package.
4. `constant_liar` The strategy first selects the highest scoring candidate, and then refits the surrogate using a "lie" value at the point selected and repeats until `n` points have been selected whereupon the lie values are removed from the surrogate and the selected points are evaluated on the true model and the surrogate is refit with these values.

Table 5.37 provides details of the `adaptive_sampling` method for adaptive sampling.

4.7.1.6 Polynomial chaos expansion method

The polynomial chaos expansion (PCE) is a general framework for the approximate representation of random response functions in terms of finite-dimensional series expansions in standardized random variables

$$R = \sum_{i=0}^{P} \alpha_i \Psi_i(\xi)$$

where α_i is a deterministic coefficient, Ψ_i is a multidimensional orthogonal polynomial and ξ is a vector of standardized random variables. An important distinguishing feature of the methodology is that the functional relationship between random inputs and outputs is captured, not merely the output statistics as in the case of many

Description	Keyword	Associated Data	Status	Default
adaptive sampling method	adaptive_-sampling	none	Required group	N/A
Number of initial LHS samples	samples	integer	Required	N/A
Number of samples on the emulator to generate a new true sample each iteration	emulator_-samples	integer	Optional	400
Random seed	seed	integer	Optional	Randomly generated seed
Random number generator	rng	mt19937 | rnum2	Optional	Mersenne twister (mt19937)
Fitness metric	fitness_-metric	predicted_-variance | distance | gradient	Optional	predicted_-variance
Batch selection strategy	batch_-selection	naive | distance_-penalty | topology | constant_-liar	Optional	naive
Batch size (number of points added each iteration)	batch_size	integer	Optional	1
File name for previously generated points to be imported as the basis for the initial GP	points_file	string	annotated|freeform	N/A

Table 4.37: Specification detail for adaptive sampling method

nondeterministic methodologies. Dakota provides access to PCE methods through the **NonDPolynomialChaos** class. Refer to the Uncertainty Quantification Capabilities chapter of the Users Manual [Adams et al., 2010] for additional information on the PCE algorithm.

To select the basis Ψ_i of the expansion, three approaches may be employed, as previously shown in Tables 5.29, 5.30, and 5.31: Wiener, Askey, and Extended. The Wiener option uses a Hermite orthogonal polynomial basis for all random variables and employs the same nonlinear variable transformation as the local and global reliability methods (and therefore has the same variable support). The Askey option, however, employs an extended basis of Hermite, Legendre, Laguerre, Jacobi, and generalized Laguerre orthogonal polynomials. The Extended option avoids the use of any nonlinear variable transformations by augmenting the Askey approach with numerically-generated orthogonal polynomials for non-Askey probability density functions. The selection of Wiener versus Askey versus Extended is partially automated and partially under the user's control. The Extended option is the default and supports only Gaussian correlations (see Tables 5.29, 5.30, and 5.31). This default can be overridden by the user by supplying the keyword `askey` to request restriction to the use of Askey bases only or by supplying the keyword `wiener` to request restriction to the use of exclusively Hermite bases. If needed to support prescribed correlations (not under user control), the Extended and Askey options will fall back to the Wiener option *on a per variable basis*. If the prescribed correlations are also unsupported by Wiener expansions, then Dakota will exit with an error. Additional details include:

- Askey polynomial selections include Hermite for normal (optimal) as well as bounded normal, lognormal, bounded lognormal, gumbel, frechet, and weibull (sub-optimal); Legendre for uniform (optimal) as well as loguniform, triangular, and bin-based histogram (sub-optimal); Laguerre for exponential (optimal); Jacobi for beta (optimal); and generalized Laguerre for gamma (optimal).

- Extended polynomial selections replace each of the sub-optimal Askey basis selections with numerically-generated polynomials that are orthogonal to the prescribed probability density functions (for bounded normal, lognormal, bounded lognormal, loguniform, triangular, gumbel, frechet, weibull, and bin-based histogram).

The `p_refinement` keyword specifies the usage of automated polynomial order refinement, which can be either `uniform` or `dimension_adaptive`. The `dimension_adaptive` option is supported for the tensor-product quadrature and Smolyak sparse grid options (see Table 5.39 below), and `uniform` is supported for tensor and sparse grids as well as regression approaches (`collocation_points` or `collocation_ratio`, see Table 5.40 below). Each of these refinement cases makes use of the `max_iterations` and `convergence_tolerance` method independent controls (see Table 5.1); the former control limits the number of refinement iterations, and the latter control terminates refinement when the two-norm of the change in the response covariance matrix (or, in goal-oriented approaches, the two-norm of change in the statistical quantities of interest (QOI)) falls below the tolerance. The `dimension_adaptive` case can be further specified to utilize `sobol`, `decay`, or `generalized` refinement controls. The former two cases employ anisotropic tensor/sparse grids in which the anisotropic dimension preference (leading to anisotropic integrations/expansions with differing refinement levels for different random dimensions) is determined using either total Sobol' indices from variance-based decomposition (`sobol` case: high indices result in high dimension preference) or using spectral coefficient decay rates from a rate estimation technique similar to Richardson extrapolation (`decay` case: low decay rates result in high dimension preference). In these two cases as well as the `uniform` refinement case, the `quadrature_order` or `sparse_grid_level` are ramped by one on each refinement iteration until either of the two convergence controls is satisfied. For the `uniform` refinement case with regression approaches, the `expansion_order` is ramped by one on each iteration while the oversampling ratio (either defined by `collocation_ratio` or inferred from `collocation_points` based on the initial expansion) is held fixed. Finally, the `generalized` `dimension_adaptive` case is the default adaptive approach; it refers to the generalized sparse grid algorithm, a greedy approach in which candidate index sets are evaluated for their impact on the statistical QOI, the most

influential sets are selected and used to generate additional candidates, and the index set frontier of a sparse grid is evolved in an unstructured and goal-oriented manner (refer to User's Manual PCE descriptions for additional specifics).

The `variance_based_decomp` and `drop_tolerance` are also the same as those described in Nondeterministic sampling method, but since PCE outputs main, interaction, and total effects by default, the `univariate_effects` option has been added to allow suppression of the interaction effects since the output volume of these results can be prohibitive for high dimensional problems. Similar to suppression of these interactions is the covariance control, which can be selected to be `diagonal_covariance` or `full_covariance`, with the former supporting suppression of the off-diagonal covariance terms (to save compute and memory resources and reduce output volume).

As for Nondeterministic sampling method and Global reliability methods, The default behavior is to form expansions over aleatory uncertain continuous variables. To form expansions over a broader set of variables, one needs to specify `active` followed by `state`, `epistemic`, `design`, or `all` in the variables specification block. For continuous design, continuous state, and continuous epistemic uncertain variables included in the expansion, Legendre chaos bases are used to model the bounded intervals for these variables. However, these variables are not assumed to have any particular probability distribution, only that they are independent variables. Moreover, when probability integrals are evaluated, only the aleatory random variable domain is integrated, leaving behind a polynomial relationship between the statistics and the remaining design/state/epistemic variables.

Table 5.38 shows these general PCE controls and it, along with Tables 5.39, 5.40, and 5.42 to follow, provide the details of the polynomial chaos expansion specifications beyond those of Table 5.28.

To obtain the coefficients α_i of the expansion, six options are provided and the specification details for these options are provided in Tables 5.39 and 5.40:

1. multidimensional integration by a tensor-product of Gaussian quadrature rules (specified with `quadrature_order`, and, optionally, `dimension_preference`). The default rule selection is to employ `non_nested` Gauss rules including Gauss-Hermite (for normals or transformed normals), Gauss-Legendre (for uniforms or transformed uniforms), Gauss-Jacobi (for betas), Gauss-Laguerre (for exponentials), generalized Gauss-Laguerre (for gammas), and numerically-generated Gauss rules (for other distributions when using an Extended basis). For the case of `p_refinement` or the case of an explicit `nested` override, Gauss-Hermite rules are replaced with Genz-Keister nested rules and Gauss-Legendre rules are replaced with Gauss-Patterson nested rules, both of which exchange lower integrand precision for greater point reuse. By specifying a `dimension_preference`, where higher preference leads to higher order polynomial resolution, the tensor grid may be rendered anisotropic. The dimension specified to have highest preference will be set to the specified `quadrature_order` and all other dimensions will be reduced in proportion to their reduced preference; any non-integral portion is truncated. To synchronize with tensor-product integration, a tensor-product expansion is used, where the order p_i of the expansion in each dimension is selected to be half of the integrand precision available from the rule in use, rounded down. In the case of non-nested Gauss rules with integrand precision $2m_i - 1$, p_i is one less than the quadrature order m_i in each dimension (a one-dimensional expansion contains the same number of terms, $p+1$, as the number of Gauss points). The total number of terms, *N*, in a tensor-product expansion involving *n* uncertain input variables is
$$N = 1 + P = \prod_{i=1}^{n} (p_i + 1)$$
In some advanced use cases (e.g., multifidelity UQ), multiple grid resolutions can be employed; for this reason, the `quadrature_order` specification supports an array input.

2. multidimensional integration by the Smolyak sparse grid method (specified with `sparse_grid_level`

Description	Keyword	Associated Data	Status	Default
Polynomial chaos expansion method	polynomial_chaos	none	Required group	N/A
Alternate basis of orthogonal polynomials	askey | wiener	none	Optional	Extended basis of orthogonal polynomials (Askey + numerically generated)
Automated polynomial order refinement	p_refinement	uniform | dimension_adaptive	Optional group	No p-refinement
Dimension-adaptive refinement control	sobol | decay | generalized	none	Optional group	generalized
Variance-based decomposition (VBD)	variance_based_decomp	none	Optional group	VBD indices not computed/printed
Restriction of VBD indices to main/total	univariate_effects	none	Optional	Unrestricted: interaction effects included
VBD tolerance for omitting small indices	drop_tolerance	real	Optional	All VBD indices displayed
Covariance control	diagonal_covariance | full_covariance	none	Optional	diagonal_covariance for response vector > 10; else full_covariance

Table 4.38: Specification detail for polynomial chaos expansion method: general controls.

and, optionally, `dimension_preference`). The underlying one-dimensional integration rules are the same as for the tensor-product quadrature case; however, the default rule selection is `nested` for sparse grids (Genz-Keister for normals/transformed normals and Gauss-Patterson for uniforms/transformed uniforms). This default can be overridden with an explicit `non_nested` specification (resulting in Gauss-Hermite for normals/transformed normals and Gauss-Legendre for uniforms/transformed uniforms). As for tensor quadrature, the `dimension_preference` specification enables the use of anisotropic sparse grids (refer to the PCE description in the User's Manual for the anisotropic index set constraint definition). Similar to anisotropic tensor grids, the dimension with greatest preference will have resolution at the full `sparse_grid_level` and all other dimension resolutions will be reduced in proportion to their reduced preference. For PCE with either isotropic or anisotropic sparse grids, a summation of tensor-product expansions is used, where each anisotropic tensor-product quadrature rule underlying the sparse grid construction results in its own anisotropic tensor-product expansion as described in case 1. These anisotropic tensor-product expansions are summed into a sparse PCE using the standard Smolyak summation (again, refer to the User's Manual for additional details). As for `quadrature_order`, the `sparse_grid_level` specification admits an array input for enabling specification of multiple grid resolutions used by certain advanced solution methodologies.

3. multidimensional integration by Stroud cubature rules [Stroud, 1971] and extensions [Xiu, 2008], as specified with `cubature_integrand`. A total-order expansion is used, where the isotropic order p of the expansion is half of the integrand order, rounded down. The total number of terms N for an isotropic total-order expansion of order p over n variables is given by

$$N \;=\; 1 + P \;=\; 1 + \sum_{s=1}^{p} \frac{1}{s!} \prod_{r=0}^{s-1} (n+r) \;=\; \frac{(n+p)!}{n!p!}$$

   Since the maximum integrand order is currently five for normal and uniform and two for all other types, at most second- and first-order expansions, respectively, will be used. As a result, cubature is primarily useful for global sensitivity analysis, where the Sobol' indices will provide main effects and, at most, two-way interactions. In addition, the random variable set must be independent and identically distributed (*iid*), so the use of `askey` or `wiener` transformations may be required to create *iid* variable sets in the transformed space (as well as to allow usage of the higher order cubature rules for normal and uniform). Note that global sensitivity analysis often assumes uniform bounded regions, rather than precise probability distributions, so the *iid* restriction would not be problematic in that case.

4. multidimensional integration by Latin hypercube sampling (specified with `expansion_samples`). In this case, the expansion order p cannot be inferred from the numerical integration specification and it is necessary to provide an `expansion_order` to specify p for a total-order expansion.

5. linear regression (specified with either `collocation_points` or `collocation_ratio`). A total-order expansion is used and must be specified using `expansion_order` as described in the previous option. To avoid requiring the user to calculate N from n and p), the `collocation_ratio` allows for specification of a constant factor applied to N (e.g., `collocation_ratio` = `2`. produces samples = $2N$). In addition, the default linear relationship with N can be overridden using a real-valued exponent specified using `ratio_order`. In this case, the number of samples becomes cN^o where c is the `collocation_ratio` and o is the `ratio_order`. The `use_derivatives` flag informs the regression approach to include derivative matching equations (limited to gradients at present) in the least squares solutions, enabling the use of fewer collocation points for a given expansion order and dimension (number of points required becomes $\frac{cN^o}{n+1}$). When admissible, a constrained least squares approach is employed in which response values are first reproduced exactly and error in reproducing response derivatives is minimized. Two collocation grid options are supported: the default is Latin hypercube sampling ("point collocation"), and

Description	Keyword	Associated Data	Status	Default
Quadrature order for PCE coefficient estimation	`quadrature_order`	list of integers (one per grid resolution)	Required (1 of 7 selections)	N/A
Sparse grid level for PCE coefficient estimation	`sparse_grid_level`	list of integers (one per grid resolution)	Required (1 of 7 selections)	N/A
Cubature integrand order for PCE coefficient estimation	`cubature_integrand`	integer (1, 2, 3, or 5 for normal or uniform; 1 or 2 for exponential, beta, or gamma, 2 for all other distribution types)	Required (1 of 7 selections)	N/A
Dimension preference for anisotropic tensor and sparse grids	`dimension_preference`	list of reals	Optional	isotropic grids
Sparse grid growth rule restriction	`restricted` | `unrestricted`	none	Optional group	`restricted` (except for generalized sparse grids)
Integration rule point nesting for `quadrature_order` or `sparse_grid_level`	`nested` | `non_nested`	None	Optional	quadrature: `non_nested` unless automated refinement; sparse grids: `nested`

Table 4.39: Specification detail for polynomial chaos expansion method: quadrature, cubature, and sparse grid approaches.

an alternate approach of "probabilistic collocation" is also available through inclusion of the `tensor_grid` keyword. In this alternate case, the collocation grid is defined using a subset of tensor-product quadrature points: the order of the tensor-product grid is selected as one more than the expansion order in each dimension (to avoid sampling at roots of the basis polynomials) and then the tensor multi-index is uniformly sampled to generate a non-repeated subset of tensor quadrature points.

6. coefficient import from a file (specified with `expansion_import_file`). A total-order expansion is assumed and must be specified using `expansion_order`.

   If `collocation_points` or `collocation_ratio` is specified, the PCE coefficients will be determined by regression. Table 5.41 lists a set of optional regression specifications. If no regression specification is provided, appropriate defaults are defined. Specifically SVD-based least-squares will be used for solving over-determined systems and under-determined systems will be solved using LASSO. For the situation when the number of function values is smaller than the number of terms in a PCE, but the total number of samples including gradient values is greater than the number of terms, the resulting over-determined system will be solved using equality constrained least squares. Technical information on the various methods listed below can be found in the Linear regression section of the Theory Manual. Some of the regression methods (OMP, LASSO, and LARS) are able to produce a set of possible PCE coefficient vectors (see the Linear regression section in the Theory Manual). If cross validation is inactive, then only one solution, consistent with the `noise_tolerance`, will be returned. If cross validation is active, Dakota will choose between possible coefficient vectors found internally by the regression method across the set of expansion orders (1,...,expansion_order) and the set of specified noise tolerances and return the one with the lowest cross validation error indicator.

If *n* is small (e.g., two or three), then tensor-product Gaussian quadrature is quite effective and can be the preferred choice. For moderate to large *n* (e.g., five or more), tensor-product quadrature quickly becomes too expensive and the sparse grid and regression approaches are preferred. Random sampling for coefficient estimation is generally not recommended due to its slow convergence rate. For incremental studies, approaches 4 and 5 support reuse of previous samples through the `incremental_lhs` (refer to Nondeterministic sampling method for description of incremental LHS) and `reuse_samples` (refer to Global approximations for description of the "all" option of sample reuse) specifications, respectively.

In the quadrature and sparse grid cases, growth rates for nested and non-nested rules can be synchronized for consistency. For a non-nested Gauss rule used within a sparse grid, linear one-dimensional growth rules of $m = 2l + 1$ are used to enforce odd quadrature orders, where *l* is the grid level and *m* is the number of points in the rule. The precision of this Gauss rule is then $i = 2m - 1 = 4l + 1$. For nested rules, order growth with level is typically exponential; however, the default behavior is to restrict the number of points to be the lowest order rule that is available that meets the one-dimensional precision requirement implied by either a level *l* for a sparse grid ($i = 4l+1$) or an order *m* for a tensor grid ($i = 2m-1$). This behavior is known as "restricted growth" or "delayed sequences." To override this default behavior in the case of sparse grids, the `unrestricted` keyword can be used; it cannot be overridden for tensor grids using nested rules since it also provides a mapping to the available nested rule quadrature orders. An exception to the default usage of restricted growth is the `dimension_adaptive` `p_refinement` `generalized` sparse grid case described previously, since the ability to evolve the index sets of a sparse grid in an unstructured manner eliminates the motivation for restricting the exponential growth of nested rules.

Additional specifications include the level mappings described in Uncertainty Quantification Methods and the `sample_type`, `samples`, `seed`, `fixed_seed`, and `rng` specifications described in Nondeterministic sampling method, where the `sample_type` options are restricted to `random` and `lhs`. Each of these sampling specifications refer to sampling on the PCE approximation for the purposes of generating approximate statistics, which should be distinguished from simulation sampling for generating the chaos coefficients as described in options 4 and 5 above (although options 4 and 5 will share the `sample_type`, `seed`, and `rng` settings, if provided).

Description	Keyword	Associated Data	Status	Default
Expansion order	expansion_-order	list of integers (scalar input accepted for isotropic)	Required	N/A
Number simulation samples to estimate coeffs	expansion_-samples	integer	Required (1 of 7 selections)	N/A
Use incremental LHS for expansion_-samples	incremental_-lhs	none	Optional	no sample reuse in coefficient estimation
Number collocation points to estimate coeffs	collocation_-points	integer	Required (1 of 7 selections)	N/A
Collocation point oversampling ratio to estimate coeffs	collocation_-ratio	real	Required (1 of 7 selections)	N/A
Order of collocation oversampling relationship	ratio_order	real	Optional	1.
Derivative usage with collocation_-{points or ratio}	use_-derivatives	none	Optional	estimate by function values alone
Reuse points flag with collocation_-{points or ratio}	reuse_points	none	Optional	no sample reuse in coefficient estimation
Tensor grid flag with collocation_-{points or ratio}	tensor_grid	none	Optional	regression with LHS sample set ("point collocation")
PCE coeffs import file	expansion_-import_file	string	Required (1 of 7 selections)	N/A

Table 4.40: Specification detail for polynomial chaos expansion method: expansion sampling, regression, and expansion import options.

Description	Keyword	Associated Data	Status	Default
Least squares regression	least_-squares	svd | equality_-constrained	Optional	svd
L1 minimization via Basis Pursuit (BP)	basis_-pursuit	none	Optional	N/A
L1 minimization via Basis Pursuit DeNoising (BPDN)	basis_-pursuit_-denoising	none	Optional	N/A
L1 minimization via Orthogonal Matching Pursuit (OMP)	orthogonal_-matching_-pursuit	real	Optional	N/A
L1 minimization via Least Absolute Shrinkage Operator (LASSO)	least_-absolute_-shrinkage	none	Optional	N/A
L1 minimization via Least Angle Regression (LARS)	least_-angle_-regression	none	Optional	N/A
Noise tolerance(s) used in conjuction with BPDN, OMP, LARS, LASSO as an exit condition	noise_-tolerance	list of reals	Optional	1e-3 for BPDN, 0. otherwise (algorithms run until termination)
l2_penalty used for elastic net modification of LASSO	l2_penalty	real	Optional	0. (reverts to standard LASSO formulation)
Specify whether to use cross validation	cross_-validation	none	Optional	N/A

Table 4.41: Specification detail for polynomial chaos expansion method: expansion sampling, regression, and expansion import options.

The `sample_refinement` specification is similar to that of Local reliability methods, with the difference that the number of refinement samples is not under the user's control (these evaluations are approximation-based, so management of this expense is less critical). This option allows for refinement of probability and generalized reliability results using importance sampling. Table 5.42 provide the details of these sampling specifications for polynomial chaos.

Description	Keyword	Associated Data	Status	Default
Sampling type (random sampling for and on expansion)	`sample_type`	`random` | `lhs`	Optional group	`lhs`
Number of samples on PCE for generating statistics	`samples`	integer	Optional (required if sampling-based statistics are requested)	0
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple PCE runs
Random number generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
Importance sampling refinement	`sample_-refinement`	`import` | `adapt_import` | `mm_adapt_-import`	Optional group	No sampling refinement

Table 4.42: Specification detail for polynomial chaos expansion method: sampling controls.

The advanced use case of multifidelity UQ automatically becomes active if the model selected for iteration by the method specification is a multifidelity surrogate model (refer to Hierarchical approximations). In this case, an expansion will first be formed for the model discrepancy (the difference between response results if `additive correction` or the ratio of results if `multiplicative correction`), using the first `quadrature_-order` or `sparse_grid_level` value along with any specified refinement strategy. Second, an expansion will be formed for the low fidelity surrogate model, using the second `quadrature_order` or `sparse_-grid_level` value (if present; the first is reused if not present) along with any specified refinement strategy. Then the two expansions are combined (added or multiplied) into an expansion that approximates the high fidelity model, from which the final set of statistics are generated. For polynomial chaos expansions, this high fidelity expansion can differ significantly in form from the low fidelity and discrepancy expansions, particularly in the `multiplicative` case where it is expanded to include all of the basis products.

4.7.1.7 Stochastic collocation method

The stochastic collocation (SC) method is very similar to the PCE method described above, with the key difference that the orthogonal polynomial basis functions are replaced with interpolation polynomial bases. The interpolation polynomials may be either local or global and either value-based or gradient-enhanced. In the local case, valued-based are piecewise linear splines and gradient-enhanced are piecewise cubic splines, and in the global case, valued-based are Lagrange interpolants and gradient-enhanced are Hermite interpolants. A value-based expansion

takes the form

$$R = \sum_{i=1}^{N_p} r_i L_i(\xi)$$

where N_p is the total number of collocation points, r_i is a response value at the ith collocation point, L_i is the ith multidimensional interpolation polynomial, and ξ is a vector of standardized random variables. The ith interpolation polynomial assumes the value of 1 at the ith collocation point and 0 at all other collocation points, involving either a global Lagrange polynomial basis or local piecewise splines. It is easy to see that the approximation reproduces the response values at the collocation points and interpolates between these values at other points. A gradient-enhanced expansion (selected via the `use_derivatives` keyword) involves both type 1 and type 2 basis functions as follows:

$$R = \sum_{i=1}^{N_p} [r_i H_i^{(1)}(\xi) + \sum_{j=1}^{n} \frac{dr_i}{d\xi_j} H_{ij}^{(2)}(\xi)]$$

where the ith type 1 interpolant produces 1 for the value at the ith collocation point, 0 for values at all other collocation points, and 0 for derivatives (when differentiated) at all collocation points, and the ij^{th} type 2 interpolant produces 0 for values at all collocation points, 1 for the jth derivative component at the ith collocation point, and 0 for the jth derivative component at all other collocation points. Again, this expansion reproduces the response values at each of the collocation points, and when differentiated, also reproduces each component of the gradient at each of the collocation points. Since this technique includes the derivative interpolation explicitly, it eliminates issues with matrix ill-conditioning that can occur in the gradient-enhanced PCE approach based on regression. However, the calculation of high-order global polynomials with the desired interpolation properties can be similarly numerically challenging such that the use of local cubic splines is recommended due to numerical stability.

Thus, in PCE, one forms coefficients for known orthogonal polynomial basis functions, whereas SC forms multidimensional interpolation functions for known coefficients. Dakota provides access to SC methods through the **NonDStochCollocation** class. Refer to the Uncertainty Quantification Capabilities chapter of the Users Manual [Adams et al., 2010] for additional information on the SC algorithm.

As for Polynomial chaos expansion method, the orthogonal polynomials used in defining the Gauss points that make up the interpolation grid are governed by the Wiener, Askey, or Extended options. The Wiener option uses interpolation points from Gauss-Hermite (non-nested) or Genz-Keister (nested) integration rules for all random variables and employs the same nonlinear variable transformation as the local and global reliability methods (and therefore has the same variable support). The Askey option, however, employs interpolation points from Gauss-Hermite (Genz-Keister if nested), Gauss-Legendre (Gauss-Patterson if nested), Gauss-Laguerre, Gauss-Jacobi, and generalized Gauss-Laguerre quadrature. The Extended option avoids the use of any nonlinear variable transformations by augmenting the Askey approach with Gauss points from numerically-generated orthogonal polynomials for non-Askey probability density functions. As for PCE, the Wiener/Askey/Extended selection defaults to Extended, can be overridden by the user using the keywords `askey` or `wiener`, and automatically falls back from Extended/Askey to Wiener on a per variable basis as needed to support prescribed correlations. Unlike PCE, however, SC also supports the option of `piecewise` local basis functions. These are piecewise linear splines, or in the case of gradient-enhanced interpolation via the `use_derivatives` specification, piecewise cubic Hermite splines. Both of these basis selections provide local support only over the range from the interpolated point to its nearest 1D neighbors (within a tensor grid or within each of the tensor grids underlying a sparse grid), which exchanges the fast convergence of global bases for smooth functions for robustness in the representation of nonsmooth response functions (that can induce Gibbs oscillations when using high-order global basis functions).

When local basis functions are used, the usage of nonequidistant collocation points (e.g., the Gauss point selections described above) is not well motivated, so equidistant Newton-Cotes points are employed in this case, and all random variable types are transformed to standard uniform probability space (refer to Tables 5.29, 5.30, and 5.31 for variable types for which this transformation is supported). The global gradient-enhanced interpolants (Hermite interpolation polynomials) are also restricted to uniform or transformed uniform random variables (due to the need to compute collocation weights by integration of the basis polynomials) and share the variable support shown in these tables for Piecewise SE. Due to numerical instability in these high-order basis polynomials, they are deactivated by default but can be activated by developers using a compile-time switch.

Another distinguishing characteristic of stochastic collocation relative to polynomial chaos is the ability to reformulate the interpolation problem from a `nodal` interpolation approach into a `hierarchical` formulation in which each new level of interpolation defines a set of incremental refinements (known as hierarchical surpluses) layered on top of the interpolants from previous levels. This formulation lends itself naturally to uniform or adaptive refinement strategies, since the hierarchical surpluses can be interpreted as error estimates for the interpolant. Either global or local/piecewise interpolants in either value-based or gradient-enhanced approaches can be formulated using `hierarchical` interpolation. The primary restriction for the hierarchical case is that it currently requires a sparse grid approach using nested quadrature rules (Genz-Keister, Gauss-Patterson, or Newton-Cotes for standard normals and standard uniforms in a transformed space: Askey, Wiener, or Piecewise settings may be required), although this restriction can be relaxed in the future. A selection of `hierarchical` interpolation will provide greater precision in the increments to mean, standard deviation, covariance, and reliability-based level mappings induced by a grid change within uniform or goal-oriented adaptive refinement approaches (see following section).

Automated expansion refinement can be selected as either `p_refinement` or `h_refinement`, and either refinement specification can be either `uniform` or `dimension_adaptive`. The `dimension_adaptive` case can be further specified as either `sobol` or `generalized` (`decay` not supported). Each of these automated refinement approaches makes use of the `max_iterations` and `convergence_tolerance` iteration controls (see Table 5.1). The `h_refinement` specification involves use of the same piecewise interpolants (linear or cubic Hermite splines) described above for the `piecewise` specification option (it is not necessary to redundantly specify `piecewise` in the case of `h_refinement`). In future releases, the `hierarchical` interpolation approach will enable local refinement in addition to the current `uniform` and `dimension_adaptive` options.

The `variance_based_decomp` and covariance controls are identical to that described in Polynomial chaos expansion method. As a default, the interpolation will be performed over continuous aleatory uncertain variables. To expand the dimensionality of the interpolation to include continuous design, state, and epistemic uncertain variables, it is necessary to specify which variables are active in the variables specification block. Interpolation points for these dimensions are based on Gauss-Legendre rules if non-nested, Gauss-Patterson rules if nested, and Newton-Cotes points in the case of piecewise bases. Again, when probability integrals are evaluated, only the aleatory random variable domain is integrated, leaving behind a polynomial relationship between the statistics and the remaining design/state/epistemic variables.

Table 5.43 shows these general SC controls, and it along with Table 5.44 to follow provide the details of the SC specifications beyond those of Tables 5.28 and 5.42.

To form the multidimensional interpolants L_i of the expansion, two options are provided and the specification details for these options are provided in Table 5.44.

1. interpolation on a tensor-product of Gaussian quadrature points (specified with `quadrature_order` and, optionally, `dimension_preference` for anisotropic tensor grids). As for PCE, non-nested Gauss rules are employed by default, although the presence of `p_refinement` or `h_refinement` will result in default usage of nested rules for normal or uniform variables after any variable transformations have been

Description	Keyword	Associated Data	Status	Default
Stochastic collocation method	`stoch_-collocation`	none	Required group	N/A
Alternate basis selections for defining collocation points/weights	`askey` | `wiener` | `piecewise`	none	Optional	Gauss points/weights from extended set of global orthogonal polynomials (Askey + numerically generated)
Basis formulation	`nodal` | `hierarchical`	none	Optional	`nodal`
Derivative enhancement flag	`use_-derivatives`	none	Optional	interpolation based on function values alone
Automated expansion refinement	`p_refinement` | `h_refinement`	`uniform` | `dimension_-adaptive`	Optional group	No refinement
Dimension-adaptive refinement control	`sobol` | `generalized`	none	Optional group	`generalized`
Variance-based decomposition (VBD)	`variance_-based_decomp`	none	Optional group	VBD indices not computed/printed
Restriction of VBD indices to main/total	`univariate_-effects`	none	Optional	Unrestricted: interaction effects included
VBD tolerance for omitting small indices	`drop_-tolerance`	real	Optional	All VBD indices displayed
Covariance control	`diagonal_-covariance` | `full_-covariance`	none	Optional	`diagonal_-covariance` for response vector > 10; else `full_-covariance`

Table 4.43: Specification detail for stochastic collocation method: general controls.

applied (both defaults can be overridden using explicit `nested` or `non_nested` specifications).

2. interpolation on a Smolyak sparse grid (specified with `sparse_grid_level` and, optionally, `dimension_preference` for anisotropic sparse grids) defined from Gaussian rules. As for sparse PCE, nested rules are employed unless overridden with the `non_nested` option, and the growth rules are restricted unless overridden by the `unrestricted` keyword.

Description	Keyword	Associated Data	Status	Default
Quadrature order for collocation points	`quadrature_order`	list of integers (one per grid resolution)	Required (1 of 2 selections)	N/A
Sparse grid level for collocation points	`sparse_grid_level`	list of integers (one per grid resolution)	Required (1 of 2 selections)	N/A
Dimension preference for anisotropic tensor and sparse grids	`dimension_preference`	list of reals	Optional	isotropic grids
Sparse grid growth rule restriction	`restricted` | `unrestricted`	none	Optional group	`restricted` (except for generalized sparse grids)
Integration rule point nesting	`nested` | `non_nested`	None	Optional	quadrature: `non_nested` unless automated refinement; sparse grids: `nested`

Table 4.44: Specification detail for stochastic collocation method: grid controls.

If n is small, then tensor-product Gaussian quadrature is again the preferred choice. For larger n, tensor-product quadrature quickly becomes too expensive and the sparse grid approach is preferred. For self-consistency in growth rates, nested rules employ restricted exponential growth (with the exception of the `dimension_adaptive` `p_refinement` `generalized` case) for consistency with the linear growth used for non-nested Gauss rules (integrand precision $i = 4l + 1$ for sparse grid level l and $i = 2m - 1$ for tensor grid order m).

Additional specifications include the level mappings described in Uncertainty Quantification Methods, the `sample_type`, `samples`, `seed`, `fixed_seed`, and `rng` specifications described in Nondeterministic sampling method, and the `sample_refinement` specification described in Polynomial chaos expansion method. Each of the sampling specifications is identical to those for PCE shown in Table 5.42 and refer to sampling on the interpolant for the purposes of generating approximate statistics, which should not be confused with collocation evaluations used for forming the interpolant. Finally, the process for multifidelity UQ is identical to that described in Polynomial chaos expansion method, with the exception that the high fidelity expansion generated from combining the low fidelity and discrepancy expansions retains the polynomial form of the low fidelity expansion (only the coefficients are updated).

4.7.2 Epistemic Uncertainty Quantification Methods

Epistemic uncertainty is also referred to as subjective uncertainty, reducible uncertainty, model form uncertainty, or uncertainty due to lack of knowledge. Examples of epistemic uncertainty are little or no experimental data

for an unknown physical parameter, or the existence of complex physics or behavior that is not included in the simulation model of a system. Epistemic uncertainty can be modeled probabilistically but is often modeled using non-probabilistic approaches such as interval propagation, evidence theory, possibility theory, information gap theory, etc. In Dakota, epistemic uncertainty analysis is performed using interval analysis or Dempster-Shafer theory of evidence. Epistemic (or mixed aleatory-epistemic) uncertainty may also be propagated through the use of the Nondeterministic sampling method, although in this case, the output statistics are limited to response intervals (any epistemic component suppresses all probabilistic results). Mixed uncertainty can also be addressed through use of nested UQ (refer to the Users Manual [Adams et al., 2010] for NestedModel discussion and examples); in this case, epistemic and aleatory analyses can be segregated and intervals on probabilistic results can be reported. A subtle distinction exists between `sampling` for epistemic intervals and the `lhs` option of `global_interval_est`: the former allows mixed aleatory-epistemic uncertainty within a single level, whereas the latter supports only epistemic variables and relies on nesting to address mixed uncertainty. In each of these cases, the Continuous Interval Uncertain Variable specification is used to describe the epistemic uncertainty using either simple intervals or basic probability assignments. Note that for mixed UQ problems with both aleatory and epistemic variables, if the user defines the epistemic variables as intervals and aleatory variables as probability distribution types, the method `sampling` (in a simple, single-level study) will result in intervals only on the output. Although the aleatory variables will be sampled according to their distributions, the output will only be reported as an interval given the presence of interval variables. There is also the option to perform nested sampling, where one separates the epistemic and aleatory uncertain variables, samples over epistemic variables in the outer loop and then samples the aleatory variables in the inner llop, resulting in intervals on statistics. The calculation of intervals on statistics can also be performed by using nested approaches with interval estimation or evidence methods in the outer loop and aleatory UQ methods on the inner loop such as stochastic expansion or reliability methods. More detail about these "intervals on statistics" approaches can be found in [Eldred and Swiler, 2009] and [Eldred et al., 2011].

4.7.2.1 Local Interval Estimation

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

We have the capability to perform interval analysis using either local methods (`local_interval_est`) or global methods (`global_interval_est`). If the problem is amenable to local optimization methods (e.g. can provide derivatives or use finite difference method to calculate derivatives), then one can use local methods to calculate these bounds. `local_interval_est` allows the user to specify either `sqp` which is sequential quadratic programming, or `nip` which is a nonlinear interior point method. Table 5.45 provides the specification for the local interval method.

Description	Keyword	Associated Data	Status	Default
Nondeterministic interval estimation	`local_interval_est`	none	Required group	N/A
Estimation method	`sqp` | `nip`	none	Required group	N/A

Table 4.45: Specification detail for local interval estimation used in epistemic uncertainty

4.7.2.2 Global Interval Estimation

As mentioned above, when performing interval analysis, one assumes that nothing is known about an epistemic uncertain variable except that its value lies somewhere within an interval. The goal of uncertainty quantification in this context is to determine the resulting bounds on the output (defining the output interval) given interval bounds on the inputs.

In the global approach, one uses either a global optimization method or a sampling method to assess the bounds. `global_interval_est` allows the user to specify several approaches to calculate interval bounds on the output responses. `lhs` performs Latin Hypercube Sampling and takes the minimum and maximum of the samples as the bounds (no optimization is performed). In the `ego` approach, the efficient global optimization (EGO) method is used to calculate bounds (see the EGO method on this page for more explanation). By default, the Surfpack GP (Kriging) model is used, but the Dakota implementation may be selected instead. If `use_derivatives` is specified the GP model will be built using available derivative data (Surfpack GP only). If using `sbo`, a surrogate-based optimization method will be used to find the interval bounds. The surrogate used in `sbo` is a Gaussian process surrogate. The main difference between `ego` and the `sbo` approach is the objective function that is used in the optimization. `ego` relies on an expected improvement function, while in `sbo`, the optimization proceeds using an evolutionary algorithm (`coliny_ea` described above) on the Gaussian process surrogate: it is a standard surrogate-based optimization. Also note that the `sbo` option can support optimization over discrete variables (the discrete variables are relaxed) while `ego` cannot. Finally, there is the `ea` approach. In this approach, the evolutionary algorithm from Coliny (`coliny_ea`) is used to perform the interval optimization with no surrogate model involved. Again, this option of `ea` can support interval optimization over discrete variables. When using `lhs`, `ego`, or `sbo`, one can specify a seed for the number of LHS samples, the random number generator, and the number of samples. The `ea` option allows the seed to be specified. Table 5.46 provides the specification for the global interval methods.

Description	Keyword	Associated Data	Status	Default
Nondeterministic interval estimation	`global_interval_est`	none	Required group	N/A
Estimation method	`lhs` | `ego` | `sbo` | `ea`	none	Required group	N/A
EGO GP selection	`gaussian_process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_derivatives`	none	Optional	Use function values only
Random seed generator	`rng`	`mt19937` | `rnum2`	Optional	Mersenne twister (`mt19937`)
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples	`samples`	integer	Optional	10,000 for LHS, approximately numVars^2 for EGO

Table 4.46: Specification detail for global interval estimation used in epistemic uncertainty

4.7.2.3 Local Evidence Theory (Dempster-Shafer) Methods

The above section discussed a pure interval approach. This section discusses Dempster-Shafer evidence theory. In this approach, one does not assign a probability distribution to each uncertain input variable. Rather, one divides each uncertain input variable into one or more intervals. The input parameters are only known to occur within intervals: nothing more is assumed. Each interval is defined by its upper and lower bounds, and a Basic Probability Assignment (BPA) associated with that interval. The BPA represents a probability of that uncertain variable being located within that interval. The intervals and BPAs are used to construct uncertainty measures on the outputs called "belief" and "plausibility." Belief represents the smallest possible probability that is consistent with the evidence, while plausibility represents the largest possible probability that is consistent with the evidence. For more information about the Dempster-Shafer theory of evidence, see Oberkampf and Helton, 2003 and Helton and Oberkampf, 2004.

Similar to the interval approaches, one may use global or local methods to determine plausbility and belief measures for the outputs. Note that to calculate the plausibility and belief cumulative distribution functions, one has to look at all combinations of intervals for the uncertain variables. Within each interval cell combination, the minimum and maximum value of the objective function determine the belief and plausibility, respectively. In terms of implementation, global methods use LHS sampling or global optimization to calculate the minimum and maximum values of the objective function within each interval cell, while local methods use gradient-based optimization methods to calculate these minima and maxima.

Finally, note that the nondeterministic general settings apply to the interval and evidence methods, but one needs to be careful about the interpretation and translate probabilistic measures to epistemic ones. For example, if the user specifies distribution of type complementary, a complementary plausibility and belief function will be generated for the evidence methods (as opposed to a complementary distribution function in the `sampling` case). If the user specifies a set of responses levels, both the belief and plausibility will be calculated for each response level. Likewise, if the user specifies a probability level, the probability level will be interpreted both as a belief and plausibility, and response levels corresponding to the belief and plausibility levels will be calculated. Finally, if generalized reliability levels are specified, either as inputs (`gen_reliability_levels`) or outputs (`response_levels` with `compute gen_reliabilities`), then these are directly converted to/from probability levels and the same probability-based mappings described above are performed.

Table 5.47 provides the specification for the `local_evidence` method. Note that two local optimization methods are available: `sqp` (sequential quadratic programming or `nip` (nonlinear interior point method).

Description	Keyword	Associated Data	Status	Default
Nondeterministic local evidence method	`local_-evidence`	none	Required group	N/A
Estimation method	`sqp` | `nip`	none	Required group	N/A

Table 4.47: Specification detail for local evidence theory method for epistemic uncertainty

4.7.2.4 Global Evidence Theory (Dempster-Shafer) Methods

Evidence theory has been explained above in the Local Evidence Theory section. The basic idea is that one specifies an "evidence structure" on uncertain inputs and propagates that to obtain belief and plausibility functions on the response functions. The inputs are defined by sets of intervals and Basic Probability Assignments (BPAs). Evidence propagation is computationally expensive, since the minimum and maximum function value must be calculated for each "interval cell combination." These bounds are aggregated into belief and plausibility.

Table 5.48 provides the specification for the `global_evidence` method. `global_evidence` allows the user to specify several approaches for calculating the belief and plausibility functions: `lhs`, `ego`, `sbo`, and `ea`. `lhs` performs Latin Hypercube Sampling and takes the minimum and maximum of the samples as the bounds per "interval cell combination." In the case of `ego`, the efficient global optimization (EGO) method is used to calculate bounds (see the EGO method on this page for more explanation). By default, the Surfpack GP (Kriging) model is used, but the Dakota implementation may be selected instead. If `use_derivatives` is specified the GP model will be built using available derivative data (Surfpack GP only). If using `sbo`, a surrogate-based optimization method will be used to find the interval cell bounds. The surrogate employed in `sbo` is a Gaussian process surrogate. However, the main difference between `ego` and the `sbo` approach is the objective function being optimized. `ego` relies on an expected improvement function, while in `sbo`, the optimization proceeds using an evolutionary algorithm (`coliny_ea` described above) on the Gaussian process surrogate: it is a standard surrogate-based optimization. Also note that the `sbo` option can support optimization over discrete variables (the discrete variables are relaxed) while `ego` cannot. Finally, there is the `ea` approach. In this approach, the evolutionary algorithm from Coliny is used to perform the interval optimization with no surrogate model involved. Again, this option of `ea` can support interval optimization over discrete variables. When using `lhs`, `ego`, or `sbo`, one can specify the seed for the number of LHS samples, the random number generator, and the number of samples. `ea` will use the seed specification also.

Note that to calculate the plausibility and belief cumulative distribution functions, one has to look at all combinations of intervals for the uncertain variables. In terms of implementation, if one is using LHS sampling as outlined above, this method creates a large sample over the response surface, then examines each cell to determine the minimum and maximum sample values within each cell. To do this, one needs to set the number of samples relatively high: the default is 10,000 and we recommend at least that number. If the model you are running is a simulation that is computationally quite expensive, we recommend that you set up a surrogate model within the Dakota input file so that `global_evidence` performs its sampling and calculations on the surrogate and not on the original model. If one uses optimization methods instead to find the minimum and maximum sample values within each cell, this can also be computationally expensive.

Description	Keyword	Associated Data	Status	Default
Nondeterministic global evidence method	`global_evidence`	none	Required group	N/A
Estimation method	`lhs` | `ego` | `sbo` | `ea`	none	Required group	N/A
GP selection	`gaussian_process`	`surfpack` | `dakota`	Optional	Surfpack Gaussian process
Derivative usage	`use_derivatives`	none	Optional	Use function values only
Random seed generator	`rng`	`mt19937` or `rnum2`	Optional	Mersenne twister (`mt19937`)
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples	`samples`	integer	Optional	10,000 for LHS, approximately numVars^2 for EGO

Table 4.48: Specification detail for global evidence theory method for epistemic uncertainty

4.8 Nondeterministic Calibration Methods

This iterator branch is an important focus for upcoming releases. Capabilities for inferring statistical models from data are important for a variety of current and emerging mission areas.

4.8.1 Bayesian Calibration Methods

Currently, we are in the process of incorporating Bayesian calibration methods in Dakota. These methods take prior information on parameter values (in the form of prior distributions) and observational data (e.g. from experiments) and produce posterior distributions on the parameter values. When the computational simulation is then executed with samples from the posterior parameter distributions, the results that are produced are consistent with ("agree with") the experimental data. In the case of calibrating parameters from a computational simulation model, we require a "likelihood function" that specifies the likelihood of observing a particular observation given the model and its associated parameterization. We assume a Gaussian likelihood function currently. The algorithms that produce the posterior distributions on model parameters are Monte Carlo Markov Chain (MCMC) sampling algorithms. MCMC methods require many samples, often tens of thousands, so in the case of model calibration, often emulators of the computational simulation are used. For more details on the algorithms underlying the methods, see the Dakota User's manual.

We have two Bayesian calibration methods under development in Dakota: one called QUESO and one called GPMSA. They are specified with the `bayes_calibration queso` or `bayes_calibration gpmsa`, respectively. The QUESO method is using components from the QUESO library (Quantification of Uncertainty for Estimation, Simulation, and Optimization) developed at The University of Texas at Austin. We are using a DRAM (Delayed Rejected Adaptive Metropolis) algorithm for the MCMC sampling from the QUESO library. GPMSA (Gaussian Process Models for Simulation Analysis) is a code that has been developed at Los Alamos National Laboratory. It uses Gaussian process models as part of constructing an emulator for the expensive computational simulation. GPMSA also has extensive features for calibration, such as the capability to include a "model discrepancy" term and the capability to model functional data such as time series data.

For the QUESO method, one can use an emulator in the MCMC sampling. This will greatly improve the speed, since the Monte Carlo Markov Chain will generate thousands of samples on the emulator instead of the real simulation code. However, in the case of fast running evaluations, we recommend the use of no emulator. An emulator may be specified with the keyword `emulator`, followed by a `gaussian_process` emulator, a `pce` emulator (polynomial chaos expansion), or a `sc` emulator (stochastic collocation). For the `gaussian_process` emulator, the user must specify whether to use the `surfpack` or `dakota` version of the Gaussian process. The user can define the number of samples `emulator_samples` from which the emulator should be built. It is also possible to build the Gaussian process from points read in from the `points_file`. For `pce` or `sc`, the user can define a `sparse_grid_level`.

In terms of the MCMC sampling, one can specify the following for the QUESO method: With the `metropolis` type, we have the options `hastings` for a standard Metropolis-Hastings algorithm, or `adaptive` for the adaptive Metropolis in which the covariance of the proposal density is updated adaptively. For the delayed rejection part of the DRAM algorithm, one specifies `rejection`, followed by `standard` (no delayed rejection) or `delayed`. Finally, the user has two scale factors which help control the scaling involved in the problem. The `likelihood_scale` is a number which multiplies the likelihood. This is useful for situations with very small likelihoods (e.g. the model is either very far away from the data or there is a lot of data so the likelihood function involves multiplying many likelihoods together and becomes very small). The second factor is a `proposal_covariance_scale` which scales the proposal covariance. This may be useful when the input variables being calibrated are of different magnitudes: one may want to take a larger step in a direction with a larger magnitude, for example. Finally, we offer the option to calibrate the sigma terms with the `calibrate_sigma` flag. The

sigma terms refer to the error associated with the Gaussian process: sigma is used in the likelihood calculation. If experimental measurement error is available to inform sigma, that is very useful, but often measurement uncertainty is not available. Note that if `calibrate_sigma` is specified, a separate sigma term will be calibrated for each calibration term. Thus, if there are 50 calibration terms (e.g. experimental points against which we are trying to match the model), 50 sigma values will be added to the calibration process. Calibration of the sigma values is turned off by default: only the design parameters are calibrated in default mode.

For the GPMSA method, one can define the number of samples which will be used in construction of the emulator, `emulator_samples`. The emulator involves Gaussian processes in GPMSA, so the user does not specify anything about emulator type. At this point, the only controls active for GPMSA are `emulator_samples`, `seed` and `rng`, and `samples` (the number of MCMC samples).

As mentioned above, the Bayesian capability in Dakota currently relies on the QUESO library developed by The University of Texas at Austin. This integrated capability is still in prototype form and available to close collaborators of the Dakota team. If you are interested in this capability, contact the Dakota developers at `dakota-developers@development.sandia.gov`.

Table 5.49 provides the specification details for the `bayes_calibration` methods.

Description	Keyword	Associated Data	Status	Default
Bayesian calibration	`bayes_calibration`	`queso` | `gpmsa`	Required group	N/A
Emulator type	`emulator` `gaussian_process` (`surfpack` | `dakota`) | `pce` | `sc`	`emulator_samples` (for gaussian process), `sparse_grid_level` (for pce or sc)	Optional	N/A
Metropolis type for the MCMC algorithm	`metropolis`	`hastings` | `adaptive`	Optional	hastings
Rejection type for the MCMC algorithms	`rejection`	`standard` | `delayed`	Optional	standard
Random seed generator	`rng`	`mt19937` or `rnum2`	Optional	Mersenne twister (`mt19937`)
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples taken in the MCMC sampling	`samples`	integer	Optional	10000
Likelihood scale factor	`likelihood_scale`	real	Optional	none
Proposal covariance scaling	`proposal_covariance_scale`	list of reals	Optional	none
Calibrate sigma flag	`calibrate_sigma`	none	Optional	The sigma terms are not calibrated

Table 4.49: Specification detail for the Bayesian calibration methods

4.9 Solution Verification Methods

Solution verification procedures estimate the order of convergence of the simulation response data during the course of a refinement study. This branch of methods is new and currently only contains one algorithm: Richardson extrapolation.

4.9.1 Richardson Extrapolation

This method utilizes state variables as active variables (continuous state only at this time; discrete state to follow later) for parameterizing the refinement of the discretizations, and employs the `max_iterations` and `convergence_tolerance` method independent controls to manage the iterative procedures. The refinement path is determined from the `initial_state` of the `continuous_state` variables specification (refer to State Variables) in combination with the `refinement_rate`, where each of the state variables is treated as an independent refinement factor and each of the initial state values is repeatedly divided by the refinement rate value to define new discretization states. Three algorithm options are currently provided:

1. The `estimate_order` option is the simplest option. For each of the refinement factors, it evaluates three points along the refinement path and uses these results to perform an estimate of the convergence order for each response function.

1. The `converge_order` option is initialized using the `estimate_order` aproach, and additional refinements are performed along the refinement path until the convergence order estimates converge (two-norm of the change in response orders is less than the convergence tolerance).

1. The `converge_qoi` option is similar to the `converge_order` option, except that the convergence criterion is that the two-norm of the response discretization errors (computed from extrapolation) must be less than the convergence tolerance.

In each of these cases, convergence order for a response quantity of interest (QOI) is estimated from

$$p = ln(\frac{QOI_3 - QOI_2}{QOI_2 - QOI_1})/ln(r)$$

where r is the uniform refinement rate specified by `refinement_rate`. Table 5.50 provides the specification details for the `richardson_extrap` method.

Description	Keyword	Associated Data	Status	Default
Richardson extrapolation	`richardson_extrap`	none	Required group	N/A
Order estimation approach	`estimate_order` | `converge_order` | `converge_qoi`	none	Required	N/A
Refinement rate	`refinement_rate`	real	Optional	2.

Table 4.50: Specification detail for Richardson extrapolation method for solution verification

4.10 Design of Computer Experiments Methods

Design and Analysis of Computer Experiments (DACE) methods compute response data sets at a selection of points in the parameter space. Three libraries are provided for performing these studies: DDACE, FSUDace, and PSUADE. The design of experiments methods do not currently make use of any of the method independent controls.

4.10.1 DDACE

The Distributed Design and Analysis of Computer Experiments (DDACE) library provides the following DACE techniques: grid sampling (`grid`), pure random sampling (`random`), orthogonal array sampling (`oas`), latin hypercube sampling (`lhs`), orthogonal array latin hypercube sampling (`oa_lhs`), Box-Behnken (`box_behnken`), and central composite design (`central_composite`). It is worth noting that there is some overlap in sampling techniques with those available from the nondeterministic branch. The current distinction is that the nondeterministic branch methods are designed to sample within a variety of probability distributions for uncertain variables, whereas the design of experiments methods treat all variables as having uniform distributions. As such, the design of experiments methods are well-suited for performing parametric studies and for generating data sets used in building global approximations (see Global approximations), but are not currently suited for assessing the effect of uncertainties. If a design of experiments over both design/state variables (treated as uniform) and uncertain variables (with probability distributions) is desired, then `sampling` can support this with `active all` specified in the **Variables** specification block. Dakota provides access to the DDACE library through the **DDACEDesignCompExp** class.

In terms of method dependent controls, the specification structure is straightforward. First, there is a set of design of experiments algorithm selections separated by logical OR's (`grid` or `random` or `oas` or `lhs` or `oa_lhs` or `box_behnken` or `central_composite`). Second, there are optional specifications for the random seed to use in generating the sample set (`seed`), for fixing the seed (`fixed_seed`) among multiple sample sets (see Nondeterministic sampling method for discussion), for the number of samples to perform (`samples`), and for the number of symbols to use (`symbols`). The `seed` control is used to make sample sets repeatable, and the `symbols` control is related to the number of replications in the sample set (a larger number of symbols equates to more stratification and fewer replications). The `main_effects` control prints Analysis-of-Variance main effects results (e.g. ANOVA tables with p-values per variable). The `main_effects` control is only operational with the orthogonal arrays or Latin Hypercube designs, not for Box Behnken or Central Composite designs. The `quality_metrics` control is available for the DDACE library. This control turns on calculation of volumetric quality measures which measure the uniformity of the point samples. More details on the quality measures are given under the description of the FSU sampling methods. The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in contributing to the output variance. More details on variance based decomposition are given in Nondeterministic sampling method. Design of experiments specification detail is given in Table 5.51.

4.10.2 FSUDace

The Florida State University Design and Analysis of Computer Experiments (FSUDace) library provides the following DACE techniques: quasi-Monte Carlo sampling (`fsu_quasi_mc`) based on the Halton sequence (`halton`) or the Hammersley sequence (`hammersley`), and Centroidal Voronoi Tessellation (`fsu_cvt`). All three methods generate sets of uniform random variables on the interval [0,1]. If the user specifies lower and upper bounds for a variable, the [0,1] samples are mapped to the [lower, upper] interval. The quasi-Monte Carlo and CVT methods are designed with the goal of low discrepancy. Discrepancy refers to the nonuniformity of

Description	Keyword	Associated Data	Status	Default
Design of experiments method	`dace`	none	Required group	N/A
dace algorithm selection	`grid` | `random` | `oas` | `lhs` | `oa_lhs` | `box_behnken` | `central_-composite`	none	Required	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple DACE runs
Number of samples	`samples`	integer	Optional	minimum required
Number of symbols	`symbols`	integer	Optional	default for sampling algorithm
Main effects	`main_effects`	none	Optional	No main_effects
Quality metrics	`quality_-metrics`	none	Optional	No quality_metrics
Variance based decomposition	`variance_-based_decomp`	none	Optional	No variance_-based_decomp

Table 4.51: Specification detail for design of experiments methods

the sample points within the hypercube. Discrepancy is defined as the difference between the actual number and the expected number of points one would expect in a particular set B (such as a hyper-rectangle within the unit hypercube), maximized over all such sets. Low discrepancy sequences tend to cover the unit hypercube reasonably uniformly. Quasi-Monte Carlo methods produce low discrepancy sequences, especially if one is interested in the uniformity of projections of the point sets onto lower dimensional faces of the hypercube (usually 1-D: how well do the marginal distributions approximate a uniform?) CVT does very well volumetrically: it spaces the points fairly equally throughout the space, so that the points cover the region and are isotropically distributed with no directional bias in the point placement. There are various measures of volumetric uniformity which take into account the distances between pairs of points, regularity measures, etc. Note that CVT does not produce low-discrepancy sequences in lower dimensions, however: the lower-dimension (such as 1-D) projections of CVT can have high discrepancy.

The quasi-Monte Carlo sequences of Halton and Hammersley are deterministic sequences determined by a set of prime bases. Generally, we recommend that the user leave the default setting for the bases, which are the lowest primes. Thus, if one wants to generate a sample set for 3 random variables, the default bases used are 2, 3, and 5 in the Halton sequence. To give an example of how these sequences look, the Halton sequence in base 2 starts with points 0.5, 0.25, 0.75, 0.125, 0.625, etc. The first few points in a Halton base 3 sequence are 0.33333, 0.66667, 0.11111, 0.44444, 0.77777, etc. Notice that the Halton sequence tends to alternate back and forth, generating a point closer to zero then a point closer to one. An individual sequence is based on a radix inverse function defined on a prime base. The prime base determines how quickly the [0,1] interval is filled in. Generally, the lowest primes are recommended.

The Hammersley sequence is the same as the Halton sequence, except the values for the first random variable are equal to 1/N, where N is the number of samples. Thus, if one wants to generate a sample set of 100 samples for 3 random variables, the first random variable has values 1/100, 2/100, 3/100, etc. and the second and third variables are generated according to a Halton sequence with bases 2 and 3, respectively. For more information about these sequences, see [Halton, 1960, Halton and Smith, 1964, and Kocis and Whiten, 1997].

The specification for specifying quasi-Monte Carlo (`fsu_quasi_mc`) is given below in Table 5.52. The user must specify if the sequence is (`halton`) or (`hammersley`). The user must also specify the number of samples to generate for each variable (`samples`). Then, there are three optional lists the user may specify. The first list determines where in the sequence the user wants to start. For example, for the Halton sequence in base 2, if the user specifies sequence_start = 2, the sequence would not include 0.5 and 0.25, but instead would start at 0.75. The default `sequence_start` is a vector with 0 for each variable, specifying that each sequence start with the first term. The `sequence_leap` control is similar but controls the "leaping" of terms in the sequence. The default is 1 for each variable, meaning that each term in the sequence be returned. If the user specifies a sequence_leap of 2 for a variable, the points returned would be every other term from the QMC sequence. The advantage to using a leap value greater than one is mainly for high-dimensional sets of random deviates. In this case, setting a leap value to the next prime number larger than the largest prime base can help maintain uniformity when generating sample sets for high dimensions. For more information about the efficacy of leaped Halton sequences, see [Robinson and Atcitty, 1999]. The final specification for the QMC sequences is the prime base. It is recommended that the user not specify this and use the default values. For the Halton sequence, the default bases are primes in increasing order, starting with 2, 3, 5, etc. For the Hammersley sequence, the user specifies (s-1) primes if one is generating an s-dimensional set of random variables.

The `fixed_sequence` control is similar to `fixed_seed` for other sampling methods. If `fixed_sequence` is specified, the user will get the same sequence (meaning the same set of samples) for subsequent calls of the QMC sampling method (for example, this might be used in a surrogate based optimization method or a parameter study where one wants to fix the uncertain variables). The `latinize` command takes the QMC sequence and "latinizes" it, meaning that each original sample is moved so that it falls into one strata or bin in each dimension as in Latin Hypercube sampling. The default setting is NOT to latinize a QMC sample. However, one may

be interested in doing this in situations where one wants better discrepancy of the 1-dimensional projections (the marginal distributions). The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in contributing to the output variance. More details on variance based decomposition are given in Nondeterministic sampling method.

Finally, `quality_metrics` calculates four quality metrics relating to the volumetric spacing of the samples. The four quality metrics measure different aspects relating to the uniformity of point samples in hypercubes. Desirable properties of such point samples are: are the points equally spaced, do the points cover the region, and are they isotropically distributed, with no directional bias in the spacing. The four quality metrics we report are h, chi, tau, and d. h is the point distribution norm, which is a measure of uniformity of the point distribution. Chi is a regularity measure, and provides a measure of local uniformity of a set of points. Tau is the second moment trace measure, and d is the second moment determinant measure. All of these values are scaled so that smaller is better (the smaller the metric, the better the uniformity of the point distribution). Complete explanation of these measures can be found in [Gunzburger and Burkardt, 2004.].

The FSU CVT method (`fsu_cvt`) produces a set of sample points that are (approximately) a Centroidal Voronoi Tessellation. The primary feature of such a set of points is that they have good volumetric spacing; the points tend to arrange themselves in a pattern of cells that are roughly the same shape. To produce this set of points, an almost arbitrary set of initial points is chosen, and then an internal set of iterations is carried out. These iterations repeatedly replace the current set of sample points by an estimate of the centroids of the corresponding Voronoi subregions. [Du, Faber, and Gunzburger, 1999].

The user may generally ignore the details of this internal iteration. If control is desired, however, there are a few variables with which the user can influence the iteration. The user may specify `max_iterations`, the number of iterations carried out; `num_trials`, the number of secondary sample points generated to adjust the location of the primary sample points; and `trial_type`, which controls how these secondary sample points are generated. In general, the variable with the most influence on the quality of the final sample set is `num_trials`, which determines how well the Voronoi subregions are sampled. Generally, `num_trials` should be "large", certainly much bigger than the number of sample points being requested; a reasonable value might be 10,000, but values of 100,000 or 1 million are not unusual.

CVT has a seed specification similar to that in DDACE: there are optional specifications for the random seed to use in generating the sample set (`seed`), for fixing the seed (`fixed_seed`) among multiple sample sets (see Nondeterministic sampling method for discussion), and for the number of samples to perform (`samples`). The `seed` control is used to make sample sets repeatable. Finally, the user has the option to specify the method by which the trials are created to adjust the centroids. The `trial_type` can be one of three types: `random`, where points are generated randomly; `halton`, where points are generated according to the Halton sequence; and `grid`, where points are placed on a regular grid over the hyperspace.

Finally, latinization is available for CVT as with QMC. The `latinize` control takes the CVT sequence and "latinizes" it, meaning that each original sample is moved so that it falls into one strata or bin in each dimension as in Latin Hypercube sampling. The default setting is NOT to latinize a CVT sample. However, one may be interested in doing this in situations where one wants better discrepancy of the 1-dimensional projections (the marginal distributions). The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in contributing to the output variance. More details on variance based decomposition are given in Nondeterministic sampling method. The `quality_metrics` control is available for CVT as with QMC. This command turns on calculation of volumetric quality measures which measure the "goodness" of the uniformity of the point samples. More details on the quality measures are given under the description of the QMC methods.

The specification detail for the FSU CVT method is given in Table 5.53.

Description	Keyword	Associated Data	Status	Default
FSU Quasi-Monte Carlo	`fsu_quasi_mc`	none	Required group	N/A
Sequence type	`halton` | `hammersley`	none	Required group	N/A
Number of samples	`samples`	integer	Optional	(0) for standalone sampling, (minimum required) for surrogates
Sequence starting indices	`sequence_-start`	integer list (one integer per variable)	Optional	Vector of zeroes
Sequence leaping indices	`sequence_-leap`	integer list (one integer per variable)	Optional	Vector of ones
Prime bases for sequences	`prime_base`	integer list (one integer per variable)	Optional	Vector of the first s primes for s-dimensions in Halton, First (s-1) primes for Hammersley
Fixed sequence flag	`fixed_-sequence`	none	Optional	sequence not fixed: sampling patterns are variable among multiple QMC runs
Latinization of samples	`latinize`	none	Optional	No latinization
Variance based decomposition	`variance_-based_decomp`	none	Optional	No variance_-based_decomp
Quality metrics	`quality_-metrics`	none	Optional	No quality_metrics

Table 4.52: Specification detail for FSU Quasi-Monte Carlo sequences

Description	Keyword	Associated Data	Status	Default
FSU CVT sampling	`fsu_cvt`	none	Required group	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Fixed seed flag	`fixed_seed`	none	Optional	seed not fixed: sampling patterns are variable among multiple CVT runs
Number of samples	`samples`	integer	Required	(0) for standalone sampling, (minimum required) for surrogates
Number of trials	`num_trials`	integer	Optional	10000
Trial type	`trial_type`	`random` | `grid` | `halton`	Optional	`random`
Latinization of samples	`latinize`	none	Optional	No latinization
Variance based decomposition	`variance_-based_decomp`	none	Optional	No variance_-based_decomp
Quality metrics	`quality_-metrics`	none	Optional	No quality_metrics

Table 4.53: Specification detail for FSU Centroidal Voronoi Tesselation sampling

4.10.3 PSUADE

The Problem Solving Environment for Uncertainty Analysis and Design Exploration (PSUADE) is a Lawrence Livermore National Laboratory tool for metamodeling, sensitivity analysis, uncertainty quantification, and optimization. Its features include non-intrusive and parallel function evaluations, sampling and analysis methods, an integrated design and analysis framework, global optimization, numerical integration, response surfaces (MARS and higher order regressions), graphical output with Pgplot or Matlab, and fault tolerance [C.H. Tong, 2005].

The Morris One-At-A-Time (MOAT) method, originally proposed by Morris [M.D. Morris, 1991], is a screening method, designed to explore a computational model to distinguish between input variables that have negligible, linear and additive, or nonlinear or interaction effects on the output. The computer experiments performed consist of individually randomized designs which vary one input factor at a time to create a sample of its elementary effects. The PSUADE implementation of MOAT is selected with method keyword `psuade_moat`. The number of samples (`samples`) must be a positive integer multiple of (number of continuous design variable + 1) and will be automatically adjusted if misspecified. The number of partitions (`partitions`) applies to each variable being studied and must be odd (the number of MOAT levels per variable is partitions + 1). This will also be adjusted at runtime as necessary. For information on practical use of this method, see [Saltelli, et al., 2004]. The specification detail for the PSUADE MOAT method is given in Table 5.54.

Description	Keyword	Associated Data	Status	Default
PSUADE MOAT method	psuade_moat	none	Required group	N/A
Random seed	seed	integer	Optional	randomly generated seed
Number of samples	samples	integer	Optional	10∗(num_cdv + 1)
Number of partitions	partitions	integer	Optional	3

Table 4.54: Specification detail for PSUADE methods

4.11 Parameter Study Methods

Dakota's parameter study methods compute response data sets at a selection of points in the parameter space. These points may be specified as a vector, a list, a set of centered vectors, or a multi-dimensional grid. Capability overviews and examples of the different types of parameter studies are provided in the Users Manual [Adams et al., 2010]. Dakota implements all of the parameter study methods within the **ParamStudy** class.

With the exception of output verbosity (a setting of `silent` will suppress some parameter study diagnostic output), Dakota's parameter study methods do not make use of the method independent controls. Therefore, the parameter study documentation which follows is limited to the method dependent controls for the vector, list, centered, and multidimensional parameter study methods.

4.11.1 Vector parameter study

Dakota's vector parameter study computes response data sets at selected intervals along a vector in parameter space. It is often used for single-coordinate parameter studies (to study the effect of a single variable on a response set), but it can be used more generally for multiple coordinate vector studies (to investigate the response variations along some n-dimensional vector such as an optimizer search direction). This study is selected using the `vector_parameter_study` specification followed by either a `final_point` or a `step_vector` specification.

The vector for the study can be defined in several ways (refer to dakota.input.summary). First, a `final_point` specification, when combined with the initial values from the variables specification (in Variables Commands, see `initial_point` and `initial_state` for design and state variables as well as inferred initial values for uncertain variables), uniquely defines an n-dimensional vector's direction and magnitude through its start and end points. The values included in the `final_point` specification are the actual variable values for discrete sets, not the underlying set index value. The intervals along this vector are then specified with a `num_steps` specification, for which the distance between the initial values and the `final_point` is broken into `num_steps` intervals of equal length. For continuous and discrete range variables, distance is measured in the actual values of the variables, but for discrete set variables (either integer or real sets for design, uncertain, or state types), distance is instead measured in index offsets. Since discrete sets may have nonuniform offsets in their enumerated set values but have uniform offsets in their index values, defining steps in terms of set indices allows for meaningful parameter study specifications for these variable types. This study performs function evaluations at both ends, making the total number of evaluations equal to `num_steps+1`. The study has stringent requirements on performing appropriate steps with any discrete range and discrete set variables. A `num_steps` specification must result in discrete range and set index steps that are integral: no remainder is currently permitted in the integer step calculation and no rounding to integer steps will occur. The `final_point` specification detail is given in

Table 5.55.

Description	Keyword	Associated Data	Status	Default
Vector parameter study	vector_-parameter_-study	none	Required group	N/A
Termination point of vector	final_point	list of reals (actual values; no set indices)	Required group	N/A
Number of steps along vector	num_steps	integer	Required	N/A

Table 4.55: final_point specification detail for the vector parameter study

The other technique for defining a vector in the study is the step_vector specification. This parameter study begins at the initial values and adds the increments specified in step_vector to obtain new simulation points. For discrete set types (design, uncertain, or state; real or integer), the steps are set index offsets, not steps between the actual set values. This increment process is performed num_steps times, and since the initial values are included, the total number of simulations is again equal to num_steps+1. The step_vector specification detail is given in Table 5.56.

Description	Keyword	Associated Data	Status	Default
Vector parameter study	vector_-parameter_-study	none	Required group	N/A
Step vector	step_vector	list of reals (index offset components are cast to integers)	Required group	N/A
Number of steps along vector	num_steps	integer	Required	N/A

Table 4.56: step_vector specification detail for the vector parameter study

4.11.2 List parameter study

Dakota's list parameter study allows for evaluations at user selected points of interest which need not follow any particular structure. This study is selected using the list_parameter_study method specification followed by a list_of_points specification.

The number of real values in the list_of_points specification must be a multiple of the total number of variables (including continuous and discrete types) contained in the variables specification. This parameter study simply performs simulations for the first parameter set (the first n entries in the list), followed by the next parameter set (the next n entries), and so on, until the list of points has been exhausted. Since the initial values from the variables specification will not be used, they need not be specified. For discrete set types, the actual values should be specified, not the set indices, although the values will be validated for membership within the set value specifications. The list parameter study specification detail is given in Table 5.57.

Description	Keyword	Associated Data	Status	Default
List parameter study	`list_-parameter_-study`	none	Required group	N/A
List of points to evaluate	`list_of_-points`	list of reals (actual values; no set indices)	Required	N/A

Table 4.57: Specification detail for the list parameter study

4.11.3 Centered parameter study

Dakota's centered parameter study computes response data sets along multiple coordinate-based vectors, one per parameter, centered about the initial values from the variables specification. This is useful for investigation of function contours with respect to each parameter individually in the vicinity of a specific point (e.g., post-optimality analysis for verification of a minimum), thereby avoiding the cost associated with a multidimensional grid. It is selected using the `centered_parameter_study` method specification followed by `step_vector` and `steps_per_variable` specifications. The `step_vector` specification provides the size of the increments for each variable (employed sequentially, not all at once as for Vector parameter study) in either actual values (continuous and discrete range) or index offsets (discrete set). The `steps_per_variable` specification provides the number of increments per variable (again, employed sequentially) in each of the plus and minus directions. The centered parameter study specification detail is given in Table 5.58.

Description	Keyword	Associated Data	Status	Default
Centered parameter study	`centered_-parameter_-study`	none	Required group	N/A
Step vector	`step_vector`	list of reals (index offset components are cast to integers)	Required group	N/A
Number of steps per variable	`steps_per_-variable`	list of integers	Required	N/A

Table 4.58: Specification detail for the centered parameter study

4.11.4 Multidimensional parameter study

Dakota's multidimensional parameter study computes response data sets for an n-dimensional grid of points. Each continuous and discrete range variable is partitioned into equally spaced intervals between its upper and lower bounds, each discrete set variable is partitioned into equally spaced index intervals, and each combination of the values defined by the boundaries of these partitions is evaluated.

This study is selected using the `multidim_parameter_study` method specification followed by a `partitions` specification, where the `partitions` list specifies the number of partitions for each variable. The number of entries in the partitions list must be equal to the total number of variables contained in the variables specification. As for the vector and centered studies, remainders within the integer division of the step calculations are not permitted for discrete range or set types and therefore no integer rounding occurs, so the partitions specification must be carefully selected in the presence of these types. Since the initial values from the variables

specification will not be used, they need not be specified. The multidimensional parameter study specification detail is given in Table 5.59.

Description	Keyword	Associated Data	Status	Default
Multidimensional parameter study	`multidim_-parameter_-study`	none	Required group	N/A
Partitions per variable	`partitions`	list of integers	Required	N/A

Table 4.59: Specification detail for the multidimensional parameter study

Chapter 5

Model Commands

5.1 Model Description

A Dakota model specifies how variables are mapped into a set of responses in support of an iterative method and the model specification in an input file dictates the components to be used in constructing a particular model instance. This specification selects a **Model** from the model hierarchy, which includes **SingleModel**, **DataFitSurrModel**, **HierarchSurrModel**, and **NestedModel** derived classes. Depending on the type of derived model, different sub-specifications are needed to construct different components of the model.

Several examples follow. The first example shows a minimal specification for a single model, which is the default model when no models are specified by the user.

```
model,
single
```

This example does not provide any pointer strings and therefore relies on the default behavior of constructing the model with the last variables, interface, and responses specifications parsed.

The next example displays a surrogate model specification which selects a quadratic polynomial from among the global approximation methods. It uses a pointer to a design of experiments method for generating the data needed for building the global approximation, reuses any old data available for the current approximation region, and employs the first-order multiplicative approach to correcting the approximation each time correction is requested.

```
model,
id_model = 'M1'
variables_pointer = 'V1'
responses_pointer = 'R1'
surrogate global
  polynomial quadratic
  dace_method_pointer = 'DACE'
  reuse_samples region
  correction multiplicative first_order
```

This example demonstrates the use of identifiers and pointers. It provides the optional model independent specifications for model identifier, variables pointer, and responses pointer (see Model Independent Controls) as well as model dependent specifications for global surrogates (see Global approximations).

Finally, an advanced nested model example would be

```
model,
id_model = 'M1'
variables_pointer = 'V1'
responses_pointer = 'R1'
nested
  optional_interface_pointer = 'OI1'
    optional_interface_responses_pointer = 'OIR1'
  sub_method_pointer = 'SM1'
    primary_variable_mapping   = ''  ''  'X'     'Y'
    secondary_variable_mapping = ''  ''  'mean'  'mean'
    primary_response_mapping   = 1. 0. 0. 0. 0. 0. 0. 0. 0.
    secondary_response_mapping = 0. 0. 0. 1. 3. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 3. 0.
```

This example also supplies model independent controls for model identifier, variables pointer, and responses pointer (see Model Independent Controls), and supplies model dependent controls for specifying details of the nested mapping (see Nested Model Controls).

5.2 Model Specification

As alluded to in the examples above, the model specification has the following structure:

```
model,
<model independent controls>
<model selection>
  <model dependent controls>
```

The `<model independent controls>` are those controls which are valid for all models. Referring to dakota.input.summary, these controls are defined externally from and prior to the model selection blocks. The model selection blocks are all required group specifications separated by logical OR's, where the model selection must be `single` OR `surrogate` OR `nested`. If a `surrogate` model is specified, a secondary selection must be made for its type: `global`, `multipoint`, `local`, or `hierarchical`. The `<model dependent controls>` are those controls which are only meaningful for a specific model. These controls are defined within each model selection block. Defaults for model independent and model dependent controls are defined in **DataModel**. The following sections provide additional detail on the model independent controls followed by the model selections and their corresponding model dependent controls.

5.3 Model Independent Controls

The model independent controls include a model identifier string, pointers to variables and responses specifications, and a model type specification. The model identifier string is supplied with `id_model` and is used to provide a unique identifier string for use within method specifications (refer to `model_pointer` in Method Independent Controls).

The type of model can be `single`, `nested`, or `surrogate`. Each of these model specifications supports `variables_pointer` and `responses_pointer` strings for identifying the variables and responses specifications used in constructing the model (by cross-referencing with `id_variables` and `id_responses` strings from particular variables and responses keyword specifications). These pointers are valid for each model type since each model contains a set of variables that is mapped into a set of responses -- only the specifics of the mapping differ. Additional pointers are used for each model type for constructing the components of the variable

to response mapping. As a strategy specification identifies one or more methods and a method specification identifies a model, a model specification identifies variables, responses, and (for some types) interface specifications. This top-down flow specifies all of the object interrelationships.

For each of these pointer specifications, if a pointer string is specified and no corresponding id string is available, Dakota will exit with an error message. If the pointer is optional and no pointer string is specified, then the last specification parsed will be used. It is appropriate to omit optional cross-referencing whenever the relationships are unambiguous due to the presence of only one specification.

Table 6.1 provides the specification detail for the model independent controls involving identifiers, model type controls, and pointers.

Description	Keyword	Associated Data	Status	Default
Model set identifier	`id_model`	string	Optional	method use of last model parsed
Model type	`single` | `surrogate` | `nested`	none	Required group	N/A (`single` if no model specification)
Variables set pointer	`variables_pointer`	string	Optional	model use of last variables parsed
Responses set pointer	`responses_pointer`	string	Optional	model use of last responses parsed

Table 5.1: Specification detail for the model independent controls: identifiers, model types, and pointers

5.4 Single Model Controls

In the `single` model case, a single interface is used to map the variables into responses. The optional `interface_pointer` specification identifies this interface by cross-referencing with the `id_interface` string input from a particular interface keyword specification.

Table 6.2 provides the specification detail for single models.

Description	Keyword	Associated Data	Status	Default
Interface set pointer	`interface_pointer`	string	Optional	model use of last interface parsed

Table 5.2: Specification detail for single models

5.5 Surrogate Model Controls

In the `surrogate` model case, the specification first allows a mixture of surrogate and actual response mappings through the use of the optional `id_surrogates` specification. This identifies the subset of the response functions by number that are to be approximated (the default is all functions). The valid response function identifiers range from 1 through the total number of response functions (see Function Specification). Next, the specification selects a global, multipoint, local, or hierarchical approximation. Table 6.3 provides the specification detail for surrogate models.

Description	Keyword	Associated Data	Status	Default
Surrogate response ids	id_-surrogates	list of integers	Optional	All response functions are approximated
Surrogate type selection	global | multipoint | local | hierarchical	none	Required group	N/A

Table 5.3: Specification detail for the surrogate models

Each of these surrogate types provides an approximate representation of a "truth" model which is used to perform the parameter to response mappings. This approximation is built and updated using data from the truth model. This data is generated in some cases using a design of experiments iterator applied to the truth model (global approximations with a dace_method_pointer). In other cases, truth model data from a single point (local, hierarchical approximations), from a few previously evaluated points (multipoint approximations), or from the restart database (global approximations with reuse_samples) can be used. Surrogate models are used extensively in the surrogate-based optimization and least squares methods (see **SurrBasedMinimizer** and its derived classes and Surrogate-Based Minimization Methods), in which the goals are to reduce expense by minimizing the number of truth function evaluations and to smooth out noisy data with a global data fit. However, the use of surrogate models is not restricted to optimization techniques; uncertainty quantification and optimization under uncertainty methods are other primary users.

The following sections present the global, multipoint, local, and hierarchical specification groups in further detail.

5.5.1 Global approximations

The global surrogate model requires specification of one of the following approximation types: polynomial, gaussian_process, neural_network, mars, moving_least_squares, or radial_basis. These specifications will create a polynomial, a Gaussian process (Kriging interpolation), layered perceptron artificial neural network approximation, multivariate adaptive regression spline, moving least squares, or radial basis function approximation, respectively. All these approximations are implemented in SurfPack except an alternate Gaussian process model, which is implemented directly in Dakota.

In the polynomial case, the order of the polynomial (linear, quadratic, or cubic) must be specified.

Dakota has two Gaussian process (GP) or Kriging models, one that resides in Dakota proper and the other in Dakota's Surfpack subpackage. Choice of the GP implmentation is made by specifying gaussian_process (alternately kriging) followed by either dakota or surfpack. Historically these models were drastically different, but in Dakota 5.1, they became quite similar. They now differ in that the Surfpack GP has a richer set of features/options and tends to be more accurate than the Dakota version. Due to how the Surfpack GP handles ill-conditioned correlation matrices (which significantly contributes to its greater accuracy), the Surfpack GP can be a factor of two or three slower than Dakota's. As of Dakota 5.2, the Surfpack implementation is the default in all contexts except Bayesian calibration. The Dakota GP is provided for continuity, but may be deprecated in the future.

The trend function of both GP models is selected using the trend keyword, with options constant, linear, or reduced_quadratic (default). The Surfpack GP has the additional option of (a full) quadratic. The reduced_quadratic trend function includes the main effects, but not mixed/interaction terms.

The Dakota GP always determines its correlation parameters via maximum likelihood estimate (MLE) using a global optimization procedure. Ill-conditioning induced by poorly spaced sample points is a common challenge in the construction of Gaussian process models. Dakota's GP deals with ill-conditioning in two ways. First, when it encounters a non-invertible correlation matrix it iteratively increases the size of a "nugget," but in such cases the resulting approximation smooths rather than interpolates the data. Second, it has a `point_selection` option (default off) that uses a greedy algorithm to select a well-spaced subset of points prior to the construction of the GP. In this case, the GP will only interpolate the selected subset. Typically, one should not need point selection in trust-region methods because a small number of points are used to develop a surrogate within each trust region. Point selection is most beneficial when constructing with a large number of points, typically more than order one hundred, though this depends on the number of variables and spacing of the sample points.

By default, the Surfpack GP also selects its correlation parameters using a global optimizer, but its search is concentrated in a smaller feasible region of correlation parameters. However, one may force the Surfpack GP to use a different search algorithm by specifying `optimization_method` followed by any of `'global'`, `'local'`, `'sampling'`, or `'none'`. The none option and starting location of the local search default to the center of the small feasible region of correlation lengths. However, one may also directly specify `correlation_lengths` as a list of N real numbers where N is the number of input dimensions. As of Dakota 5.2, the Surfpack GP handles ill-conditioning by selecting the subset of points that tightly meets a constraint on the condition number and provides the maximum amount of useful information for the correlation lengths chosen by MLE. For each set of correlation lengths examined during the MLE optimization, the optimal subset of points for those correlation lengths is used. The subset of points used to build the Surfpack GP is the one associated with the most likely set of correlation lengths. This differs from the `point_selection` option of the Dakota GP which initially chooses a well-spaced subset of points and finds the correlation parameters that are most likely for that one subset. It is similar to the Dakota GP's point selection in that the Surfpack GP will only interpolate the selected subset of points.

The `use_derivatives` keyword will cause the Surfpack GP to be constructed from a combination of function value and gradient information (if available). This is only beneficial when a function value plus a gradient can be computed at a computational cost comparable to a function value only. This rules out gradients computed by finite differences (which also tend to be insufficiently accurate). However, the accurate and inexpensive derivatives calculated by analytical, automatic differentiation, or continuous adjoint techniques can be cost effective and beneficial. Although a function value plus gradient can be often be obtained more cheaply than two function value-only evaluations, each scalar derivative equation costs as much as a scalar function value equation during the construction of the emulator. Since the cost to build a GP is cubic in the number of equations, the cost to build a gradient-enhanced GP/Kriging model will be roughly (N+1)^3 times greater than one that is not gradient-enhanced, where N is the number of input dimensions. Also, a poorly spaced sample design has a greater adverse affect on the conditioning of the correlation matrix when derivative information is included. This can result in significantly more points being discarded and hence providing little benefit to the derivative enhanced GP/Kriging model's accuracy.

Tables 6.4 and 6.5 summarize the global approximation selection specifications.

Table 6.6 summarizes the remaining (optional) global approximation specifications, including controls for the number of points requested to build the surrogate, the source of the points (`dace_method_pointer` and `reuse_samples`), the use of derivative information in constructions (`use_derivatives`), calculation of fitness metrics, and the `correction` method.

The number of points used in building a global approximation is determined by three point counts:

- Model points: Each global surrogate type has an associated minimum reasonable and recommended number of build points, selected via `minimum_points` and `recommended_points` (default), respectively, within a global surrogate model specification. Alternately an integer `total_points` may be specified to

Description	Keyword	Associated Data	Status	Default
Global approximations	global	none	Required group (1 of 4 selections)	N/A
Polynomial	polynomial	linear | quadratic | cubic	Required group (1 of 7 selections)	N/A
Dakota Gaussian process	gaussian_process	dakota	Required (1 of 7 selections)	N/A
GP trend function	trend	constant | linear | reduced_quadratic	Optional	reduced_quadratic trend
GP point selection	point_selection	none	Optional	no point selection
Surfpack Gaussian process (Kriging)	gaussian_process	surfpack	Required group (1 of 7 selections)	N/A
Surfpack GP trend function	trend	constant | linear | reduced_quadratic | quadratic	Optional	reduced_quadratic trend
Surfpack GP correlation lengths	correlation_lengths	list of reals	Optional for none/local	internally computed correlation_lengths
Surfpack GP optimization method	optimization_method	'global' | 'local' | 'sampling' | 'none'	Optional	global
Surfpack GP maximum trials	max_trials	integer	Optional	
Surfpack GP gradient enhancement	use_derivatives	none	Optional	no gradient enhancement

Table 5.4: Specification detail for global approximations: global approximation type, Part I

Description	Keyword	Associated Data	Status	Default
Artificial neural network	neural_network	none	Required (1 of 7 selections)	N/A
ANN number nodes	nodes	integer	Optional	
ANN range	range	real	Optional	
ANN random weight	random_weight	integer	Optional	
Multivariate adaptive regression splines	mars	none	Required (1 of 7 selections)	N/A
MARS maximum bases	max_bases	integer	Optional	
MARS interpolation	interpolation	linear | cubic	Optional	
Moving least squares	moving_least_squares	none	Required (1 of 7 selections)	N/A
MLS polynomial order	poly_order	integer	Optional	
MLS weight function	weight_function	integer	Optional	
Radial basis functions	radial_basis_functions	none	Required group (1 of 7 selections)	N/A
RBF number of bases	bases	integer	Optional	
RBF maximum points	max_pts	integer	Optional	
RBF minimum partitions	min_partition	integer	Optional	
RBF maximum subsets	bases	integer	Optional	

Table 5.5: Specification detail for global approximations: global approximation type, Part II

request a specific number of points in each surrogate build (a lower bound of minimum points is enforced). This count, then, can be summarized as Model points = (minimum | recommended | max(minimum, total)).

- Reused points: any samples available for reuse via the `reuse_samples` keyword (described below) are always used to build. This count defaults to zero.

- Iterator points: `samples` specified within the DACE method specification (identified via `dace_method_pointer`) always provide a lower bound on the number of fresh DACE samples used to build the surrogate. This count defaults to zero.

The number of new points to be evaluated by the DACE method is defined by New points = max(Model points - Reused points, Iterator points), using the logic that the number of points needed is Model points minus Reused points, but respecting a lower bound DACE samples specification when present. The total number of points used in the surrogate build is then New points + Reused points. The DACE iterator will only be invoked if it has new samples to perform, and if new samples are required and no DACE iterator has been provided, an error will result.

The `dace_method_pointer` links to design of experiments iterator used to generate truth model data for building a global data fit. The `reuse_samples` specification can be used to employ old data (either from previous function evaluations performed in the run or from function evaluations read from a restart database or text file) in the building of new global approximations. The default is no reuse of old data (since this can induce directional bias), and the settings of `all`, `region`, and `points_file` result in reuse of all available data, reuse of all data available in the current trust region, and reuse of all data from a specified text file, respectively. Both `dace_method_pointer` and `reuse_samples` are optional specifications, which gives the user maximum flexibility in using design of experiments data, restart/text file data, or both. Note that the default format for a samples data file has changed in Dakota 5.2 (see User's Manual).

The `use_derivatives` flag specifies that any available derivative information should be used in global approximation builds, for those global surrogate types that support it (currently, polynomial regression and the Surfpack GP).

To assess the goodness of fit of a global surrogate, a variety of diagnostics are available for the following global approximation methods: polynomial regressions, kriging, mars, moving least squares, neural networks, and radial basis functions. The diagnostics are specified by the keyword `diagnostics`, followed by a list of strings specifying the actual metrics. The diagnostic metrics available are: `sum_squared`, `mean_squared`, `root_mean_squared`, `max_squared`, `sum_scaled`, `max_scaled`, `sum_abs`, `mean_abs`, `max_abs`, `press`, `cv`, and `rsquared`. Most of these diagnostics refer to some operation on the residuals (the difference between the surrogate model and the truth model at the data points upon which the surrogate is built). For example, `sum_squared` refers to the sum of the squared residuals, `mean_abs` refers to the mean of the absolute value of the residuals, and `max_scaled` refers to the maximum scaled value of the residual, where the scaled value is taken by dividing the residual by the true function value at that point. In addition, `press` refers to a leave-one-out cross validation metric, and `cv` refers to a general cross validation metric. `rsquared` refers to the R-squared value typically used in regression analysis (the proportion of the variability in the response that can be accounted for by the surrogate model).

The `correction` specification specifies that the approximation will be corrected to match truth data, either matching truth values in the case of `zeroth_order` matching, matching truth values and gradients in the case of `first_order` matching, or matching truth values, gradients, and Hessians in the case of `second_order` matching. For `additive` and `multiplicative` corrections, the correction is local in that the truth data is matched at a single point, typically the center of the approximation region. The `additive` correction adds a scalar offset (`zeroth_order`), a linear function (`first_order`), or a quadratic function (`second_order`) to the approximation to match the truth data at the point, and the `multiplicative` correction multiplies

the approximation by a scalar (zeroth_order), a linear function (first_order), or a quadratic function (second_order) to match the truth data at the point. The additive first_order case is due to [Lewis and Nash, 2000] and the multiplicative first_order case is commonly known as beta correction [Haftka, 1991]. For the combined correction, the use of both additive and multiplicative corrections allows the satisfaction of an additional matching condition, typically the truth function values at the previous correction point (e.g., the center of the previous trust region). The combined correction is then a multipoint correction, as opposed to the local additive and multiplicative corrections. Each of these correction capabilities is described in detail in [Eldred et al., 2004a].

5.5.2 Multipoint approximations

Multipoint approximations use data from previous design points to improve the accuracy of local approximations. Currently, the Two-point Adaptive Nonlinearity Approximation (TANA-3) method of [Xu and Grandhi, 1998] is supported. This method requires response value and gradient information from two points, and uses a first-order Taylor series if only one point is available. The truth model to be used to generate the value/gradient data used in the approximation is identified through the required actual_model_pointer specification. Table 6.7 summarizes the multipoint approximation specifications.

5.5.3 Local approximations

Local approximations use value, gradient, and possibly Hessian data from a single point to form a series expansion for approximating data in the vicinity of this point. The currently available local approximation is the taylor_series selection. The order of the Taylor series may be either first-order or second-order, which is automatically determined from the gradient and Hessian specifications in the responses specification (see Gradient Specification and Hessian Specification) for the truth model.

The truth model to be used to generate the value/gradient/Hessian data used in the series expansion is identified through the required actual_model_pointer specification. The use of a model pointer (as opposed to an interface pointer) allows additional flexibility in defining the approximation. In particular, the derivative specification for the truth model may differ from the derivative specification for the approximation , and the truth model results being approximated may involve a model recursion (e.g., the values/gradients from a nested model). Table 6.8 summarizes the local approximation interface specifications.

5.5.4 Hierarchical approximations

Hierarchical approximations use corrected results from a low fidelity model as an approximation to the results of a high fidelity "truth" model. These approximations are also known as model hierarchy, multifidelity, variable fidelity, and variable complexity approximations. The required low_fidelity_model_pointer specification points to the low fidelity model specification. This model is used to generate low fidelity responses which are then corrected and returned to an iterator. The required high_fidelity_model_pointer specification points to the specification for the high fidelity truth model. This model is used only for verifying low fidelity results and updating low fidelity corrections. The correction specification specifies which correction technique will be applied to the low fidelity results in order to match the high fidelity results at one or more points. In the hierarchical case (as compared to the global case), the correction specification is required, since the omission of a correction technique would effectively eliminate the purpose of the high fidelity model. If it is desired to use a low fidelity model without corrections, then a hierarchical approximation is not needed and a single model should be used. Refer to Global approximations for additional information on available correction approaches.

Description	Keyword	Associated Data	Status	Default
Number of build points	total_points | minimum_points | recommended_points	integer | none | none	Optional	recommended_points
Design of experiments method pointer	dace_method_pointer	string	Optional	no design of experiments data
Sample reuse in global approximation	reuse_samples	all | region | none	Optional group	all if points_file defined, none otherwise
File import of samples for global approximation builds	points_file	string	Optional	no point import from a file
Data file in annotated format	annotated	boolean	Optional	annotated
Data file in freeform format	freeform	boolean	Optional	annotated
Use of derivative data in global approximation builds	use_derivatives	none	Optional	derivative data not used in global approximation builds
Print diagnostic metrics about the surrogate goodness of fit	diagnostics	string list (as many as desired): sum_squared, mean_squared, root_mean_squared, max_squared, sum_scaled, max_scaled, sum_abs, mean_abs, max_abs, press, cv, and rsquared	Optional	No diagnostics printed
Surrogate correction approach	correction	additive | multiplicative | combined; zeroth_order | first_order | second_order	Optional group	no surrogate correction

Table 5.6: Specification detail for global approximations: build and correction controls

Description	Keyword	Associated Data	Status	Default
Multipoint approximation	multipoint	none	Required group (1 of 4 selections)	N/A
Two-point adaptive nonlinear approximation	tana	none	Required	N/A
Pointer to the truth model specification	actual_-model_-pointer	string	Required	N/A

Table 5.7: Specification detail for multipoint approximations

Description	Keyword	Associated Data	Status	Default
Local approximation	local	none	Required group (1 of 4 selections)	N/A
Taylor series local approximation	taylor_-series	none	Required	N/A
Pointer to the truth model specification	actual_-model_-pointer	string	Required	N/A

Table 5.8: Specification detail for local approximations

Table 6.9 summarizes the hierarchical approximation specifications.

Description	Keyword	Associated Data	Status	Default
Hierarchical approximation	hierarchical	none	Required group (1 of 4 selections)	N/A
Pointer to the low fidelity model specification	low_-fidelity_-model_-pointer	string	Required	N/A
Pointer to the high fidelity model specification	high_-fidelity_-model_-pointer	string	Required	N/A
Surrogate correction approach	correction	additive | multiplicative | combined; zeroth_order | first_order | second_order	Required group	N/A

Table 5.9: Specification detail for hierarchical approximations

5.6 Nested Model Controls

Instead of appealing directly to a primary interface, a nested model maps parameters to responses by executing a sub-iterator for each evaluation of the nested model; the sub-iterator in turn operates on a sub-model. The sub-iterator responses may be combined with non-nested contributions from an optional interface specification.

In the `nested` model case, a `sub_method_pointer` must be provided in order to specify the nested iterator, and `optional_interface_pointer` and `optional_interface_responses_pointer` provide an optional group specification for the optional interface portion of nested models (where `optional_interface_pointer` points to the interface specification and `optional_interface_responses_pointer` points to a responses specification describing the data to be returned by this interface). This interface is used to provide non-nested data, which is then combined with data from the nested iterator using the `primary_response_mapping` and `secondary_response_mapping` inputs (see mapping discussion below).

Table 6.10 provides the specification detail for nested model pointers.

Description	Keyword	Associated Data	Status	Default
Optional interface set pointer	`optional_interface_pointer`	string	Optional group	no optional interface
Responses pointer for nested model optional interfaces	`optional_interface_responses_pointer`	string	Optional	reuse of top-level responses specification
Sub-method pointer for nested models	`sub_method_pointer`	string	Required	N/A

Table 5.10: Specification detail for nested models

Nested models may employ mappings for both the variable inputs to the sub-model and the response outputs from the sub-model. In the former case, the `primary_variable_mapping` and `secondary_variable_mapping` specifications are used to map from the active top-level variables into the sub-model variables, and in the latter case, the `primary_response_mapping` and `secondary_response_mapping` specifications are used to compute the sub-model response contributions to the top-level responses.

For the variable mappings, the primary and secondary specifications provide lists of strings which are used to target specific sub-model variables and their sub-parameters, respectively. The primary strings are matched to continuous or discrete variable labels such as `'cdv_1'` (either user-supplied or default labels), and the secondary strings are matched to either real or integer random variable distribution parameters such as `'mean'` or `'num_trials'` (the form of the uncertain distribution parameter keyword that is appropriate for a single variable instance) or continuous or discrete design/state variable sub-parameters such as `'lower_bound'` or `'upper_bound'` (again, keyword form appropriate for a single variable instance). No coercion of types is supported, so real-valued top-level variables should map to either real-valued sub-model variables or real-valued sub-parameters and integer-valued top-level variables should map to either integer-valued sub-model variables or integer-valued sub-parameters. As long as these real versus integer constraints are satisfied, mappings are free to cross variable types (design, aleatory uncertain, epistemic uncertain, state) and domain types (continuous, discrete). Both `primary_variable_mapping` and `secondary_variable_mapping` specifications are optional, which is designed to support the following three possibilities:

1. If both primary and secondary variable mappings are specified, then an active top-level variable value will be inserted into the identified sub-parameter (the secondary mapping) for the identified sub-model variable (the primary mapping).
2. If a primary mapping is specified but a secondary mapping is not, then an active top-level variable value will be inserted into the identified sub-model variable value (the primary mapping).
3. If a primary mapping is not specified (corresponding secondary mappings, if specified, are ignored), then an active top-level variable value will be inserted into a corresponding sub-model variable, based on matching of variable types (e.g., top-level and sub-model variable specifications both allocate a set of `'continuous_design'` variables which are active at the top level). Multiple sub-model variable types may be updated in this manner, provided that they are all active in the top-level variables. Since there is a direct variable correspondence for these default insertions, sub-model bounds and labels are also updated from the top-level bounds and labels in order to eliminate the need for redundant input file specifications. Thus, it is typical for the sub-model variables specification to only contain the minimal required information, such as the number of variables, for these insertion targets. The sub-model must allocate enough space for each of the types that will accept default insertions, and the leading set of matching sub-model variables are updated (i.e., the sub-model may allocate more than needed and the trailing set will be unmodified).

These different variable mapping possibilities may be used in any combination by employing empty strings (″) for particular omitted mappings (the number of strings in user-supplied primary and secondary variable mapping specifications must equal the total number of active top-level variables, including both continuous and discrete types). The ordering of the active variables is the same as shown in dakota.input.summary and as presented in Variables Commands.

If inactive variables are present at the outer level, then the default type 3 mapping is used for these variables; that is, outer loop inactive variables are inserted into inner loop variables (active or inactive) based on matching of variable types, top-level bounds and labels are also propagated, the inner loop must allocate sufficient space to receive the outer loop values, and the leading subset within this inner loop allocation is updated. This capability is important for allowing nesting beyond two levels, since an active variable at the outer-most loop may become inactive at the next lower level, but still needs to be further propagated down to lower levels in the recursion.

For the response mappings, the primary and secondary specifications provide real-valued multipliers to be applied to sub-iterator response results so that the responses from the inner loop can be mapped into a new set of responses at the outer loop. For example, if the nested model is being employed within a mixed aleatory-epistemic uncertainty quantification, then aleatory statistics from the inner loop (such as moments of the response) are mapped to the outer level, where minima and maxima of these aleatory statistics are computed as functions of the epistemic parameters. The response mapping defines a matrix which scales the values from the inner loop and determines their position in the outer loop response vector. Each row of the mapping corresponds to one outer loop response, where each column of the mapping corresponds to a value from the inner loop. Depending on the number of responses and the particular attributes calculated on the inner loop, there will be a vector of inner loop response values that need to be accounted for in the mapping. This vector of inner loop response results is defined as follows for different sub-iterator types:

- optimization: the final objective function(s) and nonlinear constraints
- nonlinear least squares: the final least squares terms and nonlinear constraints
- aleatory uncertainty quantification (UQ): for each response function, a mean statistic, a standard deviation statistic, and all probability/reliability/generalized reliability/response level results for any user-specified `response_levels`, `probability_levels`, `reliability_levels`, and/or `gen_reliability_levels`, in that order.

- epistemic and mixed aleatory/epistemic UQ using interval estimation methods: lower and upper interval bounds for each response function.
- epistemic and mixed aleatory/epistemic UQ using evidence methods: for each response function, lower and upper interval bounds (belief and plausibility) for all probability/reliability/generalized reliability/response level results computed from any user-specified `response_levels`, `probability_levels`, `reliability_levels`, and/or `gen_reliability_levels`, in that order.
- parameter studies and design of experiments: for optimization and least squares response data sets, the best solution found (lowest constraint violation if infeasible, lowest composite objective function if feasible). For generic response data sets, a best solution metric is not defined, so the sub-iterator response vector is empty in this case.

The primary values map sub-iterator response results into top-level objective functions, least squares terms, or generic response functions, depending on the declared top-level response set. The secondary values map sub-iterator response results into top-level nonlinear inequality and equality constraints. Refer to **NestedModel::response_mapping()** for additional details.

An example of variable and response mappings is provided below:

```
primary_variable_mapping   = ''  ''  'X'     'Y'
secondary_variable_mapping = ''  ''  'mean'  'mean'
primary_response_mapping   = 1. 0. 0. 0. 0. 0. 0. 0. 0.
secondary_response_mapping = 0. 0. 0. 1. 3. 0. 0. 0. 0.
                             0. 0. 0. 0. 0. 0. 1. 3. 0.
```

The variable mappings correspond to 4 top-level variables, the first two of which employ the default mappings from active top-level variables to sub-model variables of the same type (option 3 above) and the latter two of which are inserted into the mean distribution parameters of sub-model variables `'X'` and `'Y'` (option 1 above). The response mappings define a 3 by 9 matrix corresponding to 9 inner loop response attributes and 3 outer loop response functions (one primary response function and 2 secondary functions, such as one objective and two constraints). Each row of the response mapping is a vector which is multiplied (i.e, with a dot-product) against the 9 sub-iterator values to determine the outer loop function. Consider a UQ example with 3 response functions, each providing a mean, a standard deviation, and one level mapping (if no level mappings are specified, the responses would only have a mean and standard deviation). The primary response mapping can be seen to extract the first value from the inner loop, which would correspond to the mean of the first response function. This mapped sub-iterator response becomes a single objective function, least squares term, or generic response function at the outer level, as dictated by the top-level response specification. The secondary response mapping maps the fourth sub-iterator response function plus 3 times the fifth sub-iterator response function (mean plus 3 standard deviations) into one top-level nonlinear constraint and the seventh sub-iterator response function plus 3 times the eighth sub-iterator response function (mean plus 3 standard deviations) into another top-level nonlinear constraint, where these top-level nonlinear constraints may be inequality or equality, as dictated by the top-level response specification. Note that a common case is for each sub-iterator response to be mapped to a unique outer loop response (for example, in the nested UQ case where one wants to determine an interval on each inner loop statistic). In these simple cases, the response mapping would define an identity matrix.

Table 6.11 provides the specification detail for the model independent controls involving nested model mappings.

Description	Keyword	Associated Data	Status	Default
Primary variable mappings for nested models	`primary_variable_mapping`	list of strings	Optional	default variable insertions based on variable type
Secondary variable mappings for nested models	`secondary_variable_mapping`	list of strings	Optional	primary mappings into sub-model variables are value-based
Primary response mappings for nested models	`primary_response_mapping`	list of reals	Optional	no sub-iterator contribution to primary functions
Secondary response mappings for nested models	`secondary_response_mapping`	list of reals	Optional	no sub-iterator contribution to secondary functions

Table 5.11: Specification detail for the model independent controls: nested model mappings

Chapter 6

Variables Commands

6.1 Variables Description

The variables section in a Dakota input file specifies the parameter set to be iterated by a particular method. This parameter set is made up of design, uncertain, and state variables. Design variables can be continuous or discrete and consist of those variables which an optimizer adjusts in order to locate an optimal design. Each of the design parameters can have an initial point and a descriptive tag. Continuous and discrete range types include lower and upper bounds, and discrete set types include the admissible set values.

Uncertain variables may be categorized as either aleatory or epistemic and either continuous or discrete. Continuous aleatory uncertain variables include normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, and histogram bin distributions. Discrete aleatory uncertain variables include poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point distributions. In addition to aleatory uncertain variables defined by probability distributions, Dakota also supports epistemic uncertain variables that are non-probabilistic. These epistemic types involve continuous or discrete interval or discrete set specifications. For the interval cases, one defines either a single bounded interval per epistemic variable or a set of basic probability assignments (BPAs) that define belief structures, where each BPA defines the level of belief that the value of a particular variable lies within a particular interval. For discrete epistemic sets, the basic probability assignments are provided for each set value as opposed to each interval.

Each uncertain variable specification contains descriptive tags and most contain, either explicitly or implicitly, distribution lower and upper bounds. Distribution lower and upper bounds are explicit portions of the normal, lognormal, uniform, loguniform, triangular, and beta specifications, whereas they are implicitly defined for histogram bin, histogram point, and interval variables (from the extreme values within the bin/point/interval specifications) as well as for binomial (0 to `num_trials`) and hypergeometric (0 to min(`num_drawn`, `num_selected`)) variables. When used with design of experiments and multidimensional parameter studies, distribution bounds are also inferred for normal and lognormal (if optional bounds are unspecified) as well as for exponential, gamma, gumbel, frechet, weibull, poisson, negative binomial, and geometric (which have no bounds specifications); these bounds are [0, $\mu + 3\sigma$] for exponential, gamma, frechet, weibull, poisson, negative binomial, geometric, and unspecified lognormal, and [$\mu - 3\sigma$, $\mu + 3\sigma$] for gumbel and unspecified normal. For other types of parameter studies (vector and centered), an inferred initial starting point is needed for the uncertain variables. All uncertain variables are initialized to their means for these studies, where mean values for bounded normal and bounded lognormal may additionally be repaired to satisfy any specified distribution bounds, mean values for discrete integer range distributions are rounded down to the nearest integer, and mean values for discrete set distributions

are rounded to the nearest set value. In addition to tags and bounds specifications, normal variables include mean and standard deviation specifications, lognormal variables include lambda and zeta, mean and standard deviation, or mean and error factor specifications, triangular variables include mode specifications, exponential variables include beta specifications, beta, gamma, gumbel, frechet, and weibull variables include alpha and beta specifications, histogram bin variables include abscissa and either ordinate or count specifications, poisson variables include lambda specifications, binomial and negative binomial variables include probability per trial and number of trials specifications, geometric variables include probability per trial specifications, hypergeometric variables include the specification of the total population, selected population, and number drawn, histogram point variables include abscissa and count specifications, and interval variables include basic probability assignments per interval.

State variables can be continuous or discrete and consist of "other" variables which are to be mapped through the simulation interface. Each state variable specification can have an initial state and descriptors. Continuous and discrete range types include lower and upper bounds, and discrete set types include the admissible set values. State variables provide a convenient mechanism for parameterizing additional model inputs, such as mesh density, simulation convergence tolerances and time step controls, and can be used to enact model adaptivity in future strategy developments.

The ordering of variables is important, and a consistent ordering is employed throughout the Dakota software. It is the same ordering as shown in dakota.input.summary and as presented in the outline of this chapter. That ordering can be summarized as continuous followed by discrete integer followed by discrete real within each of the following types: design, aleatory uncertain, epistemic uncertain, and state. Ordering of variable types below this granularity (e.g., from normal to histogram bin within continuous aleatory uncertain) is defined somewhat arbitrarily, but is enforced consistently throughout the code.

Several examples follow. In the first example, two continuous design variables are specified:

```
variables,
continuous_design = 2
  initial_point    0.9    1.1
  upper_bounds     5.8    2.9
  lower_bounds     0.5   -2.9
  descriptors   'radius' 'location'
```

In the next example, defaults are employed. In this case, `initial_point` will default to a vector of `0.` values, `upper_bounds` will default to vector values of `DBL_MAX` (the maximum number representable in double precision for a particular platform, as defined in the platform's `float.h` C header file), `lower_bounds` will default to a vector of `-DBL_MAX` values, and `descriptors` will default to a vector of `'cdv_i'` strings, where `i` ranges from one to two:

```
variables,
continuous_design = 2
```

In the following example, the syntax for a normal-lognormal distribution is shown. One normal and one lognormal uncertain variable are completely specified by their means and standard deviations. In addition, the dependence structure between the two variables is specified using the `uncertain_correlation_matrix`.

```
variables,
        normal_uncertain    =  1
          means             =  1.0
          std_deviations    =  1.0
          descriptors       =  'TF1n'
        lognormal_uncertain =  1
```

```
          means             =  2.0
          std_deviations    =  0.5
          descriptors       =  'TF2ln'
        uncertain_correlation_matrix =  1.0 0.2
                                        0.2 1.0
```

An example of the syntax for a state variables specification follows:

```
variables,
        continuous_state = 1
          initial_state       4.0
          lower_bounds        0.0
          upper_bounds        8.0
          descriptors        'CS1'
        discrete_state_range = 1
          initial_state       104
          lower_bounds        100
          upper_bounds        110
          descriptors        'DS1'
```

And in a more advanced example, a variables specification containing a set identifier, continuous and discrete design variables, normal and uniform uncertain variables, and continuous and discrete state variables is shown:

```
variables,
id_variables = 'V1'
continuous_design = 2
  initial_point    0.9    1.1
  upper_bounds     5.8    2.9
  lower_bounds     0.5   -2.9
  descriptors   'radius' 'location'
discrete_design_range = 1
  initial_point    2
  upper_bounds     1
  lower_bounds     3
  descriptors   'material'
normal_uncertain = 2
  means          =  248.89, 593.33
  std_deviations =   12.4,   29.7
  descriptors    =  'TF1n'   'TF2n'
uniform_uncertain = 2
  lower_bounds =  199.3,  474.63
  upper_bounds =  298.5,  712.
  descriptors  =  'TF1u'   'TF2u'
continuous_state = 2
  initial_state = 1.e-4  1.e-6
  descriptors   = 'EPSIT1' 'EPSIT2'
discrete_state_set_int = 1
  initial_state = 100
  set_values    = 100 212 375
  descriptors   = 'load_case'
```

Refer to the Dakota Users Manual [Adams et al., 2010] for discussion on how different iterators view these mixed variable sets. Further control over how variables are treated may be specified with the keyword `active`, which controls which types of variables are active for a particular iterative method, and domain, which specifies how discrete vs. continuous variables are handled.

6.2 Variables Specification

The variables specification has the following structure:

```
variables,
<set identifier>
        <active variable specification>
        <variable domain specification>
<continuous design variables specification>
<discrete design range variables specification>
<discrete design set integer variables specification>
<discrete design set real variables specification>
<normal uncertain variables specification>
<lognormal uncertain variables specification>
<uniform uncertain variables specification>
<loguniform uncertain variables specification>
<triangular uncertain variables specification>
<exponential uncertain variables specification>
<beta uncertain variables specification>
<gamma uncertain variables specification>
<gumbel uncertain variables specification>
<frechet uncertain variables specification>
<weibull uncertain variables specification>
<histogram bin uncertain variables specification>
<poisson uncertain variables specification>
<binomial uncertain variables specification>
<negative binomial uncertain variables specification>
<geometric uncertain variables specification>
<hypergeometric uncertain variables specification>
<histogram point uncertain variables specification>
<uncertain correlation specification>
<continuous interval uncertain variables specification>
<discrete interval uncertain range variables specification>
<discrete uncertain set integer variables specification>
<discrete uncertain set real variables specification>
<continuous state variables specification>
<discrete state range variables specification>
<discrete state set integer variables specification>
<discrete state set real variables specification>
```

Referring to dakota.input.summary, it is evident from the enclosing brackets that the set identifier specification, the active variable and domain specification, the uncertain correlation specification, and each of the variables specifications are all optional. The set identifier and uncertain correlation are stand-alone optional specifications, whereas the variables specifications are optional group specifications, meaning that the group can either appear or not as a unit. If any part of an optional group is specified, then all required parts of the group must appear.

The optional status of the different variable type specifications allows the user to specify only those variables which are present (rather than explicitly specifying that the number of a particular type of variables is zero). However, at least one type of variables that are active for the iterator in use must have nonzero size or an input error message will result. The following sections describe each of these specification components in additional detail.

6.3 Variables Set Identifier

The optional set identifier specification uses the keyword `id_variables` to input a unique string for use in identifying a particular variables set. A model can then identify the use of this variables set by specifying the

same string in its `variables_pointer` specification (see Model Independent Controls). For example, a model whose specification contains `variables_pointer = 'V1'` will use a variables specification containing the set identifier `id_variables = 'V1'`.

If the `id_variables` specification is omitted, a particular variables set will be used by a model only if that model omits specifying a `variables_pointer` and if the variables set was the last set parsed (or is the only set parsed). In common practice, if only one variables set exists, then `id_variables` can be safely omitted from the variables specification and `variables_pointer` can be omitted from the model specification(s), since there is no potential for ambiguity in this case. Table 7.1 summarizes the set identifier inputs.

Description	Keyword	Associated Data	Status	Default
Variables set identifier	`id_variables`	string	Optional	use of last variables parsed

Table 6.1: Specification detail for set identifier

6.4 Active Variables

There are certain situations where the user may want to explicitly control the subset of variables that is considered active for a certain Dakota method. This is done by specifying the keyword `active` in the variables specification block, followed by one of the following: `all`, `design`, `uncertain`, `aleatory`, `epistemic`, or `state`. Specifying one of these subsets of variables will allow the Dakota method to operate on the specified variable types and override the default active subset. For example, the default behavior for a nondeterministic sampling method is to sample the uncertain variables. However, if the user specifed `active all` in the variables specification block, the sampling would be performed over all variables (e.g. design and state variables in addition to the uncertain variables). This may be desired in situations such as surrogate-based optimization under uncertainty, where a surrogate may be constructed to span both design and uncertain variables. This is an example where we expand the active subset beyond the default, but in other situations, we may wish to restrict from the default. An example of this would be performing design of experiments in the presence of multiple variable types (for which all types are active by default), but only wanting to sample over the design variables for purposes of constructing a surrogate model for optimization.

If the user does not specify any explicit override of the active view of the variables, Dakota first considers the response function specification (refer to Function Specification). If the user specifies objective functions or calibration terms in the response specification block, then we can infer that the active variables should be the design variables (since design variables are used within optimization and least squares methods). If the user instead specifies the generic response type of `response_functions` (see Response functions (generic data set)), then Dakota cannot infer the active variable subset from the response specification and will instead infer it from the method selection. If the method is a parameter study, or any of the methods available under dace, psuade, or fsu methods, the active view is set to all variables. For uncertainty quantification methods, if the method is sampling, then the view is set to aleatory if only aleatory variables are present, epistemic if only epistemic variables are present, or uncertain (covering both aleatory and epistemic) if both are present. If the uncertainty method involves interval estimation or evidence calculations, the view is set to epistemic. For other uncertainty quantification methods not mentioned in the previous sentences (e.g., reliability methods or stochastic expansion methods), the default view is set to aleatory. Finally, for verification studies using the Richardson extrapolation method, the active view is set to state. Table 7.2 summarizes the way to specify the active variables.

6.5 Variables Domain

The variables domain specifies how the discrete variables are treated. If the user specifies `mixed` in the variable specification block, the continuous and discrete variables are treated separately. If the user specifies `relaxed` in the variable specification block, the discrete variables are relaxed and treated as continuous variables. This may be useful in optimization problems involving both continuous and discrete variables when a user would like to use an optimization method that is designed for continuous variable optimization. All Dakota methods have a default value of mixed for the domain type except for the branch-and-bound method which has a default domain type of relaxed. Note that the branch-and-bound method is under development at this time. Finally, note that the domain selection applies to all variable types: design, aleatory uncertain, epistemic uncertain, and state. Table 7.2 describes the domain specification.

Description	Keyword	Associated Data	Status	Default
Active variables	`active`	`all` | `design` | `uncertain` | `aleatory` | `epistemic` | `state`	Optional	Infer from response or method specification
Domain of variables	`mixed` | `relaxed`	N/A	Optional	`relaxed` (branch and bound), `mixed` (all other methods)

Table 6.2: Specification detail for active and domain settings

6.6 Design Variables

Design variable types include continuous real, discrete range of integer values (contiguous integers), discrete set of integer values, and discrete set of real values. Within each optional design variables specification group, the number of variables is always required. The following Tables 7.3 through 7.6 summarize the required and optional specifications for each design variable subtype. The `initial_point` specifications provide the point in design space from which an iterator is started and default to either zeros (continuous and discrete range) or middle values (discrete sets). The `descriptors` specifications supply strings which will be replicated through the Dakota output to identify the numerical values for these parameters; these default to numbered strings.

For continuous and discrete range variables, the `lower_bounds` and `upper_bounds` restrict the size of the feasible design space and are frequently used to prevent nonphysical designs. Default values are positive and negative machine limits for upper and lower bounds (+/- `DBL_MAX`, `INT_MAX`, `INT_MIN` from the `float.h` and `limits.h` system header files). As for linear and nonlinear inequality constraint bounds (see Method Independent Controls and Objective and constraint functions (optimization data set)), a nonexistent upper bound can be specified by using a value greater than the "big bound size" constant (1.e+30 for continuous variables, 1.e+9 for discrete integer variables) and a nonexistent lower bound can be specified by using a value less than the negation of these constants (-1.e+30 for continuous, -1.e+9 for discrete integer), although not all optimizers currently support this feature (e.g., DOT and CONMIN will treat these large bound values as actual variable bounds, but this should not be problematic in practice).

Description	Keyword	Associated Data	Status	Default
Continuous design variables	continuous_-design	integer	Optional group	no continuous design variables
Initial point	initial_-point	list of reals	Optional	vector values = 0. (repaired to bounds, if required)
Lower bounds	lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Upper bounds	upper_bounds	list of reals	Optional	vector values = +DBL_MAX
Scaling types	scale_types	list of strings	Optional	vector values = 'none'
Scales	scales	list of reals	Optional	vector values = 1. (no scaling)
Descriptors	descriptors	list of strings	Optional	vector of 'cdv_i' where i = 1,2,3...

Table 6.3: Specification detail for continuous design variables

6.6.1 Continuous Design Variables

For continuous variables, the scale_types specification includes strings specifying the scaling type for each component of the continuous design variables vector in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in scale_types may be selected from 'none', 'value', 'auto', or 'log', to select no, characteristic value, automatic, or logarithmic scaling, respectively. If a single string is specified it will apply to all components of the continuous design variables vector. Each entry in scales may be a user-specified nonzero real characteristic value to be used in scaling each variable component. These values are ignored for scaling type 'none', required for 'value', and optional for 'auto' and 'log'. If a single real value is specified it will apply to all components of the continuous design variables vector.

6.6.2 Discrete Design Range Variables

6.6.3 Discrete Design Integer Set Variables

Discrete set variables are specified with an integer list specifying how many set members there are for each variable and a list of integer or real set values for discrete_design_set_integer (Table 7.5) and discrete_design_set_real (Table 7.6), respectively.

6.6.4 Discrete Design Real Set Variables

6.7 Aleatory Uncertain Variables

Aleatory uncertain variables involve continuous or discrete probability distribution specifications. Continuous probability distributions including normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma,

Description	Keyword	Associated Data	Status	Default
Discrete design range variables	discrete_-design_range	integer	Optional group	no discrete design variables
Initial point	initial_-point	list of integers	Optional	vector values = 0 (repaired to bounds, if required)
Lower bounds	lower_bounds	list of integers	Optional	vector values = INT_MIN
Upper bounds	upper_bounds	list of integers	Optional	vector values = INT_MAX
Descriptors	descriptors	list of strings	Optional	vector of 'ddriv_i' where i = 1,2,3,...

Table 6.4: Specification detail for discrete design range variables

Description	Keyword	Associated Data	Status	Default
Discrete design set of integer variables	discrete_-design_set_-integer	integer	Optional group	no discrete design set of integer variables
Initial point	initial_-point	list of integers	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	num_set_-values	list of integers	Optional	equal distribution
Set values	set_values	list of integers	Required	N/A
Descriptors	descriptors	list of strings	Optional	vector of 'ddsiv_i' where i = 1,2,3,...

Table 6.5: Specification detail for discrete design set of integer variables

Description	Keyword	Associated Data	Status	Default
Discrete design set of real variables	discrete_-design_set_-real	integer	Optional group	no discrete design set of real variables
Initial point	initial_-point	list of reals	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	num_set_-values	list of integers	Optional	equal distribution
Set values	set_values	list of reals	Required	N/A
Descriptors	descriptors	list of strings	Optional	vector of 'ddsrv_i' where i = 1,2,3,...

Table 6.6: Specification detail for discrete design set of real variables

gumbel, frechet, weibull, and histogram bin distributions. Discrete probability distributions include poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point distributions. Each of these specifications is an optional group specification.

These specifications of probability distributions directly support the use of probabilistic uncertainty quantification methods such as sampling, reliability, and stochastic expansion methods. However, the inclusion of lower and upper distribution bounds for all uncertain variable types (either explicitly defined, implicitly defined, or inferred; see Variables Description) also allows the use of these variables within methods that rely on a bounded region to define a set of function evaluations (i.e., design of experiments and some parameter study methods). Each distribution also provides optional uncertain variable descriptors (default values are numbered strings) that supply identifiers that help associate the numerical values with the uncertain parameters as they appear within the Dakota output. Tables 7.7 through 7.24 summarize the details of the aleatory uncertain variable specifications.

6.7.1 Normal Distribution

Within the normal uncertain optional group specification, the number of normal uncertain variables, the means, and standard deviations are required specifications, and the distribution lower and upper bounds and variable descriptors are optional specifications. The normal distribution is widely used to model uncertain variables such as population characteristics. It is also used to model the mean of a sample: as the sample size becomes very large, the Central Limit Theorem states that the mean becomes approximately normal, regardless of the distribution of the original variables.

The density function for the normal distribution is:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_N} e^{-\frac{1}{2}\left(\frac{x-\mu_N}{\sigma_N}\right)^2}$$

where μ_N and σ_N are the mean and standard deviation of the normal distribution, respectively.

Note that if you specify bounds for a normal distribution, the sampling occurs from the underlying distribution with the given mean and standard deviation, but samples are not taken outside the bounds (see "bounded normal" distribution type in [Wyss and Jorgensen, 1998]). This can result in the mean and the standard deviation of the sample data being different from the mean and standard deviation of the underlying distribution. For example, if you are sampling from a normal distribution with a mean of 5 and a standard deviation of 3, but you specify bounds of 1 and 7, the resulting mean of the samples will be around 4.3 and the resulting standard deviation will be around 1.6. This is because you have bounded the original distribution significantly, and asymetrically, since 7 is closer to the original mean than 1.

6.7.2 Lognormal Distribution

If the logarithm of an uncertain variable X has a normal distribution, that is $\log X \sim N(\mu, \sigma)$, then X is distributed with a lognormal distribution. The lognormal is often used to model time to perform some task. It can also be used to model variables which are the product of a large number of other quantities, by the Central Limit Theorem. Finally, the lognormal is used to model quantities which cannot have negative values. Within the lognormal uncertain optional group specification, the number of lognormal uncertain variables, the means, and either standard deviations or error factors must be specified, and the distribution lower and upper bounds and variable descriptors are optional specifications. These distribution bounds can be used to truncate the tails of lognormal distributions, which as for bounded normal, can result in the mean and the standard deviation of the sample data being different from the mean and standard deviation of the underlying distribution (see "bounded lognormal" and "bounded lognormal-n" distribution types in [Wyss and Jorgensen, 1998]).

Description	Keyword	Associated Data	Status	Default
normal uncertain variables	`normal_-uncertain`	integer	Optional group	no normal uncertain variables
normal uncertain means	`means`	list of reals	Required	N/A
normal uncertain standard deviations	`std_-deviations`	list of reals	Required	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Distribution upper bounds	`upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'nuv_i'` where `i = 1,2,3,...`

Table 6.7: Specification detail for normal uncertain variables

For the lognormal variables, one may specify either the mean μ and standard deviation σ of the actual lognormal distribution, the mean μ and error factor ϵ of the actual lognormal distribution, or the mean λ ("lambda") and standard deviation ζ ("zeta") of the underlying normal distribution. The conversion equations from lognormal mean μ and either lognormal error factor ϵ or lognormal standard deviation σ to the mean λ and standard deviation ζ of the underlying normal distribution are as follows:

$$\zeta = \frac{ln(\epsilon)}{1.645}$$

$$\zeta^2 = ln(\frac{\sigma^2}{\mu^2} + 1)$$

$$\lambda = ln(\mu) - \frac{\zeta^2}{2}$$

Conversions from λ and ζ back to μ and ϵ or σ are as follows:

$$\mu = e^{\lambda + \frac{\zeta^2}{2}}$$

$$\sigma^2 = e^{2\lambda + \zeta^2}(e^{\zeta^2} - 1)$$

$$\epsilon = e^{1.645\zeta}$$

The density function for the lognormal distribution is:

$$f(x) = \frac{1}{\sqrt{2\pi}\zeta x} e^{-\frac{1}{2}\left(\frac{lnx - \lambda}{\zeta}\right)^2}$$

6.7.3 Uniform Distribution

Within the uniform uncertain optional group specification, the number of uniform uncertain variables and the distribution lower and upper bounds are required specifications, and variable descriptors is an optional specification.

Description	Keyword	Associated Data	Status	Default
lognormal uncertain variables	`lognormal_-uncertain`	integer	Optional group	no lognormal uncertain variables
lognormal uncertain means	`means`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain standard deviations	`std_-deviations`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain error factors	`error_-factors`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain lambdas	`lambdas`	list of reals	Required (1 of 3 selections)	N/A
lognormal uncertain zetas	`zetas`	list of reals	Required (1 of 3 selections)	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Optional	vector values = `0.`
Distribution upper bounds	`upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'lnuv_i'` where `i = 1,2,3,...`

Table 6.8: Specification detail for lognormal uncertain variables

The uniform distribution has the density function:

$$f(x) = \frac{1}{U_U - L_U}$$

where U_U and L_U are the upper and lower bounds of the uniform distribution, respectively. The mean of the uniform distribution is $\frac{U_U+L_U}{2}$ and the variance is $\frac{(U_U-L_U)^2}{12}$. Note that this distribution is a special case of the more general beta distribution.

Description	Keyword	Associated Data	Status	Default
uniform uncertain variables	`uniform_-uncertain`	integer	Optional group	no uniform uncertain variables
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'uuv_i'` where `i = 1,2,3,...`

Table 6.9: Specification detail for uniform uncertain variables

6.7.4 Loguniform Distribution

If the logarithm of an uncertain variable X has a uniform distribution, that is $\log X \sim U(L_{LU}, U_{LU})$, then X is distributed with a loguniform distribution. Within the loguniform uncertain optional group specification, the number of loguniform uncertain variables and the distribution lower and upper bounds are required specifications, and variable descriptors is an optional specification. The loguniform distribution has the density function:

$$f(x) = \frac{1}{x(lnU_{LU} - lnL_{LU})}$$

Description	Keyword	Associated Data	Status	Default
loguniform uncertain variables	`loguniform_uncertain`	integer	Optional group	no loguniform uncertain variables
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'luuv_i'` where `i = 1,2,3,...`

Table 6.10: Specification detail for loguniform uncertain variables

6.7.5 Triangular Distribution

The triangular distribution is often used when one does not have much data or information, but does have an estimate of the most likely value and the lower and upper bounds. Within the triangular uncertain optional group specification, the number of triangular uncertain variables, the modes, and the distribution lower and upper bounds are required specifications, and variable descriptors is an optional specification.

The density function for the triangular distribution is:

$$f(x) = \frac{2(x - L_T)}{(U_T - L_T)(M_T - L_T)}$$

if $L_T \leq x \leq M_T$, and

$$f(x) = \frac{2(U_T - x)}{(U_T - L_T)(U_T - M_T)}$$

if $M_T \leq x \leq U_T$, and 0 elsewhere. In these equations, L_T is the lower bound, U_T is the upper bound, and M_T is the mode of the triangular distribution.

6.7.6 Exponential Distribution

The exponential distribution is often used for modeling failure rates. Within the exponential uncertain optional group specification, the number of exponential uncertain variables and the beta parameters are required specifications, and variable descriptors is an optional specification.

Description	Keyword	Associated Data	Status	Default
triangular uncertain variables	`triangular_uncertain`	integer	Optional group	no triangular uncertain variables
triangular uncertain modes	`modes`	list of reals	Required	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'tuv_i'` where `i = 1,2,3,...`

Table 6.11: Specification detail for triangular uncertain variables

The density function for the exponential distribution is given by:

$$f(x) = \frac{1}{\beta} e^{\frac{-x}{\beta}}$$

where $\mu_E = \beta$ and $\sigma_E^2 = \beta^2$. Note that this distribution is a special case of the more general gamma distribution.

Description	Keyword	Associated Data	Status	Default
exponential uncertain variables	`exponential_uncertain`	integer	Optional group	no exponential uncertain variables
exponential uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'euv_i'` where `i = 1,2,3,...`

Table 6.12: Specification detail for exponential uncertain variables

6.7.7 Beta Distribution

Within the beta uncertain optional group specification, the number of beta uncertain variables, the alpha and beta parameters, and the distribution upper and lower bounds are required specifications, and the variable descriptors is an optional specification. The beta distribution can be helpful when the actual distribution of an uncertain variable is unknown, but the user has a good idea of the bounds, the mean, and the standard deviation of the uncertain variable. The density function for the beta distribution is

$$f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \frac{(x-L_B)^{\alpha-1}(U_B-x)^{\beta-1}}{(U_B-L_B)^{\alpha+\beta-1}}$$

where $\Gamma(\alpha)$ is the gamma function and $B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$ is the beta function. To calculate mean and standard deviation from the alpha, beta, upper bound, and lower bound parameters of the beta distribution, the following expressions may be used.

$$\mu_B = L_B + \frac{\alpha}{\alpha+\beta}(U_B - L_B)$$

$$\sigma_B^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}(U_B - L_B)^2$$

Solving these for α and β gives:

$$\alpha = (\mu_B - L_B)\frac{(\mu_B - L_B)(U_B - \mu_B) - \sigma_B^2}{\sigma_B^2(U_B - L_B)}$$

$$\beta = (U_B - \mu_B)\frac{(\mu_B - L_B)(U_B - \mu_B) - \sigma_B^2}{\sigma_B^2(U_B - L_B)}$$

Note that the uniform distribution is a special case of this distribution for parameters $\alpha = \beta = 1$.

Description	Keyword	Associated Data	Status	Default
beta uncertain variables	`beta_-uncertain`	integer	Optional group	no beta uncertain variables
beta uncertain alphas	`alphas`	list of reals	Required	N/A
beta uncertain betas	`betas`	list of reals	Required	N/A
Distribution lower bounds	`lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`upper_bounds`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'buv_i'` where `i = 1,2,3,...`

Table 6.13: Specification detail for beta uncertain variables

6.7.8 Gamma Distribution

The gamma distribution is sometimes used to model time to complete a task, such as a repair or service task. It is a very flexible distribution. Within the gamma uncertain optional group specification, the number of gamma uncertain variables and the alpha and beta parameters are required specifications, and variable descriptors is an optional specification.

The density function for the gamma distribution is given by:

$$f(x) = \frac{x^{\alpha-1}e^{\frac{-x}{\beta}}}{\beta^\alpha\Gamma(\alpha)}$$

where $\mu_{GA} = \alpha\beta$ and $\sigma_{GA}^2 = \alpha\beta^2$. Note that the exponential distribution is a special case of this distribution for parameter $\alpha = 1$.

6.7.9 Gumbel Distribution

Within the gumbel optional uncertain group specification, the number of gumbel uncertain variables, and the alpha and beta parameters are required specifications. The Gumbel distribution is also referred to as the Type I

Description	Keyword	Associated Data	Status	Default
gamma uncertain variables	`gamma_-uncertain`	integer	Optional group	no gamma uncertain variables
gamma uncertain alphas	`alphas`	list of reals	Required	N/A
gamma uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'gauv_i'` where `i = 1,2,3,...`

Table 6.14: Specification detail for gamma uncertain variables

Largest Extreme Value distribution. The distribution of maxima in sample sets from a population with a normal distribution will asymptotically converge to this distribution. It is commonly used to model demand variables such as wind loads and flood levels.

The density function for the Gumbel distribution is given by:

$$f(x) = \alpha e^{-\alpha(x-\beta)} exp(-e^{-\alpha(x-\beta)})$$

where $\mu_{GU} = \beta + \frac{0.5772}{\alpha}$ and $\sigma_{GU} = \frac{\pi}{\sqrt{6}\alpha}$.

Description	Keyword	Associated Data	Status	Default
gumbel uncertain variables	`gumbel_-uncertain`	integer	Optional group	no gumbel uncertain variables
gumbel uncertain alphas	`alphas`	list of reals	Required	N/A
gumbel uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'guuv_i'` where `i = 1,2,3,...`

Table 6.15: Specification detail for gumbel uncertain variables

6.7.10 Frechet Distribution

With the frechet uncertain optional group specification, the number of frechet uncertain variables and the alpha and beta parameters are required specifications. The Frechet distribution is also referred to as the Type II Largest Extreme Value distribution. The distribution of maxima in sample sets from a population with a lognormal distribution will asymptotically converge to this distribution. It is commonly used to model non-negative demand variables.

The density function for the frechet distribution is:

$$f(x) = \frac{\alpha}{\beta}(\frac{\beta}{x})^{\alpha+1} e^{-(\frac{\beta}{x})^{\alpha}}$$

where $\mu_F = \beta\Gamma(1 - \frac{1}{\alpha})$ and $\sigma_F^2 = \beta^2[\Gamma(1 - \frac{2}{\alpha}) - \Gamma^2(1 - \frac{1}{\alpha})]$

Description	Keyword	Associated Data	Status	Default
frechet uncertain variables	`frechet_-uncertain`	integer	Optional group	no frechet uncertain variables
frechet uncertain alphas	`alphas`	list of reals	Required	N/A
frechet uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'fuv_i'` where `i = 1,2,3,...`

Table 6.16: Specification detail for frechet uncertain variables

6.7.11 Weibull Distribution

The Weibull distribution is commonly used in reliability studies to predict the lifetime of a device. Within the weibull uncertain optional group specification, the number of weibull uncertain variables and the alpha and beta parameters are required specifications. The Weibull distribution is also referred to as the Type III Smallest Extreme Value distribution. It is also used to model capacity variables such as material strength.

The density function for the weibull distribution is given by:

$$f(x) = \frac{\alpha}{\beta}\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-\left(\frac{x}{\beta}\right)^{\alpha}}$$

where $\mu_W = \beta\Gamma(1+\frac{1}{\alpha})$ and $\sigma_W = \sqrt{\frac{\Gamma(1+\frac{2}{\alpha})}{\Gamma^2(1+\frac{1}{\alpha})} - 1}\mu_W$

Description	Keyword	Associated Data	Status	Default
weibull uncertain variables	`weibull_-uncertain`	integer	Optional group	no weibull uncertain variables
weibull uncertain alphas	`alphas`	list of reals	Required	N/A
weibull uncertain betas	`betas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'wuv_i'` where `i = 1,2,3,...`

Table 6.17: Specification detail for weibull uncertain variables

6.7.12 Histogram Bin Distribution

Histogram uncertain variables are typically used to model a set of empirical data. A bin histogram is a continuous aleatory distribution that allows the user to specify bins of non-zero width (where the uncertain variable may lie) along with the relative frequencies that are associated with each bin.

Within the histogram bin uncertain optional group specification, the number of histogram bin uncertain variables is a required specification, the number of pairs is an optional key for apportionment of abscissas/ordinates/counts, specification of abscissas and either ordinates or counts is required, and the variable descriptors is an optional

specification. When using a histogram bin variable, one must define at least one bin (with two bounding value pairs).

The `abscissas` specification define abscissa values ("x" coordinates) for the PDF of each histogram variable. When paired with `counts`, the specifications provide sets of (x,c) pairs for each histogram variable where `c` defines a count (i.e., a frequency or relative probability) associated with a bin. If using bins of unequal width and specification of probability densities is more natural, then the `counts` specification can be replaced with a `ordinates` specification ("y" coordinates) in order to support interpretation of the input as (x,y) pairs defining the profile of a "skyline" PDF. Conversion between the two specifications is straightforward: a count/frequency is a cumulative probability quantity defined from the product of the ordinate density value and the `x` bin width. Thus, in the cases of bins of equal width, ordinate and count specifications are equivalent. In addition, ordinates and counts may be relative values; it is not necessary to scale them as all user inputs will be normalized.

To fully specify a bin-based histogram with `n` bins (potentially of unequal width), `n+1` (x,c) or (x,y) pairs must be specified with the following features:

- `x` is the parameter value for the left boundary of a histogram bin and `c` is the corresponding count for that bin. Alternatively, `y` defines the ordinate density value for this bin within a skyline PDF. The right boundary of the bin is defined by the left boundary of the next pair.
- the final pair specifies the right end of the last bin and must have a `c` or `y` value of zero.
- the `x` values must be strictly increasing.
- all `c` or `y` values must be positive, except for the last which must be zero.
- a minimum of two pairs must be specified for each bin-based histogram.

The number of pairs specifications provide for the proper association of multiple sets of (x,c) or (x,y) pairs with individual histogram variables. For example, in the following specification

```
histogram_bin_uncertain = 2
  num_pairs =        3           4
  abscissas =  5  8 10 .1 .2 .3 .4
  counts    = 17 21  0 12 24 12  0
```

`num_pairs` associates the first 3 (x,c) pairs from `abscissas` and `counts` ((5,17),(8,21),(10,0)) with one bin-based histogram variable, where one bin is defined between 5 and 8 with a count of 17 and another bin is defined between 8 and 10 with a count of 21. The following set of 4 (x,c) pairs ((.1,12),(.2,24),(.3,12),(.4,0)) defines a second bin-based histogram variable containing three equal-width bins with counts 12, 24, and 12 (middle bin is twice as probable as the other two).

6.7.13 Poisson Distribution

The Poisson distribution is used to predict the number of discrete events that happen in a given time interval. The expected number of occurences in the time interval is λ, which must be a positive real number. For example, if events occur on average 4 times per year and we are interested in the distribution of events over six months, λ would be 2 in this case. However, if we were interested in the distribution of events occuring over 5 years, λ would be 20.

Description	Keyword	Associated Data	Status	Default
histogram bin uncertain variables	`histogram_-bin_-uncertain`	integer	Optional group	no histogram bin uncertain variables
key to apportionment among bin-based histogram variables	`num_pairs`	list of integers	Optional	equal distribution
sets of abscissas for bin-based histogram variables	`abscissas`	list of reals	Required	N/A
sets of ordinates for bin-based histogram variables	`ordinates`	list of reals	Required (1 of 2 selections)	N/A
sets of counts for bin-based histogram variables	`counts`	list of reals	Required (1 of 2 selections)	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'hubv_i'` where `i = 1,2,3,...`

Table 6.18: Specification detail for histogram bin uncertain variables

The density function for the poisson distribution is given by:

$$f(x) = \frac{\lambda e^{-\lambda}}{x!}$$

where λ is the frequency of events happening, and `x` is the number of events that occur. The poisson distribution returns samples representing number of occurrences in the time period of interest.

Description	Keyword	Associated Data	Status	Default
poisson uncertain variables	`poisson_-uncertain`	integer	Optional group	no poisson uncertain variables
poisson uncertain lambdas	`lambdas`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'puv_i'` where `i = 1,2,3,...`

Table 6.19: Specification detail for poisson uncertain variables

6.7.14 Binomial Distribution

The binomial distribution is typically used to predict the number of failures (or defective items or some type of event) in a total of `n` independent tests or trials, where each trial has the probability `p` of failing or being defective. Each particular test can be considered as a Bernoulli trial.

The density function for the binomial distribution is given by:

$$f(x) = \left(\begin{array}{c} n \\ x \end{array} \right) p^x (1-p)^{(n-x)}$$

where `p` is the probability of failure per trial and `n` is the number of trials.

Description	Keyword	Associated Data	Status	Default
binomial uncertain variables	`binomial_-uncertain`	integer	Optional group	no binomial uncertain variables
binomial uncertain prob_per_trial	`prob_per_-trial`	list of reals	Required	N/A
binomial uncertain num_trials	`num_trials`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'biuv_i'` where `i = 1,2,3,...`

Table 6.20: Specification detail for binomial uncertain variables

6.7.15 Negative Binomial Distribution

The negative binomial distribution is typically used to predict the number of times to perform a test to have a total of `n` successes, where each test has a probability `p` of success.

The density function for the negative binomial distribution is given by:

$$f(x) = \binom{n+x-1}{x} p^n (1-p)^x$$

where `p` is the probability of success per trial and `n` is the number of successful trials.

Description	Keyword	Associated Data	Status	Default
negative binomial uncertain variables	`negative_binomial_uncertain`	integer	Optional group	no negative binomial uncertain variables
negative binomial uncertain success prob_per_trial	`prob_per_trial`	list of reals	Required	N/A
negative binomial uncertain success num_trials	`num_trials`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'nbuv_i'` where `i = 1,2,3,...`

Table 6.21: Specification detail for negative binomial uncertain variables

6.7.16 Geometric Distribution

The geometric distribution represents the number of successful trials that might occur before a failure is observed.

The density function for the geometric distribution is given by:

$$f(x) = p(1-p)^x$$

where `p` is the probability of failure per trial.

Description	Keyword	Associated Data	Status	Default
geometric uncertain variables	`geometric_uncertain`	integer	Optional group	no geometric uncertain variables
geometric uncertain prob_per_trial	`prob_per_trial`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'geuv_i'` where `i = 1,2,3,...`

Table 6.22: Specification detail for geometric uncertain variables

6.7.17 Hypergeometric Distribution

The hypergeometric distribution is used to define the number of failures (or the number of successes; the number of some type of event) in a set of tests that has a known proportion of failures. The hypergeometric is often

described using an urn model. For example, say we have a total population containing N balls, and we know that m of the balls are white and the remaining balls are green. If we draw n balls from the urn without replacement, the hypergeometric distribution describes the distribution of the number of white balls drawn from the urn.

The density function for the hypergeometric distribution is given by:

$$f(x) = \frac{\begin{pmatrix} m \\ x \end{pmatrix} \begin{pmatrix} N-m \\ n-x \end{pmatrix}}{\begin{pmatrix} N \\ n \end{pmatrix}}$$

where N is the total population, m is the number of items in the selected population (e.g. the number of white balls in the full urn of N items), and n is the number of balls drawn.

Description	Keyword	Associated Data	Status	Default
hypergeometric uncertain variables	`hypergeometric_uncertain`	integer	Optional group	no hypergeometric uncertain variables
hypergeometric uncertain total_population	`total_population`	list of integers	Required	N/A
hypergeometric uncertain selected_population	`selected_population`	list of integers	Required	N/A
hypergeometric uncertain num_drawn	`num_drawn`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'hguv_i'` where `i = 1,2,3,...`

Table 6.23: Specification detail for hypergeometric uncertain variables

6.7.18 Histogram Point Distribution

As mentioned above, histogram uncertain variables are typically used to model a set of empirical data. A point histogram is a discrete aleatory distribution that allows the user to specify a set of real-valued points and associated frequency values.

Point histograms are similar to Discrete Design Real Set Variables and Discrete State Real Set Variables, but differ in the inclusion of information on the relative probabilities of observing the different values within the set.

Within the histogram point uncertain optional group specification, the number of histogram point uncertain variables is a required specification, the number of pairs is an optional key for apportionment of abscissas and counts, the sets of abscissas and counts are required, and the variable descriptors are optional. When using a histogram point variable, one must define at least one set of abscissa/count pairs. As for Histogram Bin Distribution, the `abscissas` specifications define abscissa values ("x" coordinates) for the PDF of each histogram variable. When paired with `counts`, the specifications provide sets of (x,c) pairs for each histogram variable where `c` defines a count (i.e., a frequency or relative probability) associated with a point.

To fully specify a point-based histogram with `n` points, `n` (x,c) pairs (note that (x,c) and (x,y) are equivalent in this case) must be specified with the following features:

- `x` is the point value and `c` is the corresponding count for that value.
- the `x` values must be strictly increasing.
- all `c` values must be positive.
- a minimum of one pair must be specified for each point-based histogram.

The `num_pairs` specification provides for the proper association of multiple sets of (x,c) or (x,y) pairs with individual histogram variables. For example, in the following specification,

```
histogram_point_uncertain = 2
  num_pairs =   2           3
  abscissas = 3 4 100 200 300
  counts    = 1 1   1   2   1
```

`num_pairs` associates the (x,c) pairs ((3,1),(4,1)) with one point-based histogram variable (where the values 3 and 4 are equally probable) and associates the (x,c) pairs ((100,1),(200,2),(300,1)) with a second point-based histogram variable (where the value 200 is twice as probable as either 100 or 300).

Description	Keyword	Associated Data	Status	Default
histogram point uncertain variables	`histogram_point_uncertain`	integer	Optional group	no histogram point uncertain variables
key to apportionment among point-based histogram variables	`num_pairs`	list of integers	Optional	equal distribution
sets of abscissas for point-based histogram variables	`abscissas`	list of reals	Required	N/A
sets of counts for point-based histogram variables	`counts`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'hupv_i'` where `i = 1,2,3,...`

Table 6.24: Specification detail for histogram point uncertain variables

6.7.19 Correlations

Aleatory uncertain variables may have correlations specified through use of an `uncertain_correlation_matrix` specification. This specification is generalized in the sense that its specific meaning depends on the non-

deterministic method in use. When the method is a nondeterministic sampling method (i.e., `sampling`), then the correlation matrix specifies *rank correlations* [Iman and Conover, 1982]. When the method is instead a reliability (i.e., `local_reliability` or `global_reliability`) or stochastic expansion (i.e., `polynomial_chaos` or `stoch_collocation`) method, then the correlation matrix specifies *correlation coefficients* (normalized covariance) [Haldar and Mahadevan, 2000]. In either of these cases, specifying the identity matrix results in uncorrelated uncertain variables (the default). The matrix input should be symmetric and have all n^2 entries where *n* is the total number of aleatory uncertain variables (all normal, lognormal, uniform, loguniform, triangular, exponential, beta, gamma, gumbel, frechet, weibull, histogram bin, poisson, binomial, negative binomial, geometric, hypergeometric, and histogram point specifications, in that order). Table 7.25 summarizes the specification details:

Description	Keyword	Associated Data	Status	Default
correlations in aleatory uncertain variables	`uncertain_correlation_matrix`	list of reals	Optional	identity matrix (uncorrelated)

Table 6.25: Specification detail for aleatory uncertain correlations

6.8 Epistemic Uncertain Variables

In addition to continuous and discrete aleatory probability distributions, Dakota provides support for continuous and discrete epistemic uncertainties through its continuous interval, discrete interval, and discrete set specifications. Interval-based variables do not represent probability distributions. In the case of continuous intervals, they can specify a single interval per variable which may be used in interval analysis, where the goal is to determine the interval bounds on the output corresponding to the interval bounds on the input. More detailed continuous interval representations can specify a set of belief structures based on intervals that may be contiguous, overlapping, or disjoint. This is used in specifying the inputs necessary for an epistemic uncertainty analysis using Dempster-Shafer theory of evidence. In the discrete case, interval variables may be used to specify categorical choices which are epistemic. For example, if we have three possible forms for a physics model (model 1, 2, or 3) and there is epistemic uncertainty about which one is correct, a discrete uncertain interval or a discrete set may be used to represent this type of uncertainty.

6.8.1 Continuous Interval Uncertain Variable

The continuous interval uncertain variable is NOT a probability distribution. Although it may seem similar to a histogram, the interpretation of this uncertain variable is different. It is used in epistemic uncertainty analysis, where one is trying to model uncertainty due to lack of knowledge. In Dakota, epistemic uncertainty analysis is performed using either interval estimation or Dempster-Shafer theory of evidence. In these approaches, one does not assign a probability distribution to each uncertain input variable. Rather, one divides each uncertain input variable into one or more intervals. The input parameters are only known to occur within intervals; nothing more is assumed. Each interval is defined by its upper and lower bounds, and a Basic Probability Assignment (BPA) associated with that interval. The BPA represents a probability of that uncertain variable being located within that interval. The intervals and BPAs are used to construct uncertainty measures on the outputs called "belief" and "plausibility." Belief represents the smallest possible probability that is consistent with the evidence, while plausibility represents the largest possible probability that is consistent with the evidence. For more information about these approaches, refer to Epistemic Uncertainty Quantification Methods. As an example, in the following specification:

```
continuous_interval_uncertain = 2
  num_intervals  =           3       2
  interval_probs = 0.2 0.5 0.3 0.4 0.6
  lower_bounds   = 2.0 4.0 4.5 1.0 3.0
  upper_bounds   = 2.5 5.0 6.0 5.0 5.0
```

there are 2 interval uncertain variables. The first one is defined by three intervals, and the second by two intervals. The three intervals for the first variable have basic probability assignments of 0.2, 0.5, and 0.3, respectively, while the basic probability assignments for the two intervals for the second variable are 0.4 and 0.6. The basic probability assignments for each interval variable must sum to one. The interval bounds for the first variable are [2, 2.5], [4, 5], and [4.5, 6], and the interval bounds for the second variable are [1.0, 5.0] and [3.0, 5.0]. Note that the intervals can be overlapping or disjoint. Table 7.26 summarizes the specification details for the `interval_uncertain` variable.

Description	Keyword	Associated Data	Status	Default
continuous interval uncertain variables	`continuous_interval_uncertain`	integer	Optional group	no continuous interval uncertain variables
number of intervals defined for each continuous interval variable	`num_intervals`	list of integers	Optional	Equal apportionment of intervals among variables
basic probability assignments per continuous interval	`interval_probabilities`	list of reals	Optional	Equal probability assignments for each interval (1/num_intervals[i])
lower bounds of continuous intervals	`lower_bounds`	list of reals	Required	None
upper bounds of continuous intervals	`upper_bounds`	list of reals	Required	None
Descriptors	`descriptors`	list of strings	Optional	vector of `'ciuv_i'` where `i = 1,2,3,...`

Table 6.26: Specification detail for interval uncertain variables

6.8.2 Discrete Interval Uncertain Variables

Discrete variables may be used to represent things like epistemic model form uncertainty, as mentioned above. For example, if one wants to analyze the effect of model 1 vs. model 2 vs. model 3 in an epistemic analysis (either an interval analysis or a Dempster-Shafer evidence theory analysis), one can use a discrete epistemic variable to represent the uncertainty in the model form. There are three types of discrete epistemic variables, similar to discrete design and discrete state types: a discrete interval variable that may take any integer value within bounds (e.g., [1, 4], allowing values of 1, 2, 3, or 4); a discrete uncertain set of integers, in which the discrete value may take any value within the integer set (for example, the set may be defined as 1, 2, and 4); and a discrete uncertain

set of reals, in which the discrete variable may take any value defined within the real set (for example, a parameter may have two allowable real values, 3.285 or 4.79). The following Tables 7.27 through 7.29 summarize the required and optional specifications for each discrete epistemic variable subtype. Note that, relative to the discrete design and discrete state specifications, discrete epistemic specifications add interval or set probabilities while removing the notion of an initial value.

Description	Keyword	Associated Data	Status	Default
Discrete interval uncertain variables	`discrete_-interval_-uncertain`	integer	Optional group	No discrete interval uncertain variables
Number of intervals defined for each interval variable	`num_-intervals`	list of integers	Optional	Equal apportionment of intervals among variables
Basic probability assignments per interval	`interval_-probabilities`	list of reals	Optional	Equal probability assignments for each interval (1/num_-intervals[i])
Lower bounds	`lower_bounds`	list of integers	Required	None
Upper bounds	`upper_bounds`	list of integers	Required	None
Descriptors	`descriptors`	list of strings	Optional	vector of `'diuv_i'` where `i = 1,2,3,...`

Table 6.27: Specification detail for discrete interval uncertain variables

6.8.3 Discrete Uncertain Integer Set Variables

6.8.4 Discrete Uncertain Real Set Variables

6.9 State Variables

State variables provide a convenient mechanism for managing additional model parameterizations such as mesh density, simulation convergence tolerances, and time step controls. Types include continuous real, discrete range of integer values (contiguous integers), discrete set of integer values, and discrete set of real values. Within each optional state variables specification group, the number of variables is always required. The following Tables 7.30 through 7.33 summarize the required and optional specifications for each state variable subtype. The `initial_-state` specifications provide the initial values for the state variables which will be passed through to the simulator (e.g., in order to define parameterized modeling controls). The remaining specifications are analagous to those for Design Variables.

Description	Keyword	Associated Data	Status	Default
Discrete uncertain set of integer variables	`discrete_-uncertain_-set_integer`	integer	Optional group	no discrete uncertain set integer variables
Number of values for each variable	`num_set_-values`	list of integers	Optional	Equal apportionment of set values among variables
Set values	`set_values`	list of integers	Required	None
Probabilities for each set member	`set_-probabilities`	list of reals	Optional	Equal probability assignments for each set member (1/num_set_-values[i])
Descriptors	`descriptors`	list of strings	Optional	vector of `'dusiv_i'` where `i = 1,2,3,...`

Table 6.28: Specification detail for discrete uncertain set integer variables

Description	Keyword	Associated Data	Status	Default
Discrete state set of real variables	`discrete_-uncertain_-set_real`	integer	Optional group	no discrete uncertain set real variables
Number of values for each variable	`num_set_-values`	list of integers	Optional	Equal apportionment of set values among variables
Set values	`set_values`	list of reals	Required	None
Probabilities for each set member	`set_-probabilities`	list of reals	Optional	Equal probability assignments for each set member (1/num_set_-values[i])
Descriptors	`descriptors`	list of strings	Optional	vector of `'dusrv_i'` where `i = 1,2,3,...`

Table 6.29: Specification detail for discrete uncertain set real variables

Description	Keyword	Associated Data	Status	Default
Continuous state variables	`continuous_-state`	integer	Optional group	No continuous state variables
Initial states	`initial_-state`	list of reals	Optional	vector values = `0.` (repaired to bounds, if required)
Lower bounds	`lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Upper bounds	`upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'csv_i'` where `i = 1,2,3,...`

Table 6.30: Specification detail for continuous state variables

Description	Keyword	Associated Data	Status	Default
Discrete state range variables	`discrete_-state_range`	integer	Optional group	No discrete state variables
Initial states	`initial_-state`	list of integers	Optional	vector values = `0` (repaired to bounds, if required)
Lower bounds	`lower_bounds`	list of integers	Optional	vector values = `INT_MIN`
Upper bounds	`upper_bounds`	list of integers	Optional	vector values = `INT_MAX`
Descriptors	`descriptors`	list of strings	Optional	vector of `'dsriv_i'` where `i = 1,2,3,...`

Table 6.31: Specification detail for discrete state range variables

6.9.1 Continuous State Variables

6.9.2 Discrete State Range Variables

6.9.3 Discrete State Integer Set Variables

Description	Keyword	Associated Data	Status	Default
Discrete state set of integer variables	`discrete_state_set_integer`	integer	Optional group	no discrete state set of integer variables
Initial state	`initial_state`	list of integers	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	`num_set_values`	list of integers	Optional	equal distribution
Set values	`set_values`	list of integers	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'dssiv_i'` where `i = 1,2,3,...`

Table 6.32: Specification detail for discrete state set of integer variables

6.9.4 Discrete State Real Set Variables

Description	Keyword	Associated Data	Status	Default
Discrete state set of real variables	`discrete_state_set_real`	integer	Optional group	no discrete state set of real variables
Initial state	`initial_state`	list of reals	Optional	middle set values (mean indices, rounded down)
Number of values for each variable	`num_set_values`	list of integers	Optional	equal distribution
Set values	`set_values`	list of reals	Required	N/A
Descriptors	`descriptors`	list of strings	Optional	vector of `'dssrv_i'` where `i = 1,2,3,...`

Table 6.33: Specification detail for discrete state set of real variables

Chapter 7

Interface Commands

7.1 Interface Description

The interface section in a Dakota input file specifies how function evaluations will be performed in order to map a set of parameters into a set of responses. Function evaluations are performed using either algebraic mappings, interfaces to simulation codes, or a combination of the two.

When employing algebraic mappings, the AMPL solver library [Gay, 1997] is used to evaluate a directed acyclic graph (DAG) specification from a separate `stub.nl` file. Separate `stub.col` and `stub.row` files are also required to declare the string identifiers of the subset of inputs and outputs, respectively, that will be used in the algebraic mappings.

When employing mappings with simulation codes, the simulations may be available internally or externally to Dakota. The interface invokes the simulation using either system calls, forks, direct function invocations, or computational grid invocations. In the system call and fork cases, the simulation is external to Dakota and communication between Dakota and the simulation occurs through parameter and response files. In the direct function case, the simulation is internal to Dakota and communication occurs through the function parameter list. The direct case can involve linked simulation codes or test functions which are compiled into the Dakota executable. The test functions allow for rapid testing of algorithms without process creation overhead or engineering simulation expense. The grid case is experimental and under development, but is intended to support simulations which are external to Dakota and geographically distributed.

Several examples follow. The first example shows a fork interface specification which specifies the names of the analysis executable and the parameters and results files, and that parameters and responses files will be tagged and saved. Refer to Fork interface for more information on the use of these options.

```
interface,
fork
  analysis_drivers = 'rosenbrock'
  parameters_file  = 'params.in'
  results_file     = 'results.out'
  file_tag
  file_save
```

The next example shows a similar specification, except that an external `rosenbrock` executable has been replaced by use of the internal `rosenbrock` test function from the **DirectApplicInterface** class. Refer to Direct function interface for more information on this specification.

```
interface,
direct
  analysis_drivers = 'rosenbrock'
```

The final example demonstrates an interface employing both algebraic and simulation-based mappings. The results from the individual mappings are overlaid based on the variable and response descriptors used by the individual mappings.

```
interface,
algebraic_mappings = 'ampl/fma.nl'
fork
  analysis_driver = 'text_book'
  parameters_file = 'tb.in'
  results_file    = 'tb.out'
  file_tag
asynchronous
```

7.2 Interface Specification

The interface specification has the following top-level structure:

```
interface,
<interface independent controls>
<algebraic mappings specification>
<simulation interface selection>
  <simulation interface dependent controls>
```

The `<interface independent controls>` are those controls which are valid for all interfaces. Referring to dakota.input.summary, these controls are defined externally from the algebraic mappings and simulation interface selection blocks (before and after). Both the algebraic mappings specification and the simulation interface selection are optional specifications, allowing the use of algebraic mappings alone, simulation-based mappings alone, or a combination. The simulation interface selection blocks are all required group specifications separated by logical OR's, where the interface selection must be `system`, `fork`, `direct`, or `grid`. The `<interface dependent controls>` are those controls which are only meaningful for a specific simulation interface selection. These controls are defined within each interface selection block. Defaults for interface independent and simulation interface dependent controls are defined in **DataInterface**. The following sections provide additional detail on the interface independent controls followed by the algebraic mappings specification, the simulation interface selections, and their corresponding simulation interface dependent controls.

7.3 Interface Independent Controls

The optional set identifier specification uses the keyword `id_interface` to input a string for use in identifying a particular interface specification. A model can then identify the use of this interface by specifying the same string in its `interface_pointer` specification (see Model Commands). For example, a model whose specification contains `interface_pointer = 'I1'` will use an interface specification with `id_interface = 'I1'`. If the `id_interface` specification is omitted, a particular interface specification will be used by a model only if that model omits specifying a `interface_pointer` and if the interface set was the last set parsed (or is the only set parsed). In common practice, if only one interface set exists, then `id_interface` can be safely omitted from the interface specification and `interface_pointer` can be omitted from the model specification(s), since there is no potential for ambiguity in this case.

Table 8.1 summarizes the set identifier interface independent control.

Description	Keyword	Associated Data	Status	Default
Interface set identifier	id_interface	string	Optional	use of last interface parsed

Table 7.1: Specification detail for interface independent controls: set identifier

Table 8.2 summarizes the interface independent controls associated with parallel computing.

Description	Keyword	Associated Data	Status	Default
Asynchronous interface usage	asynchronous	none	Optional group	synchronous interface usage
Asynchronous evaluation concurrency	evaluation_-concurrency	integer	Optional	local: unlimited concurrency, hybrid: no concurrency
Self-schedule local evals	local_-evaluation_-self_-scheduling	none	Optional (1 of 2)	self-scheduled local evals
Static-schedule local evals	local_-evaluation_-static_-scheduling	none	Optional (1 of 2)	self-scheduled local evals
Asynchronous analysis concurrency	analysis_-concurrency	integer	Optional	local: unlimited concurrency, hybrid: no concurrency
Number of evaluation servers	evaluation_-servers	integer	Optional	no override of auto configure
Self scheduling of evaluations	evaluation_-self_-scheduling	none	Optional	no override of auto configure
Static scheduling of evaluations	evaluation_-static_-scheduling	none	Optional	no override of auto configure
Number of analysis servers	analysis_-servers	integer	Optional	no override of auto configure
Self scheduling of analyses	analysis_-self_-scheduling	none	Optional	no override of auto configure
Static scheduling of analyses	analysis_-static_-scheduling	none	Optional	no override of auto configure

Table 7.2: Specification detail for interface independent controls: parallelism

The optional asynchronous flag specifies use of asynchronous protocols (i.e., background system calls, non-blocking forks, POSIX threads) when evaluations or analyses are invoked. The evaluation_concurrency and analysis_concurrency specifications serve a dual purpose:

- when running Dakota on a single processor in `asynchronous` mode, the default concurrency of evaluations and analyses is all concurrency that is available. The `evaluation_concurrency` and `analysis_concurrency` specifications can be used to limit this concurrency in order to avoid machine overload or usage policy violation.

- when running Dakota on multiple processors in message passing mode, the default concurrency of evaluations and analyses on each of the servers is one (i.e., the parallelism is exclusively that of the message passing). With the `evaluation_concurrency` and `analysis_concurrency` specifications, a hybrid parallelism can be selected through combination of message passing parallelism with asynchronous parallelism on each server.

The optional `evaluation_servers` and `analysis_servers` specifications support user overrides of the automatic parallel configuration for the number of evaluation servers and the number of analysis servers. Similarly, the optional `evaluation_self_scheduling`, `evaluation_static_scheduling`, `analysis_self_scheduling`, and `analysis_static_scheduling` specifications can be used to override the automatic parallel configuration of scheduling approach at the evaluation and analysis parallelism levels. That is, if the automatic configuration is undesirable for some reason, the user can enforce a desired number of partitions and a desired scheduling policy at these parallelism levels. Refer to **ParallelLibrary** and the Parallel Computing chapter of the Users Manual [Adams et al., 2010] for additional information.

When performing asynchronous local evaluations, the local evaluation scheduling keywords control how new evaluation jobs are dispatched when one completes. If `local_evaluation_self_scheduling` (default) is specified, each completed evaluation will be replaced by the next in the local evaluation queue. If `local_evaluation_static_scheduling` is specified, each completed evaluation will be replaced by an evaluation number congruent modulo the `evaluation_concurrency`. This is helpful for relative node scheduling as described in `Dakota/examples/parallelism`. For example, assuming only local concurrency (no MPI), if the local concurrency is 7 and job 2 completes, it will be replaced with job 9. This can result in idle processors if runtimes are non-uniform.

7.4 Algebraic mappings

If desired, one can define algebraic input-output mappings using the AMPL code [Fourer et al., 2003] and save these mappings in 3 files: `stub.nl`, `stub.col`, and `stub.row`, where `stub` is a particular root name describing a particular problem. These files names can be communicated to Dakota using the `algebraic_mappings` input. This string may either specify the `stub.nl` filename, or alternatively, just the `stub` itself.

Dakota then uses `stub.col` and `stub.row` to extract the input and output identifier strings and employs the AMPL solver library [Gay, 1997] to process the DAG specification in `stub.nl`. The variable and objective function names declared within AMPL should be a subset of the variable descriptors and response descriptors used by Dakota (see Variables Commands and Response Labels). Ordering is not important, as Dakota will reorder data as needed.

Table 8.3 summarizes the algebraic mappings specification.

Description	Keyword	Associated Data	Status	Default
Algebraic mappings file	`algebraic_mappings`	string	Optional	no algebraic mappings

Table 7.3: Specification detail for algebraic mappings

7.5 Simulation interfaces

Each simulation interface uses one or more simulator programs, and optionally filter programs, to perform the parameter to response mapping. The simulator and filter programs are invoked with system calls, forks, direct function calls, or computational grid invocations. In the system call and fork cases, a separate process is created for the simulator program and files are used for transfer of parameter and response data between Dakota and the simulator program. This approach is simple and reliable and does not require any modification to simulator programs. In the direct function case, subroutine parameter lists are used to pass the parameter and response data. This approach requires modification to simulator programs so that they can be linked into Dakota; however it can be more efficient through the elimination of process creation overhead and deactivation of unnecessary simulator functions (e.g., output), can be less prone to loss of precision in that data can be passed directly rather than written to and read from a file, and can enable completely internal management of multiple levels of parallelism through the use of MPI communicator partitioning. In the grid case, computational grid services are utilized in order to enable distribution of simulations across different computer resources. This capability targets Condor and/or Globus services but is currently experimental and incomplete.

Table 8.4 summarizes the interface independent controls associated with the simulator programs.

Description	Keyword	Associated Data	Status	Default
Analysis drivers	`analysis_drivers`	list of strings	Required	N/A
Additional identifiers for use by the `analysis_drivers`	`analysis_components`	list of strings	Optional	no additional identifiers
Input filter	`input_filter`	string	Optional	no input filter
Output filter	`output_filter`	string	Optional	no output filter
Failure capturing	`failure_capture`	`abort` | `retry` (with integer data) | `recover` (with list of reals data) | `continuation`	Optional group	abort
Feature deactivation	`deactivate`	`active_set_vector`, `evaluation_cache`, and/or `restart_file`	Optional group	Active set vector control, function evaluation cache, and restart file features are active

Table 7.4: Specification detail for simulation interface controls: drivers, filters, failure capturing, and feature management

The required `analysis_drivers` specification provides the names of executable analysis programs or scripts which comprise a function evaluation. The specification can also give values to environment variables that the programs will see; for details, see the subsection on Syntax for Filter and Driver Strings in the Interfaces chapter of the Users Manual [Adams et al., 2010]. The common case of a single analysis driver is simply accommodated by specifying a list of one driver (this also provides backward compatibility with previous Dakota versions). The optional `analysis_components` specification allows the user to provide additional identifiers (e.g., mesh file names) for use by the analysis drivers. This is particularly useful when the same analysis driver is to be reused

multiple times for slightly different analyses. The specific content within the strings is open-ended and can involve whatever syntax is convenient for a particular analysis driver. The number of analysis components n_c should be an integer multiple of the number of drivers n_d, and the first n_c/n_d component strings will be passed to the first driver, etc. The optional `input_filter` and `output_filter` specifications provide the names of separate pre- and post-processing programs or scripts which assist in mapping Dakota parameters files into analysis input files and mapping analysis output files into Dakota results files, respectively. If there is only a single analysis driver, then it is usually most convenient to combine pre- and post-processing requirements into a single analysis driver script and omit the separate input and output filters. However, in the case of multiple analysis drivers, the input and output filters provide a convenient location for non-repeated pre- and post-processing requirements. That is, input and output filters are only executed once per function evaluation, regardless of the number of analysis drivers, which makes them convenient locations for data processing operations that are shared among the analysis drivers.

Failure capturing in interfaces is governed by the optional `failure_capture` specification. Supported directives for mitigating captured failures are `abort` (the default), `retry`, `recover`, and `continuation`. The `retry` selection supports an integer input for specifying a limit on retries, and the `recover` selection supports a list of reals for specifying the dummy function values (only zeroth order information is supported) to use for the failed function evaluation. Refer to the Simulation Code Failure Capturing chapter of the Users Manual [Adams et al., 2010] for additional information.

The optional `deactivate` specification block includes three features which a user may deactivate in order to simplify interface development, increase execution speed, and/or reduce memory and disk requirements:

- Active set vector (ASV) control: deactivation of this feature using a `deactivate active_set_vector` specification allows the user to turn off any variability in ASV values so that active set logic can be omitted in the user's simulation interface. This option trades some efficiency for simplicity in interface development. The default behavior is to request the minimum amount of data required by an algorithm at any given time, which implies that the ASV values may vary from one function evaluation to the next. Since the user's interface must return the data set requested by the ASV values, this interface must contain additional logic to account for any variations in ASV content. Deactivating this ASV control causes Dakota to always request a "full" data set (the full function, gradient, and Hessian data that is available from the interface as specified in the responses specification) on each function evaluation. For example, if ASV control has been deactivated and the responses section specifies four response functions, analytic gradients, and no Hessians, then the ASV on every function evaluation will be { 3 3 3 3 }, regardless of what subset of this data is currently needed. While wasteful of computations in many instances, this simplifies the interface and allows the user to return the same data set on every evaluation. Conversely, if ASV control is active (the default behavior), then the ASV requests in this example might vary from { 1 1 1 1 } to { 2 0 0 2 }, etc., according to the specific data needed on a particular function evaluation. This will require the user's interface to read the ASV requests and perform the appropriate logic in conditionally returning only the data requested. In general, the default ASV behavior is recommended for the sake of computational efficiency, unless interface development time is a critical concern. Note that in both cases, the data returned to Dakota from the user's interface must match the ASV passed in, or else a response recovery error will result. However, when the ASV control is deactivated, the ASV values are invariant and need not be checked on every evaluation. *Note:* Deactivating the ASV control can have a positive effect on load balancing for parallel Dakota executions. Thus, there is significant overlap in this ASV control option with speculative gradients (see Method Independent Controls). There is also overlap with the mode override approach used with certain optimizers (see **SNLLOptimizer** and **SNLLLeastSq**) to combine individual value, gradient, and Hessian requests.

- Function evaluation cache: deactivation of this feature using a `deactivate evaluation_cache` specification allows the user to avoid retention of the complete function evaluation history in memory. This can

be important for reducing memory requirements in large-scale applications (i.e., applications with a large number of variables or response functions) and for eliminating the overhead of searching for duplicates within the function evaluation cache prior to each new function evaluation (e.g., for improving speed in problems with 1000's of inexpensive function evaluations or for eliminating overhead when performing timing studies). However, the downside is that unnecessary computations may be performed since duplication in function evaluation requests may not be detected. For this reason, this option is not recommended when function evaluations are costly. *Note:* duplication detection within Dakota can be deactivated, but duplication detection features within specific optimizers may still be active.

- Restart file: deactivation of this feature using a `deactivate restart_file` specification allows the user to eliminate the output of each new function evaluation to the binary restart file. This can increase speed and reduce disk storage requirements, but at the expense of a loss in the ability to recover and continue a run that terminates prematurely (e.g., due to a system crash or network problem). This option is not recommended when function evaluations are costly or prone to failure. Please note that using the `deactivate restart_file` specification will result in a zero length restart file with the default name `dakota.rst`.

In addition to these simulation interface specifications, the type of interface involves a selection between `system`, `fork`, `direct`, or `grid` required group specifications. The following sections describe these group specifications in detail.

7.5.1 Fork interface

The fork interface is the most common means by which Dakota launches a separate application analysis process. The `fork` keyword anchors the group specification and the `parameters_file`, `results_file`, `verbatim`, `aprepro`, `file_tag`, and `file_save` are additional settings within the group specification. The parameters and results file names are supplied as strings using the `parameters_file` and `results_file` specifications. Both specifications are optional with the default data transfer files being Unix temporary files with system-generated names (e.g., `/usr/tmp/aaaa08861`). The parameters and results file names are passed on the command line to the analysis driver(s) and any specified input/output filters, unless the `verbatim` option is invoked, in which case the driver/filter invocation syntax is used verbatim without command line argument augmentation. For additional information on invocation syntax, see the Interfaces chapter of the Users Manual [Adams et al., 2010]. The format of data in the parameters files can be modified for direct usage with the APREPRO pre-processing tool [Sjaardema, 1992] using the `aprepro` specification (NOTE: the DPrePro preprocessing utility does not require this special formatting). File tagging (appending parameters and results files with the function evaluation number) and file saving (leaving parameters and results files in existence after their use is complete) are controlled with the `file_tag` and `file_save` flags. If these specifications are omitted, the default is no file tagging (no appended function evaluation number) and no file saving (files will be removed after a function evaluation). File tagging is most useful when multiple function evaluations are running simultaneously using files in a shared disk space, and file saving is most useful when debugging the data communication between Dakota and the simulation.

By default Dakota will remove existing results files before invoking the `analysis_driver` to avoid problems created by stale files in the current directory. To override this behavior and not delete existing files, specify `allow_existing_results`.

When performing concurrent evaluations and/or analyses, it is often necessary to cloister input and output files in separate directories to avoid conflicts. When the `work_directory` feature is enabled, Dakota will create a directory for each evaluation/analysis (with optional tagging and saving as with files), and execute the analysis driver from that working directory. If an optional `template_directory` and/or list of `template_files`

is specified, Dakota will link (or copy) those files into each working directory. The specifications for fork and system call interfaces are summarized in Table 8.5.

Description	Keyword	Associated Data	Status	Default
Fork interface	`fork`	none	Required group (1 of 7 selections)	N/A
Parameters file name	`parameters_file`	string	Optional	Unix temp files
Results file name	`results_file`	string	Optional	Unix temp files
Allow existing results files	`allow_existing_results`	none	Optional	results files removed before each evaluation
Verbatim driver/filter invocation syntax	`verbatim`	none	Optional	driver/filter invocation syntax augmented with file names
Aprepro parameters file format	`aprepro`	none	Optional	standard parameters file format
Parameters and results file tagging	`file_tag`	none	Optional	no tagging
Parameters and results file saving	`file_save`	none	Optional	file cleanup
Create work directory	`work_directory`	none	Optional	no work directory
Name of work directory	`named`	string	Optional	workdir
Tag work directory	`directory_tag`	none	Optional	no work directory tagging
Save work directory	`directory_save`	none	Optional	remove work directory
Template directory	`template_directory`	string path	Optional	no template directory
Template files	`template_files`	list of strings	Optional	no template files
Copy template files	`copy`	none	Optional	link template files
Replace existing files	`replace`	none	Optional	do not overwrite files

Table 7.5: Additional specifications for fork (and system call) interfaces

7.5.2 System call interface

The system call interface is included in Dakota for portability and backward compatibility. Users are strongly encouraged to use the fork interface if possible, reverting to system only when necessary. To enable the system call interface, replace the `fork` keyword with `system`. All other keywords have identical meanings to those for the fork interface as summarized previously in Table 8.5.

7.5.3 Direct function interface

For direct function interfaces, `processors_per_analysis` is an additional optional setting within the required group which can be used to specify multiprocessor analysis partitions. As with the `evaluation_servers`, `analysis_servers`, `evaluation_self_scheduling`, `evaluation_static_scheduling`, `analysis_self_scheduling`, and `analysis_static_scheduling` specifications described above in Interface Independent Controls, `processors_per_analysis` provides a means for the user to override the automatic parallel configuration (refer to **ParallelLibrary** and the Parallel Computing chapter of the Users Manual [Adams et al., 2010]) for the number of processors used for each analysis partition. Note that if both `analysis_servers` and `processors_per_analysis` are specified and they are not in agreement, then `analysis_servers` takes precedence. The direct interface specifications are summarized in Table 8.6.

Description	Keyword	Associated Data	Status	Default
Direct function interface	`direct`	none	Required group (1 of 7 selections)	N/A
Number of processors per analysis	`processors_per_analysis`	integer	Optional	no override of auto configure

Table 7.6: Additional specifications for direct function interfaces

The primary use of the direct interface is to invoke internal test functions that perform parameter to response mappings for simple functions as inexpensively as possible. These problems are compiled directly into the Dakota executable as part of the direct function interface class and are used for algorithm testing. Refer to **DirectApplicInterface** for currently available testers.

Dakota supports direct interfaces to a few select simulation codes. One example is ModelCenter, a commercial simulation management framework from Phoenix Integration. To utilize this interface, a user must first define the simulation specifics within a ModelCenter session and then save these definitions to a ModelCenter configuration file. The `analysis_components` specification provides the means to communicate this configuration file to Dakota's ModelCenter interface.

Other direct interfaces to simulation codes include Sandia's SALINAS structural dynamics code, Sandia's SIERRA multiphysics framework, and Sandia's SAGE computational fluid dynamics code, which are available within Sandia and supported to varying degrees.

7.5.4 Matlab, Scilab, and Python interfaces

Dakota supports library-linked interfaces to Matlab, Scilab, and Python scientific computation software, but they must be explicitly enabled when compiling Dakota from source. First consult the Users Manual [Adams et al., 2010] for discussion and examples. Contact the Dakota users mailing list for assistance building and using Dakota with these interfaces. To enable, specify one of the interfaces in Table 8.7. In all these interfaces, the `analysis_driver` is used to specify a Matlab, Scilab, or Python file which implements the parameter to response mapping.

7.5.5 Grid interface

For grid interfaces, no additional specifications are used at this time.

Description	Keyword	Associated Data	Status	Default
Matlab interface	`matlab`	none	Required group (1 of 7 selections)	N/A
Scilab interface	`scilab`	none	Required group (1 of 7 selections)	N/A
Python interface	`python`	none	Required group (1 of 7 selections)	N/A
Python NumPy dataflow	`numpy`	none	Optional	Python list dataflow

Table 7.7: Specification for scientific computing interfaces (specify one)

This capability has been used for interfaces with IDEA and JAVASpaces in the past and is currently a placeholder for future work with Condor and/or Globus services. It is not currently operational. The grid interface specification is summarized in Table 8.8.

Description	Keyword	Associated Data	Status	Default
Grid interface	`grid`	none	Required group (1 of 7 selections)	N/A

Table 7.8: Additional specifications for grid interfaces

Chapter 8

Responses Commands

8.1 Responses Description

Responses specify the data set produced by an interface after the completion of a "function evaluation." Here, the term function evaluation is used loosely to denote a data request from an iterator that is mapped through an interface in a single pass. Strictly speaking, this data request may actually involve multiple response functions and their derivatives, but the term function evaluation is widely used for this purpose. The data set is potentially comprised of a set of functions, their first derivative vectors (gradients), and their second derivative matrices (Hessians). This abstraction provides a generic data container (the **Response** class) whose contents are interpreted differently depending upon the type of iteration being performed. In the case of optimization, the set of functions consists of one or more objective functions, nonlinear inequality constraints, and nonlinear equality constraints. (Linear constraints are not part of a response set since their coefficients can be communicated to an optimizer at start up and then computed internally for all function evaluations; see Method Independent Controls). In the case of least squares iterators, the functions consist of individual residual terms or model responses together with an observed data file for comparison (as opposed to a sum of the squares objective function) as well as nonlinear inequality and equality constraints. In the case of nondeterministic iterators, the function set is made up of generic response functions for which the effect of parameter uncertainty is to be quantified. Parameter study and design of experiments iterators may be used with any of the response data set types. Thus the interpretation of the response data varies from iterator to iterator.

Gradient specification types include none, numerical, analytic, and mixed. The `no_gradients` selection indicates that gradient information is not needed in the study. The `numerical_gradients` selection means that gradient information is needed and will be computed with finite differences by Dakota or the optimization algorithm in use. The `analytic_gradients` selection means that gradient information is available directly from the simulation (finite differencing is not required). And the `mixed_gradients` selection means that some gradient information is available directly from the simulation whereas the rest will have to be estimated with finite differences.

Hessian availability is characterized as none, analytic, numerical, quasi, or mixed. Similar to gradients, the `no_hessians` selection indicates that Hessian information is not needed/available in the study, and the `analytic_hessians` selection indicates that Hessian information is available directly from the simulation. The `numerical_hessians` selection indicates that Hessian information will be estimated with finite differences. The `quasi_hessians` specification means that Hessian information will be accumulated over time using secant updates based on the existing gradient evaluations. Finally, the `mixed_hessians` selection allows

for a mixture of analytic, numerical, and quasi Hessian response data.

Responses specify the *total* data set that is available for use by the method over the course of iteration. This is distinguished from the data *subset* described by an active set vector (see Dakota File Data Formats in the Users Manual [Adams et al., 2010]) indicating the particular subset of the response data needed for a particular function evaluation. Thus, the responses specification is a broad description of the data to be used during a study whereas the active set vector indicates the subset currently needed.

Several examples follow. The first example shows an optimization data set containing an objective function and two nonlinear inequality constraints. These three functions have analytic gradient availability and no Hessian availability.

```
responses,
objective_functions = 1
nonlinear_inequality_constraints = 2
analytic_gradients
no_hessians
```

The next example shows a typical specification for a calibration data set. The six residual functions will have numerical gradients computed using the dakota finite differencing routine with central differences of 0.1% (plus/minus delta value = .001∗value).

```
responses,
calibration_terms = 6
numerical_gradients
  method_source dakota
  interval_type central
  fd_gradient_step_size = .001
no_hessians
```

The last example shows a specification that could be used with a nondeterministic sampling iterator. The three response functions have no gradient or Hessian availability; therefore, only function values will be used by the iterator.

```
responses,
response_functions = 3
no_gradients
no_hessians
```

Parameter study and design of experiments iterators are not restricted in terms of the response data sets which may be catalogued; they may be used with any of the function specification examples shown above.

8.2 Responses Specification

The responses specification has the following structure (see dakota.input.summary):

```
responses,
<set identifier>
<response descriptors>
<function specification>
<gradient specification>
<Hessian specification>
```

The set identifier and response descriptors are optional. However, the function, gradient, and Hessian specifications are all required, their type selected from the options discussed above. For example, the function specification must be one of three types:

- objective and constraint functions
- calibration (least squares) terms and constraint functions
- generic response functions

The following sections describe each of these specification components and their options in additional detail.

8.3 Responses Set Identifier

The optional set identifier specification uses the keyword `id_responses` to input a string for use in identifying a particular responses specification. A model can then identify the use of this response set by specifying the same string in its `responses_pointer` specification (see Model Independent Controls). For example, a model whose specification contains `responses_pointer = 'R1'` will use a responses set with `id_responses = 'R1'`.

If the `id_responses` specification is omitted, a particular responses specification will be used by a model only if that model omits specifying a `responses_pointer` and if the responses set was the last set parsed (or is the only set parsed). In common practice, if only one responses set exists, then `id_responses` can be safely omitted from the responses specification and `responses_pointer` can be omitted from the model specification(s), since there is no potential for ambiguity in this case. Table 9.1 summarizes the set identifier input.

Description	Keyword	Associated Data	Status	Default
Responses set identifier	`id_responses`	string	Optional	use of last responses parsed

Table 8.1: Specification detail for set identifier

8.4 Response Labels

The optional response labels specification `response_descriptors` is a list of strings which will be printed in Dakota output to identify the values for particular response functions. The default descriptor strings use a root string plus a numeric identifier. This root string is `"obj_fn"` for objective functions, `"least_sq_term"` for least squares terms, `"response_fn"` for generic response functions, `"nln_ineq_con"` for nonlinear inequality constraints, and `"nln_eq_con"` for nonlinear equality constraints. Table 9.2 summarizes the response descriptors input.

Description	Keyword	Associated Data	Status	Default
Response labels	`descriptors`	list of strings	Optional	root strings plus numeric identifiers

Table 8.2: Specification detail for response labels

8.5 Function Specification

The function specification must be one of three types: 1) a group containing objective and constraint functions, 2) a group containing calibration (least squares) terms and constraint functions, or 3) a generic response functions specification. These function sets correspond to optimization, least squares, and uncertainty quantification iterators, respectively. Parameter study and design of experiments iterators may be used with any of the three function specifications.

8.5.1 Objective and constraint functions (optimization data set)

An optimization data set is specified using `objective_functions` and optionally `sense`, `primary_scale_types`, `primary_scales`, `weights`, `nonlinear_inequality_constraints`, `lower_bounds`, `upper_bounds`, `nonlinear_equality_constraints`, `targets`, `scale_types`, and `scales`. The `objective_functions`, `nonlinear_inequality_constraints`, and `nonlinear_equality_constraints` inputs specify the number of objective functions, nonlinear inequality constraints, and nonlinear equality constraints, respectively. The number of objective functions must be 1 or greater, and the number of inequality and equality constraints must be 0 or greater. The `sense` specification provides strings for declaring "minimization" or "maximization" (can be shortened to "min" or "max"; not case sensitive) for each of the objective functions, indicating the goal for each objective within an optimization. If a single string is specified it will apply to each objective function. The `primary_scale_types` specification includes strings specifying the scaling type for each objective function value in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `primary_scale_types` may be selected from `'none'`, `'value'`, or `'log'`, to select no, characteristic value, or logarithmic scaling, respectively. Automatic scaling is not available for objective functions. If a single string is specified it will apply to each objective function. Each entry in `primary_scales` may be a user-specified nonzero characteristic value to be used in scaling each objective function. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'log'`. If a single real value is specified it will apply to each function. If the number of objective functions is greater than 1, then a `weights` specification provides a simple weighted-sum approach to combining multiple objectives:

$$f = \sum_{i=1}^{n} w_i f_i$$

If this is not specified, then each objective function is given equal weighting:

$$f = \sum_{i=1}^{n} \frac{f_i}{n}$$

where, in both of these cases, a "minimization" sense will retain a positive weighting for a minimizer and a "maximization" sense will apply a negative weighting. If scaling is specified, it is applied before multi-objective weighted sums are formed.

The `lower_bounds` and `upper_bounds` specifications provide the lower and upper bounds for 2-sided nonlinear inequalities of the form

$$g_l \leq g(x) \leq g_u$$

The defaults for the inequality constraint bounds are selected so that one-sided inequalities of the form

$$g(x) \leq 0.0$$

result when there are no user constraint bounds specifications (this provides backwards compatibility with previous Dakota versions). In a user bounds specification, any upper bound values greater than `+bigRealBoundSize` (1.e+30, as defined in **Minimizer**) are treated as +infinity and any lower bound values less than `-bigRealBoundSize` are treated as -infinity. This feature is commonly used to drop one of the bounds in order to specify a 1-sided constraint (just as the default lower bounds drop out since `-DBL_MAX` < `-bigRealBoundSize`). The same approach is used for nonexistent linear inequality bounds as described in Method Independent Controls and for nonexistent design variable bounds as described in Design Variables.

The `targets` specification provides the targets for nonlinear equalities of the form

$$g(x) = g_t$$

and the defaults for the equality targets enforce a value of `0.` for each constraint

$$g(x) = 0.0$$

The `scale_types` specifications include strings specifying the scaling type for each nonlinear inequality or equality constraint, respectively, in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `scale_types` may be selected from `'none'`, `'value'`, `'auto'`, or `'log'`, to select no, characteristic value, automatic, or logarithmic scaling, respectively. If a single string is specified it will apply to all components of the relevant nonlinear constraint vector. Each entry in `scales` may be a user-specified nonzero characteristic value to be used in scaling each constraint component. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'auto'` and `'log'`. If a single real value is specified it will apply to each constraint.

Any linear constraints present in an application need only be input to an optimizer at start up and do not need to be part of the data returned on every function evaluation (see the linear constraints description in Method Independent Controls). Table 9.3 summarizes the optimization data set specification.

8.5.2 Calibration terms and constraint functions (least squares data set)

A calibration data set is specified using `calibration_terms` and optionally the specifications summarized in Table 9.4 and Table 9.5, including weighting/scaling, data, and constraints. Each of the calibration terms is a residual function to be driven toward zero, and the nonlinear inequality and equality constraint specifications have identical meanings to those described in Objective and constraint functions (optimization data set). These types of problems are commonly encountered in parameter estimation, system identification, and model calibration. Least squares calibration problems are most efficiently solved using special-purpose least squares solvers such as Gauss-Newton or Levenberg-Marquardt; however, they may also be solved using general-purpose optimization algorithms.

While Dakota can solve these problems with either least squares or optimization algorithms, the response data sets to be returned from the simulator are different. Least squares calibration involves a set of residual functions whereas optimization involves a single objective function (sum of the squares of the residuals), i.e.,

$$f = \sum_{i=1}^{n} R_i^2$$

where f is the objective function and the set of R_i are the residual functions. Therefore, function values and derivative data in the least squares case involve the values and derivatives of the residual functions, whereas the optimization case involves values and derivatives of the sum of squares objective function. This means that in the least squares calibration case, the user must return each of `n` residuals separately as a separate calibration term.

Description	Keyword	Associated Data	Status	Default
Number of objective functions	objective_-functions	integer	Required group	N/A
Optimization sense	sense	list of strings	Optional	vector values = 'minimize'
Objective function scaling types	primary_-scale_types	list of strings	Optional	vector values = 'none'
Objective function scales	primary_-scales	list of reals	Optional	vector values = 1. (no scaling)
Multi-objective weightings	weights	list of reals	Optional	equal weightings
Number of nonlinear inequality constraints	nonlinear_-inequality_-constraints	integer	Optional	0
Nonlinear inequality constraint lower bounds	lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Nonlinear inequality constraint upper bounds	upper_bounds	list of reals	Optional	vector values = 0.
Number of nonlinear equality constraints	nonlinear_-equality_-constraints	integer	Optional	0
Nonlinear equality constraint targets	targets	list of reals	Optional	vector values = 0.
Nonlinear constraint scaling types (for inequalities or equalities)	scale_types	list of strings	Optional	vector values = 'none'
Nonlinear constraint scales (for inequalities or equalities)	scales	list of reals	Optional	vector values = 1. (no scaling)

Table 8.3: Specification detail for optimization data sets

Switching between the two approaches sometimes requires different simulation interfaces capable of returning the different granularity of response data required, although Dakota supports automatic recasting of residuals into a sum of squares for presentation to an optimization method. Typically, the user must compute the difference between the model results and the observations when computing the residuals. However, the user has the option of specifying the observational data (e.g. from physical experiments or other sources) in a file. The specification `calibration_data_file` may be used to specify a text file containing `calibration_terms` observed data values (in a supported Dakota tabular format; default formats change in Dakota 5.2 -- see User's Manual) to be used in computing the residuals

$$R_i = y_i^M - y_i^O$$

where *M* denotes model and *O*, observation from the file. In this case the simulator should return the actual model response, as Dakota will compute the residual internally using the supplied data.

The `primary_scale_types` specification includes strings specifying the scaling type for each residual term in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `primary_scale_types` may be selected from `'none'`, `'value'`, or `'log'`, to select no, characteristic value, or logarithmic scaling, respectively. Automatic scaling is not available for calibration terms. If a single string is specified it will apply to each least squares terms. Each entry in `calibration_term_scales` may be a user-specified nonzero characteristic value to be used in scaling each term. These values are ignored for scaling type `'none'`, required for `'value'`, and optional for `'log'`. If a single real value is specified it will apply to each term. The `weights` specification provides a means to specify a relative emphasis among the vector of squared residuals through multiplication of these squared residuals by a vector of weights:

$$f = \sum_{i=1}^{n} w_i R_i^2 = \sum_{i=1}^{n} w_i (y_i^M - y_i^O)^2$$

If characteristic value scaling is additionally specified, then it is applied to each residual prior to squaring:

$$f = \sum_{i=1}^{n} w_i (\frac{y_i^M - y_i^O}{s_i})^2$$

And in the case where experimental data uncertainties are supplied, then the weights are automatically defined to be the inverse of the experimental variance:

$$f = \sum_{i=1}^{n} \frac{1}{\sigma_i^2} (\frac{y_i^M - y_i^O}{s_i})^2$$

8.5.3 Response functions (generic data set)

A generic response data set is specified using `response_functions`. Each of these functions is simply a response quantity of interest with no special interpretation taken by the method in use. This type of data set is used by uncertainty quantification methods, in which the effect of parameter uncertainty on response functions is quantified, and can also be used in parameter study and design of experiments methods (although these methods are not restricted to this data set), in which the effect of parameter variations on response functions is evaluated. Whereas objective, constraint, and residual functions have special meanings for optimization and least squares algorithms, the generic response function data set need not have a specific interpretation and the user is free to define whatever functional form is convenient. Table 9.6 summarizes the generic response function data set specification.

Description	Keyword	Associated Data	Status	Default
Number of calibration terms	calibration_terms	integer	Required	N/A
Calibration data file name	calibration_data_file	string	Optional	none
Experiments (rows) in file	num_experiments	integer	Optional	1
Data file in annotated format	annotated	boolean	Optional	annotated
Data file in freeform format	freeform	boolean	Optional	annotated
Configuration variable columns in file	num_config_variables	integer	Optional	0
Standard deviation columns in file	num_std_deviations	integer	Optional	0
Calibration scaling types	primary_scale_types	list of strings	Optional	vector values = 'none'
Calibration scales	primary_scales	list of reals	Optional	no scaling (vector values = 1.)
Calibration term weights	weights	list of reals	Optional	equal weighting

Table 8.4: Specification detail for nonlinear least squares data sets (calibration terms)

Description	Keyword	Associated Data	Status	Default
Number of nonlinear inequality constraints	nonlinear_inequality_constraints	integer	Optional	0
Nonlinear inequality lower bounds	lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Nonlinear inequality upper bounds	upper_bounds	list of reals	Optional	vector values = 0.
Number of nonlinear equality constraints	nonlinear_equality_constraints	integer	Optional	0
Nonlinear equality targets	targets	list of reals	Optional	vector values = 0.
Nonlinear scaling types (for inequalities or equalities)	scale_types	list of strings	Optional	vector values = 'none'
Nonlinear scales (for inequalities or equalities)	scales	list of reals	Optional	no scaling (vector values = 1.)

Table 8.5: Specification detail for nonlinear least squares data sets (constraints)

Description	Keyword	Associated Data	Status	Default
Number of response functions	response_functions	integer	Required	N/A

Table 8.6: Specification detail for generic response function data sets

8.6 Gradient Specification

The gradient specification must be one of four types: 1) no gradients, 2) numerical gradients, 3) analytic gradients, or 4) mixed gradients.

8.6.1 No gradients

The `no_gradients` specification means that gradient information is not needed in the study. Therefore, it will neither be retrieved from the simulation nor computed with finite differences. The `no_gradients` keyword is a complete specification for this case.

8.6.2 Numerical gradients

The `numerical_gradients` specification means that gradient information is needed and will be computed with finite differences using either the native or one of the vendor finite differencing routines.

The `method_source` setting specifies the source of the finite differencing routine that will be used to compute the numerical gradients: `dakota` denotes Dakota's internal finite differencing algorithm and `vendor` denotes the finite differencing algorithm supplied by the iterator package in use (DOT, CONMIN, NPSOL, NL2SOL, NLSSOL, and OPT++ each have their own internal finite differencing routines). The `dakota` routine is the default since it can execute in parallel and exploit the concurrency in finite difference evaluations (see Exploiting Parallelism in the Users Manual [Adams et al., 2010]). However, the `vendor` setting can be desirable in some cases since certain libraries will modify their algorithm when the finite differencing is performed internally. Since the selection of the `dakota` routine hides the use of finite differencing from the optimizers (the optimizers are configured to accept user-supplied gradients, which some algorithms assume to be of analytic accuracy), the potential exists for the `vendor` setting to trigger the use of an algorithm more optimized for the higher expense and/or lower accuracy of finite-differencing. For example, NPSOL uses gradients in its line search when in user-supplied gradient mode (since it assumes they are inexpensive), but uses a value-based line search procedure when internally finite differencing. The use of a value-based line search will often reduce total expense in serial operations. However, in parallel operations, the use of gradients in the NPSOL line search (user-supplied gradient mode) provides excellent load balancing without need to resort to speculative optimization approaches. In summary, then, the `dakota` routine is preferred for parallel optimization, and the `vendor` routine may be preferred for serial optimization in special cases.

When the `method_source` is `dakota`, the user may also specify the type of scaling desired when determining the finite difference step size. The choices are `absolute`, `bounds`, and `relative`. For `absolute`, the step size will be applied as is. For `bounds`, it will be scaled by the range of each parameter. For `relative`, it will be scaled by the parameter value.

The `interval_type` setting is used to select between `forward` and `central` differences in the numerical gradient calculations. The `dakota`, DOT `vendor`, and OPT++ `vendor` routines have both forward and central differences available, the CONMIN and NL2SOL `vendor` routines support forward differences only, and the NPSOL and NLSSOL `vendor` routines start with forward differences and automatically switch to central differences as the iteration progresses (the user has no control over this). The following forward difference expression

$$\nabla f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

and the following central difference expression

$$\nabla f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}$$

are used to estimate the ith component of the gradient vector.

Lastly, `fd_gradient_step_size` specifies the relative finite difference step size to be used in the computations. Either a single value may be entered for use with all parameters, or a list of step sizes may be entered, one for each parameter. The latter option of a list of step sizes is only valid for use with the Dakota finite differencing routine. For Dakota with an interval scaling type of `absolute`, the differencing interval will be `fd_gradient_step_size`. For Dakota with and interval scaling type of `bounds`, the differencing intervals are computed by multiplying `fd_gradient_step_size` with the range of the parameter. For Dakota (with an interval scaling type of `relative`), DOT, CONMIN, and OPT++, the differencing intervals are computed by multiplying the `fd_gradient_step_size` with the current parameter value. In this case, a minimum absolute differencing interval is needed when the current parameter value is close to zero. This prevents finite difference intervals for the parameter which are too small to distinguish differences in the response quantities being computed. Dakota, DOT, CONMIN, and OPT++ all use `.01*fd_gradient_step_size` as their minimum absolute differencing interval. With a `fd_gradient_step_size = .001`, for example, Dakota, DOT, CONMIN, and OPT++ will use intervals of .001∗current value with a minimum interval of 1.e-5. NPSOL and NLSSOL use a different formula for their finite difference intervals: `fd_gradient_step_size*(1+|current parameter value|)`. This definition has the advantage of eliminating the need for a minimum absolute differencing interval since the interval no longer goes to zero as the current parameter value goes to zero.

When Dakota computes gradients or Hessians by finite differences and the variables in question have bounds, it by default chooses finite-differencing steps that keep the variables within their specified bounds. Older versions of Dakota generally ignored bounds when computing finite differences. To restore the older behavior, one can add keyword `ignore_bounds` to the `response` specification when `method_source dakota` (or just `dakota`) is also specified. In forward difference or backward difference computations, honoring bounds is straightforward. To honor bounds when approximating $\partial f/\partial x_i$, i.e., component i of the gradient of f, by central differences, Dakota chooses two steps h_1 and h_2 with $h_1 \neq h_2$, such that $x + h_1 e_i$ and $x + h_2 e_i$ both satisfy the bounds, and then computes

$$\frac{\partial f}{\partial x_i} \cong \frac{h_2^2(f_1 - f_0) - h_1^2(f_2 - f_0)}{h_1 h_2 (h_2 - h_1)},$$

with $f_0 = f(x)$, $f_1 = f(x + h_1 e_i)$, and $f_2 = f(x + h_2 e_i)$.

Table 9.7 summarizes the numerical gradient specification.

8.6.3 Analytic gradients

The `analytic_gradients` specification means that gradient information is available directly from the simulation (finite differencing is not required). The simulation must return the gradient data in the Dakota format (enclosed in single brackets; see Dakota File Data Formats in the Users Manual [Adams et al., 2010]) for the case of file transfer of data. The `analytic_gradients` keyword is a complete specification for this case.

8.6.4 Mixed gradients

The `mixed_gradients` specification means that some gradient information is available directly from the simulation (analytic) whereas the rest will have to be finite differenced (numerical). This specification allows the user to make use of as much analytic gradient information as is available and then finite difference for the rest.

Description	Keyword	Associated Data	Status	Default
Numerical gradients	numerical_-gradients	none	Required group	N/A
Method source	method_-source	dakota | vendor	Optional group	dakota
Interval scaling type	dakota	absolute | bounds | relative	Optional group	relative
Interval type	interval_-type	forward | central	Optional group	forward
Finite difference step size	fd_-gradient_-step_size	list of reals	Optional	0.001
Ignore variable bounds	ignore_bounds	none	Optional	bounds respected

Table 8.7: Specification detail for numerical gradients

For example, the objective function may be a simple analytic function of the design variables (e.g., weight) whereas the constraints are nonlinear implicit functions of complex analyses (e.g., maximum stress). The id_-analytic_gradients list specifies by number the functions which have analytic gradients, and the id_-numerical_gradients list specifies by number the functions which must use numerical gradients. Each function identifier, from 1 through the total number of functions, must appear once and only once within the union of the id_analytic_gradients and id_numerical_gradients lists. The method_source, interval_type, and fd_gradient_step_size specifications are as described previously in Numerical gradients and pertain to those functions listed by the id_numerical_gradients list. Table 9.8 summarizes the mixed gradient specification.

8.7 Hessian Specification

Hessian availability must be specified with either no_hessians, numerical_hessians, quasi_-hessians, analytic_hessians, or mixed_hessians.

8.7.1 No Hessians

The no_hessians specification means that the method does not require Dakota to manage the computation of any Hessian information. Therefore, it will neither be retrieved from the simulation nor computed by Dakota. The no_hessians keyword is a complete specification for this case. Note that, in some cases, Hessian information may still be being approximated internal to an algorithm (e.g., within a quasi-Newton optimizer such as optpp_-q_newton); however, Dakota has no direct involvement in this process and the responses specification need not include it.

8.7.2 Numerical Hessians

The numerical_hessians specification means that Hessian information is needed and will be computed with finite differences using either first-order gradient differencing (for the cases of analytic_gradients

Description	Keyword	Associated Data	Status	Default
Mixed gradients	mixed_-gradients	none	Required group	N/A
Analytic derivatives function list	id_-analytic_-gradients	list of integers	Required	N/A
Numerical derivatives function list	id_-numerical_-gradients	list of integers	Required	N/A
Method source	method_-source	dakota | vendor	Optional group	dakota
Interval scaling type	dakota	absolute | bounds | relative	Optional group	relative
Interval type	interval_-type	forward | central	Optional group	forward
Finite difference step size	fd_step_size	list of reals	Optional	0.001
Ignore variable bounds	ignore_-bounds	none	Optional	bounds respected

Table 8.8: Specification detail for mixed gradients

or for the functions identified by id_analytic_gradients in the case of mixed_gradients) or first- or second-order function value differencing (all other gradient specifications). In the former case, the following expression

$$\nabla^2 f(\mathbf{x})_i \cong \frac{\nabla f(\mathbf{x} + h\mathbf{e}_i) - \nabla f(\mathbf{x})}{h}$$

estimates the ith Hessian column, and in the latter case, the following expressions

$$\nabla^2 f(\mathbf{x})_{i,j} \cong \frac{f(\mathbf{x} + h_i\mathbf{e}_i + h_j\mathbf{e}_j) - f(\mathbf{x} + h_i\mathbf{e}_i) - f(\mathbf{x} - h_j\mathbf{e}_j) + f(\mathbf{x})}{h_i h_j}$$

and

$$\nabla^2 f(\mathbf{x})_{i,j} \cong \frac{f(\mathbf{x} + h\mathbf{e}_i + h\mathbf{e}_j) - f(\mathbf{x} + h\mathbf{e}_i - h\mathbf{e}_j) - f(\mathbf{x} - h\mathbf{e}_i + h\mathbf{e}_j) + f(\mathbf{x} - h\mathbf{e}_i - h\mathbf{e}_j)}{4h^2}$$

provide first- and second-order estimates of the ij^{th} Hessian term. Prior to Dakota 5.0, Dakota always used second-order estimates. In Dakota 5.0 and newer, the default is to use first-order estimates (which honor bounds on the variables and require only about a quarter as many function evaluations as do the second-order estimates), but specifying central after numerical_hessians causes Dakota to use the old second-order estimates, which do not honor bounds. In optimization algorithms that use Hessians, there is little reason to use second-order differences in computing Hessian approximations.

The fd_hessian_step_size specifies the relative finite difference step size to be used in these differences. Either a single value may be entered for use with all parameters, or a list of step sizes may be entered, one for each parameter. When the interval scaling type is absolute, the differencing intervals are fd_hessian_step_-size. When the interval scaling type is bounds, the differencing intervals are computed by multiplying the fd_hessian_step_size with the range of the parameter. When the interval scaling type is relative, the differencing intervals are computed by multiplying the fd_hessian_step_size with the current parameter

value. A minimum absolute differencing interval of `.01*fd_hessian_step_size` is used when the current parameter value is close to zero. Table 9.9 summarizes the numerical Hessian specification.

Description	Keyword	Associated Data	Status	Default
Numerical Hessians	`numerical_-hessians`	none	Required group	N/A
Interval scaling type	`absolute` | `bounds` | `relative`	`none`	Optional	`relative`
Finite difference step size	`fd_step_size`	list of reals	Optional	`0.001` (1st-order), `0.002` (2nd-order)
Difference order	`forward` | `central`	none	Optional	forward

Table 8.9: Specification detail for numerical Hessians

8.7.3 Quasi Hessians

The `quasi_hessians` specification means that Hessian information is needed and will be approximated using secant updates (sometimes called "quasi-Newton updates", though any algorithm that approximates Newton's method is a quasi-Newton method). Compared to finite difference numerical Hessians, secant approximations do not expend additional function evaluations in estimating all of the second-order information for every point of interest. Rather, they accumulate approximate curvature information over time using the existing gradient evaluations. The supported secant approximations include the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update (specified with the keyword `bfgs`)

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

and the Symmetric Rank 1 (SR1) update (specified with the keyword `sr1`)

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}$$

where B_k is the kth approximation to the Hessian, $s_k = x_{k+1} - x_k$ is the step and $y_k = \nabla f_{k+1} - \nabla f_k$ is the corresponding yield in the gradients. In both cases, an initial scaling of $\frac{y_k^T y_k}{y_k^T s_k} I$ is used for B_0 prior to the first update. In addition, both cases employ basic numerical safeguarding to protect against numerically small denominators within the updates. This safeguarding skips the update if $|y_k^T s_k| < 10^{-6} s_k^T B_k s_k$ in the BFGS case or if $|(y_k - B_k s_k)^T s_k| < 10^{-6} ||s_k||_2 ||y_k - B_k s_k||_2$ in the SR1 case. In the BFGS case, additional safeguarding can be added using the `damped` option, which utilizes an alternative damped BFGS update when the curvature condition $y_k^T s_k > 0$ is nearly violated. Table 9.10 summarizes the quasi Hessian specification.

8.7.4 Analytic Hessians

The `analytic_hessians` specification means that Hessian information is available directly from the simulation. The simulation must return the Hessian data in the Dakota format (enclosed in double brackets; see Dakota

Description	Keyword	Associated Data	Status	Default
Quasi Hessians	quasi_-hessians	bfgs | sr1	Required group	N/A
Numerical safeguarding of BFGS update	damped	none	Optional	undamped BFGS

Table 8.10: Specification detail for quasi Hessians

File Data Formats in Users Manual [Adams et al., 2010]) for the case of file transfer of data. The analytic_-hessians keyword is a complete specification for this case.

8.7.5 Mixed Hessians

The mixed_hessians specification means that some Hessian information is available directly from the simulation (analytic) whereas the rest will have to be estimated by finite differences (numerical) or approximated by secant updating. As for mixed gradients, this specification allows the user to make use of as much analytic information as is available and then estimate/approximate the rest. The id_analytic_hessians list specifies by number the functions which have analytic Hessians, and the id_numerical_hessians and id_quasi_-hessians lists specify by number the functions which must use numerical Hessians and secant Hessian updates, respectively. Each function identifier, from 1 through the total number of functions, must appear once and only once within the union of the id_analytic_hessians, id_numerical_hessians, and id_quasi_-hessians lists. The fd_hessian_step_size and bfgs, damped bfgs, or sr1 secant update selections are as described previously in Numerical Hessians and Quasi Hessians and pertain to those functions listed by the id_numerical_hessians and id_quasi_hessians lists. Table 9.11 summarizes the mixed Hessian specification.

Description	Keyword	Associated Data	Status	Default
Mixed Hessians	`mixed_-hessians`	none	Required group	N/A
Analytic Hessians function list	`id_-analytic_-hessians`	list of integers	Required	N/A
Numerical Hessians function list	`id_-numerical_-hessians`	list of integers	Required	N/A
Interval scaling type	`absolute` | `bounds` | `relative`	`none`	Optional	`relative`
Finite difference step size	`fd_step_size`	list of reals	Optional	`0.001` (1st-order), `0.002` (2nd-order)
Quasi Hessians function list	`id_quasi_-hessians`	list of integers	Required	N/A
Quasi-Hessian update	`bfgs` | `sr1`	none	Required	N/A
Numerical safeguarding of BFGS update	`damped`	none	Optional	undamped BFGS

Table 8.11: Specification detail for mixed Hessians

Chapter 9

Bibliography

- Adams, B.M., Bauman, L.E., Bohnhoff, W.J., Dalbey, K.R., Eddy, J.P., Ebeida, M.S., Eldred, M. S., Hough, P. D., Hu, K.T., Jakeman, J.D., Swiler, L. P., and Vigil, D. M., 2013. "Dakota: A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 5.3.1 Users Manual," Sandia Technical Report SAND2010-2183, updated Jan. 2013.

- Anderson, G., and Anderson, P., 1986 *The UNIX C Shell Field Guide*, Prentice-Hall, Englewood Cliffs, NJ.

- Argaez, M., Tapia, R. A., and Velazquez, L., 2002. "Numerical Comparisons of Path-Following Strategies for a Primal-Dual Interior-Point Method for Nonlinear Programming", *Journal of Optimization Theory and Applications*, Vol. 114 (2).

- Breitung, K., 1984. "Asymptotic approximation for multinormal integrals," *J. Eng. Mech., ASCE*, Vol. 110, No. 3, pp. 357-366.

- Byrd, R. H., Schnabel, R. B., and Schultz, G. A., 1988. "Parallel quasi-Newton Methods for Unconstrained Optimization," *Mathematical Programming*, 42(1988), pp. 273-306.

- Conn, A. R. and Gould, N. I. M. and Toint, P. L., 2000. *Trust-Region Methods*, SIAM-MPS, Philadelphia.

- Dalbey, K. and L. P. Swiler, 2012. "Gaussian Process Adaptive Importance Sampling." submitted to *International Journal for Uncertainty Quantification*.

- Du,Q., V. Faber, and M. Gunzburger, 1999. "Centroidal Voronoi Tessellations: Applications and Algorithms," *SIAM Review*, Volume 41, 1999, pages 637-676.

- Eddy, J. E. and Lewis, K., 2001. "Effective Generation of Pareto Sets using Genetic Programming," Proceedings of ASME Design Engineering Technical Conference.

- El-Bakry, A. S., Tapia, R. A., Tsuchiya, T., and Zhang, Y., 1996. "On the Formulation and Theory of the Newton Interior-Point Method for Nonlinear Programming," *Journal of Optimization Theory and Applications*, (89) pp. 507-541.

- Bichon, B.J., Eldred, M.S., Swiler, L.P., Mahadevan, S., and McFarland, J.M., 2007. "Multimodal Reliability Assessment for Complex Engineering Applications using Efficient Global Optimization," paper AIAA-2007-1946 in *Proceedings of the 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (9th AIAA Non-Deterministic Approaches Conference)*, Honolulu, HI, April 23-26.

- Bichon, B.J., Eldred, M.S., Swiler, L.P., Mahadevan, S., and McFarland, J.M., 2008. "Efficient Global Reliability Analysis for Nonlinear Implicit Performance Functions," *AIAA Journal*, Vol. 46, No. 10, pp. 2459-2468.

- Eldred, M.S. and Swiler, L.P., 2009. "Efficient Algorithms for Mixed Aleatory-Epistemic Uncertainty Quantification with Application to Radiation-Hardened Electronics. Part 1: Algorithms and Benchmark Results," Sandia Technical report SAND2009-5805.

- Eldred, M.S. and Swiler, L.P. and Tang, G., 2011. "Mixed Aleatory-Epistemic Uncertainty Quantification with Stochastic Expansions and Optimization-Based Interval Estimation." *Reliability Engineering and System Safety*, Vol. 96, No. 9, pp. 1092-1113.

- Eldred, M.S. and Dunlavy, D.M., 2006. "Formulations for Surrogate-Based Optimization with Data Fit, Multifidelity, and Reduced-Order Models," paper AIAA-2006-7117 in the *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA, Sept. 6-8.

- Eldred, M.S. and Bichon, B.J., 2006. "Second-Order Reliability Formulations in DAKOTA/UQ," paper AIAA-2006-1828 in *Proceedings of the 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (8th AIAA Non-Deterministic Approaches Conference)*, Newport, Rhode Island, May 1 - 4.

- Eldred, M. S., Giunta, A. A., and Collis, S. S., 2004. "Second-Order Corrections for Surrogate-Based Optimization with Model Hierarchies," *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, Aug. 30 - Sept. 1.

- Eldred, M.S., Agarwal, H., Perez, V.M., Wojtkiewicz, S.F., Jr., and Renaud, J.E., 2004. "Investigation of Reliability Method Formulations in DAKOTA/UQ," *Proceedings of the 9th ASCE Joint Specialty Conference on Probabilistic Mechanics and Structural Reliability*, Albuquerque, NM, July 26-28.

- Gablonsky, J., 2001. "DIRECT version 2.0 Userguide Technical Report CRSC-TR01-08", Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC.

- Fourer, R., Gay, D. M., and Kernighan, B. W., 2003. "AMPL: A Modeling Language for Mathematical Programming," Duxbury Press/Brooks/Cole Publishing Co. 2nd ed.

- Gay, D. M., 1997. "Hooking Your Solver to AMPL," Technical Report 97-4-06, Bell Laboratories, Murray Hill, NJ. Available as `http://www.ampl.com/REFS/hooking2.ps.gz` .

- Gay, David M., 2008. "Specifying and Reading Program Input with NIDR", Sandia Technical Report SAND2008-2261P. Available as `http://www.sandia.gov/~dmgay/nidr08.pdf` .

- Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H., 1986. "User's Guide for NPSOL (Version 4.0): A Fortran Package for Nonlinear Programming," System Optimization Laboratory Technical Report SOL-86-2, Stanford University, Stanford, CA.

- Giunta, A. A., Swiler, L.P., Brown, S.L, Eldred, M.S., Richards, M.D., and Cyr, E.C., 2006. "The Surfpack Software Library for Surrogate Modeling of Sparse, Irregularly Spaced Multidimensional Data," paper AIAA-2006-7049 in *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA.

- Gunburger, M. and J. Burkardt, 2004. "Uniformity Measures for Point Samples in Hypercubes." Available on John Burkardt's web site: http://www.csit.fsu.edu/~burkardt/

- Haftka, R. T., 1991. "Combining Global and Local Approximations," *AIAA Journal*, Vol. 29, No. 9, pp. 1523-1525.

- Haldar, A., and Mahadevan, S., 2000. *Probability, Reliability, and Statistical Methods in Engineering Design*, John Wiley and Sons, New York.

- Halton, J. H. "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numerische Mathematik*,Volume 2 pages 84-90.

- Halton, J. H. and G. B. Smith, 1964. Algorithm 247: Radical-Inverse Quasi-Random Point Sequence, *Communications of the ACM*, Volume 7, pages 701-702.

- Hart, W. E., Giunta, A. A., Salinger, A. G., and van Bloemen Waanders, B. G., 2001. "An Overview of the Adaptive Pattern Search Algorithm and its Application to Engineering Optimization Problems," abstract in *Proceedings of the McMaster Optimization Conference: Theory and Applications*, McMaster University, Hamilton, Ontario, Canada.

- Hart, W. E., and Hunter, K. O., 1999. "A Performance Analysis of Evolutionary Pattern Search with Generalized Mutation Steps," *Proc Conf Evolutionary Computation*, pp. 672-679.

- Helton, J.C. and W.L. Oberkampf, 2004. Special Issue of Reliability Engineering and System Safety: Issue on Alternative Representations of Epistemic Uncertainty. Vol 85, Nos. 1-3, July-Sept. 2004.

- Hohenbichler, M. and Rackwitz, R. "Improvement of second-order reliability estimates by importance sampling," *ASCE Journal of Engineering Mechanics*, Vol. 114, No. 12, 1988, pp. 2195-2199.

- Hong, H.P. "Simple Approximations for Improving Second-Order Reliability Estimates," *ASCE Journal of Engineering Mechanics*, Vol. 125, No. 5, 1999, pp. 592-595.

- Gray, G. A. and Kolda, T. G., 2006. "Algorithm 856: APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization," *ACM Transactions on Mathematical Software*, Vol. 32, No. 3, pp. 485-507.

- Iman, R. L., and Conover, W. J., 1982. "A Distribution-Free Approach to Inducing Rank Correlation Among Input Variables," *Communications in Statistics: Simulation and Computation*, Vol. B11, no. 3, pp. 311-334.

- Jones, D., Schonlau, M., and W. Welch, 1998. "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, Vol. 13, pp. 455-492.

- Kocis, L. and W. Whiten, 1997. "Computational Investigations of Low-Discrepancy Sequences," *ACM Transactions on Mathematical Software*, Volume 23, Number 2, 1997, pages 266-294.

- Lewis, R. M., and Nash, S. G., 2000. "A Multigrid Approach to the Optimization of Systems Governed by Differential Equations," paper AIAA-2000-4890 in *Proceedings of the 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Long Beach, CA, Sept. 6-8.

- Meza, J. C., Oliva, R. A., Hough, P. D., and Williams, P. J., 2007. "OPT++: An Object-Oriented Toolkit for Nonlinear Optimization," *ACM Transactions on Mathematical Software* 33(2).

- More, J., and Thuente, D., 1994. "Line Search Algorithms with Guaranteed Sufficient Decrease," *ACM Transactions on Mathematical Software* 20(3):286-307.

- Morris, M. D., "Factorial Sampling Plans for Preliminary Computational Experiments," *Technometrics* 33(2): 161-174.

- Oberkampf, W.L. and J.C. Helton, 2003. "Evidence Theory for Engineering Applications." Sandia National Laboratories Technical Report SAND2003-3559P.

- Perez, V. M., J. E. Renaud, L. T. Watson, 2004. "An Interior-Point Sequetial Approximation Optimization Methodology", *Structural and Multidisciplinary Optimization*, 27(5):360-370.

- Plantenga, T. D., 2009. "HOPSPACK 2.0 User Manual", Sandia National Laboratories Technical Report Sandia National Laboratories Technical Report SAND2009-6265.

- Robinson, D.G. and C. Atcitty, 1999. "Comparison of Quasi- and Pseudo-Monte Carlo Sampling for Reliability and Uncertainty Analysis." *Proceedings of the AIAA Probabilistic Methods Conference*, St. Louis MO, AIAA99-1589.

- Saltelli, A., S. Tarantola, F. Campolongo, and M. Ratto, 2004. "Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models." John Wiley & Sons.

- Schittkowski, K., 2004. "NLPQLP: A Fortran Implementation of a Sequential Quadratic Programming Algorithm with Distributed and Non-Monotone Line Search -- User's Guide," Technical Report, Department of Mathematics, University of Bayreuth, Bayreuth, Germany.

- Sjaardema, G. D., 1992. "APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses," Sandia National Laboratories Technical Report SAND92-2291, Albuquerque, NM.

- Srinivasan, R.,2002. "Importance Sampling." Springer-Verlag.

- Stroud, A. H., 1971. "Approximate Calculation of Multiple Integrals," Prentice Hall.

- Swiler, L. P. and N. J. West, 2010. "Importance Sampling: Promises and Limitations." *Proceedings of the 12th AIAA Non-Deterministic Approaches Conference* , Orlando, FL. Paper AIAA-2010-2850.

- Tapia, R. A., and Argaez, M., "Global Convergence of a Primal-Dual Interior-Point Newton Method for Nonlinear Programming Using a Modified Augmented Lagrangian Function". (In Preparation).

- Tong, C. H., 2005. "The PSUADE Software Library." Lawrence Livermore National Laboratories UCRL UCRL-WEB-211742. Web site: http://www.llnl.gov/CASC/uncertainty_-quantification/#psuade.

- Vanderbei, R. J., and Shanno, D. F., 1999. "An interior-point algorithm for nonconvex nonlinear programming", *Computational Optimization and Applications*, 13:231-259.

- Vanderplaats, G. N., 1973. "CONMIN - A FORTRAN Program for Constrained Function Minimization," NASA TM X-62282. (see also: Addendum to Technical Memorandum, 1978).

- Vanderplaats Research and Development, Inc., 1995. "DOT Users Manual, Version 4.20," Colorado Springs.

- Weirs, V. G. and Kamm, J. R. and Swiler, L. P. and Ratto, M. and Tarantola, S. and Adams, B. M. and Rider, W. J. and Eldred, M. S, 2010. "Sensitivity Analysis Techniques Applied to a System of Hyperbolic Conservation Laws." Submitted to Reliability Engineering and System Safety, Oct. 2010.

- Wright, S. J., 1997. "Primal-Dual Interior-Point Methods", SIAM.

- Wyss, G. D., and Jorgensen, K. H., 1998. "A User s Guide to LHS: Sandia's Latin Hypercube Sampling Software," Sandia National Laboratories Technical Report SAND98-0210, Albuquerque, NM.

- Xiu, D., 2008. "Numerical integration formulas of degree two," Applied Numerical Mathematics, Vol. 58, pp. 1515-1520.

- Xu, S. and Grandhi, R. V., 1998. "Effective Two-Point Function Approximation for Design Optimization," AIAA Journal, Vol. 36, No. 12, December 1998.

Chapter 10

File Documentation

10.1 dakota.input.summary File Reference

File containing the input specification for Dakota.

10.1.1 Detailed Description

File containing the input specification for Dakota. This file is derived automatically from dakota.input.nspec, which is used in the generation of parser system files that are compiled into the Dakota executable. Therefore, these files are the definitive source for input syntax, capability options, and associated data inputs. Refer to **Instructions for Modifying Dakota's Input Specification** for information on how to modify the input specification and propagate the changes through the parsing system.

Key features of the input specification and the associated user input files include:

- In the input specification, required individual specifications simply appear, optional individual and group specifications are enclosed in [], required group specifications are enclosed in (), and either-or relationships are denoted by the | symbol. These symbols only appear in dakota.input.nspec; they must not appear in actual user input files.

- Keyword specifications (i.e., `strategy`, `method`, `model`, `variables`, `interface`, and `responses`) begin with the keyword possibly preceded by white space (blanks, tabs, and newlines) both in the input specifications and in user input files. For readability, keyword specifications may be spread across several lines. Earlier versions of Dakota (prior to 4.1) required a backslash character (\) at the ends of intermediate lines of a keyword. While such backslashes are still accepted, they are no longer required.

- Some of the keyword components within the input specification indicate that the user must supply `INTEGER`, `REAL`, `STRING`, `INTEGERLIST`, `REALLIST`, or `STRINGLIST` data as part of the specification. In a user input file, the `"="` is optional, data in a `LIST` can be separated by commas or whitespace, and the `STRING` data are enclosed in single or double quotes (e.g., `'text_book'` or "text_book").

- In user input files, input is largely order-independent (except for entries in lists of data), case insensitive,

and white-space insensitive. Although the order of input shown in the Sample dakota.in Files generally follows the order of options in the input specification, this is not required.

- In user input files, specifications may be abbreviated so long as the abbreviation is unique. For example, the `npsol_sqp` specification within the method keyword could be abbreviated as `npsol`, but `dot_sqp` should not be abbreviated as `dot` since this would be ambiguous with other DOT method specifications.

- In both the input specification and user input files, comments are preceded by `#`.

The dakota.input.summary file for Dakota version 5.1 is:

```
KEYWORD01 strategy
        [ graphics ]
        [ tabular_graphics_data
          [ tabular_graphics_file STRING ]
          ]
        [ output_precision INTEGER >= 0 ]
        [ results_output
          [ results_output_file STRING ]
          ]
        [ iterator_servers INTEGER ]
        [ iterator_self_scheduling ]
        [ iterator_static_scheduling ]
        ( hybrid
          ( sequential ALIAS uncoupled
            method_list STRINGLIST
            )
          |
          ( embedded ALIAS coupled
            global_method_pointer STRING
            local_method_pointer STRING
            [ local_search_probability REAL ]
            )
          |
          ( collaborative
            method_list STRINGLIST
            )
          )
        |
        ( multi_start
          method_pointer STRING
          [ random_starts INTEGER
            [ seed INTEGER ]
            ]
          [ starting_points REALLIST ]
          )
        |
        ( pareto_set
          method_pointer ALIAS opt_method_pointer STRING
          [ random_weight_sets INTEGER
            [ seed INTEGER ]
            ]
          [ weight_sets ALIAS multi_objective_weight_sets REALLIST ]
          )
        |
        ( single_method
          [ method_pointer STRING ]
          )
```

```
KEYWORD12 method
        [ id_method STRING ]
        [ model_pointer STRING ]
        [ output
          debug
          | verbose
          | normal
          | quiet
          | silent
          ]
        [ max_iterations INTEGER >= 0 ]
        [ max_function_evaluations INTEGER >= 0 ]
        [ speculative ]
        [ convergence_tolerance REAL ]
        [ constraint_tolerance REAL ]
        [ scaling ]
        [ final_solutions INTEGER >= 0 ]
        ( surrogate_based_local
          approx_method_name STRING
          | approx_method_pointer STRING
          [ soft_convergence_limit INTEGER ]
          [ truth_surrogate_bypass ]
          [ trust_region
            [ initial_size REAL ]
            [ minimum_size REAL ]
            [ contract_threshold REAL ]
            [ expand_threshold REAL ]
            [ contraction_factor REAL ]
            [ expansion_factor REAL ]
            ]
          [ approx_subproblem
            original_primary
            | single_objective
            | augmented_lagrangian_objective
            | lagrangian_objective
            original_constraints
            | linearized_constraints
            | no_constraints
            ]
          [ merit_function
            penalty_merit
            | adaptive_penalty_merit
            | lagrangian_merit
            | augmented_lagrangian_merit
            ]
          [ acceptance_logic
            tr_ratio
            | filter
            ]
          [ constraint_relax
            homotopy
            ]
          [ linear_inequality_constraint_matrix REALLIST ]
          [ linear_inequality_lower_bounds REALLIST ]
          [ linear_inequality_upper_bounds REALLIST ]
          [ linear_inequality_scale_types STRINGLIST ]
          [ linear_inequality_scales REALLIST ]
          [ linear_equality_constraint_matrix REALLIST ]
          [ linear_equality_targets REALLIST ]
          [ linear_equality_scale_types STRINGLIST ]
          [ linear_equality_scales REALLIST ]
          )
```

```
|
( dot_frcg
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
| dot_mmfd
| dot_bfgs
| dot_slp
| dot_sqp
|
( dot
  frcg
  | mmfd
  | bfgs
  | slp
  | sqp
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
|
( conmin_frcg
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
| conmin_mfd
|
( conmin
  frcg
  | mfd
  [ linear_inequality_constraint_matrix REALLIST ]
  [ linear_inequality_lower_bounds REALLIST ]
  [ linear_inequality_upper_bounds REALLIST ]
  [ linear_inequality_scale_types STRINGLIST ]
  [ linear_inequality_scales REALLIST ]
  [ linear_equality_constraint_matrix REALLIST ]
  [ linear_equality_targets REALLIST ]
  [ linear_equality_scale_types STRINGLIST ]
  [ linear_equality_scales REALLIST ]
  )
|
( dl_solver STRING
```

```
      [ linear_inequality_constraint_matrix REALLIST ]
      [ linear_inequality_lower_bounds REALLIST ]
      [ linear_inequality_upper_bounds REALLIST ]
      [ linear_inequality_scale_types STRINGLIST ]
      [ linear_inequality_scales REALLIST ]
      [ linear_equality_constraint_matrix REALLIST ]
      [ linear_equality_targets REALLIST ]
      [ linear_equality_scale_types STRINGLIST ]
      [ linear_equality_scales REALLIST ]
      )
    |
    ( npsol_sqp
      [ verify_level INTEGER ]
      [ function_precision REAL ]
      [ linesearch_tolerance REAL ]
      [ linear_inequality_constraint_matrix REALLIST ]
      [ linear_inequality_lower_bounds REALLIST ]
      [ linear_inequality_upper_bounds REALLIST ]
      [ linear_inequality_scale_types STRINGLIST ]
      [ linear_inequality_scales REALLIST ]
      [ linear_equality_constraint_matrix REALLIST ]
      [ linear_equality_targets REALLIST ]
      [ linear_equality_scale_types STRINGLIST ]
      [ linear_equality_scales REALLIST ]
      )
    | nlssol_sqp
    |
    ( stanford
      npsol
      | nlssol
      [ verify_level INTEGER ]
      [ function_precision REAL ]
      [ linesearch_tolerance REAL ]
      [ linear_inequality_constraint_matrix REALLIST ]
      [ linear_inequality_lower_bounds REALLIST ]
      [ linear_inequality_upper_bounds REALLIST ]
      [ linear_inequality_scale_types STRINGLIST ]
      [ linear_inequality_scales REALLIST ]
      [ linear_equality_constraint_matrix REALLIST ]
      [ linear_equality_targets REALLIST ]
      [ linear_equality_scale_types STRINGLIST ]
      [ linear_equality_scales REALLIST ]
      )
    |
    ( nlpql_sqp
      [ linear_inequality_constraint_matrix REALLIST ]
      [ linear_inequality_lower_bounds REALLIST ]
      [ linear_inequality_upper_bounds REALLIST ]
      [ linear_inequality_scale_types STRINGLIST ]
      [ linear_inequality_scales REALLIST ]
      [ linear_equality_constraint_matrix REALLIST ]
      [ linear_equality_targets REALLIST ]
      [ linear_equality_scale_types STRINGLIST ]
      [ linear_equality_scales REALLIST ]
      )
    |
    ( optpp_cg
      [ max_step REAL ]
      [ gradient_tolerance REAL ]
      [ linear_inequality_constraint_matrix REALLIST ]
      [ linear_inequality_lower_bounds REALLIST ]
      [ linear_inequality_upper_bounds REALLIST ]
```

```
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( optpp_q_newton
    | optpp_fd_newton
    | optpp_g_newton
    | optpp_newton
    [ search_method
      value_based_line_search
      | gradient_based_line_search
      | trust_region
      | tr_pds
      ]
    [ merit_function
      el_bakry
      | argaez_tapia
      | van_shanno
      ]
    [ steplength_to_boundary REAL ]
    [ centering_parameter REAL ]
    [ max_step REAL ]
    [ gradient_tolerance REAL ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( optpp_pds
    [ search_scheme_size INTEGER ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( asynch_pattern_search ALIAS coliny_apps
    [ initial_delta REAL ]
    [ contraction_factor REAL ]
    [ threshold_delta REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ synchronization
      blocking
      | nonblocking
      ]
    [ merit_function
      merit_max
```

```
      | merit_max_smooth
      | merit1
      | merit1_smooth
      | merit2
      | merit2_smooth
      | merit2_squared
      ]
    [ constraint_penalty REAL ]
    [ smoothing_factor REAL ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( moga
    [ fitness_type
      layer_rank
      | domination_count
      ]
    [ replacement_type
      elitist
      | roulette_wheel
      | unique_roulette_wheel
      |
      ( below_limit REAL
        [ shrinkage_fraction ALIAS shrinkage_percentage REAL ]
        )
      ]
    [ niching_type
      radial REALLIST
      | distance REALLIST
      |
      ( max_designs REALLIST
        [ num_designs INTEGER >= 2 ]
        )
      ]
    [ convergence_type
      metric_tracker
      [ percent_change REAL ]
      [ num_generations INTEGER >= 0 ]
      ]
    [ postprocessor_type
      orthogonal_distance REALLIST
      ]
    [ population_size INTEGER >= 0 ]
    [ log_file STRING ]
    [ print_each_pop ]
    [ initialization_type
      simple_random
      | unique_random
      | flat_file STRING
      ]
    [ crossover_type
      multi_point_binary INTEGER
      | multi_point_parameterized_binary INTEGER
      | multi_point_real INTEGER
```

```
      |
      ( shuffle_random
        [ num_parents INTEGER > 0 ]
        [ num_offspring INTEGER > 0 ]
        )
      [ crossover_rate REAL ]
      ]
    [ mutation_type
      bit_random
      | replace_uniform
      |
      ( offset_normal
        | offset_cauchy
        | offset_uniform
        [ mutation_scale REAL ]
        )
      [ mutation_rate REAL ]
      ]
    [ seed INTEGER > 0 ]
    [ linear_inequality_constraint_matrix REALLIST ]
    [ linear_inequality_lower_bounds REALLIST ]
    [ linear_inequality_upper_bounds REALLIST ]
    [ linear_inequality_scale_types STRINGLIST ]
    [ linear_inequality_scales REALLIST ]
    [ linear_equality_constraint_matrix REALLIST ]
    [ linear_equality_targets REALLIST ]
    [ linear_equality_scale_types STRINGLIST ]
    [ linear_equality_scales REALLIST ]
    )
  |
  ( soga
    [ fitness_type
      merit_function
      [ constraint_penalty REAL ]
      ]
    [ replacement_type
      elitist
      | favor_feasible
      | roulette_wheel
      | unique_roulette_wheel
      ]
    [ convergence_type
      ( best_fitness_tracker
        [ percent_change REAL ]
        [ num_generations INTEGER >= 0 ]
        )
      |
      ( average_fitness_tracker
        [ percent_change REAL ]
        [ num_generations INTEGER >= 0 ]
        )
      ]
    [ population_size INTEGER >= 0 ]
    [ log_file STRING ]
    [ print_each_pop ]
    [ initialization_type
      simple_random
      | unique_random
      | flat_file STRING
      ]
    [ crossover_type
      multi_point_binary INTEGER
```

```
        | multi_point_parameterized_binary INTEGER
        | multi_point_real INTEGER
        |
        ( shuffle_random
          [ num_parents INTEGER > 0 ]
          [ num_offspring INTEGER > 0 ]
          )
        [ crossover_rate REAL ]
        ]
      [ mutation_type
        bit_random
        | replace_uniform
        |
        ( offset_normal
          | offset_cauchy
          | offset_uniform
          [ mutation_scale REAL ]
          )
        [ mutation_rate REAL ]
        ]
      [ seed INTEGER > 0 ]
      [ linear_inequality_constraint_matrix REALLIST ]
      [ linear_inequality_lower_bounds REALLIST ]
      [ linear_inequality_upper_bounds REALLIST ]
      [ linear_inequality_scale_types STRINGLIST ]
      [ linear_inequality_scales REALLIST ]
      [ linear_equality_constraint_matrix REALLIST ]
      [ linear_equality_targets REALLIST ]
      [ linear_equality_scale_types STRINGLIST ]
      [ linear_equality_scales REALLIST ]
      )
    |
    ( coliny_pattern_search
      [ constant_penalty ]
      [ no_expansion ]
      [ expand_after_success INTEGER ]
      [ pattern_basis
        coordinate
        | simplex
        ]
      [ stochastic ]
      [ total_pattern_size INTEGER ]
      [ exploratory_moves
        multi_step
        | adaptive_pattern
        | basic_pattern
        ]
      [ synchronization
        blocking
        | nonblocking
        ]
      [ contraction_factor REAL ]
      [ constraint_penalty REAL ]
      [ initial_delta REAL ]
      [ threshold_delta REAL ]
      [ solution_target ALIAS solution_accuracy REAL ]
      [ seed INTEGER > 0 ]
      [ show_misc_options ]
      [ misc_options STRINGLIST ]
      )
    |
    ( coliny_solis_wets
```

```
    [ contract_after_failure INTEGER ]
    [ no_expansion ]
    [ expand_after_success INTEGER ]
    [ constant_penalty ]
    [ contraction_factor REAL ]
    [ constraint_penalty REAL ]
    [ initial_delta REAL ]
    [ threshold_delta REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER > 0 ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_cobyla
    [ initial_delta REAL ]
    [ threshold_delta REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER > 0 ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_direct
    [ division
      major_dimension
      | all_dimensions
      ]
    [ global_balance_parameter REAL ]
    [ local_balance_parameter REAL ]
    [ max_boxsize_limit REAL ]
    [ min_boxsize_limit REAL ]
    [ constraint_penalty REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER > 0 ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_ea
    [ population_size INTEGER > 0 ]
    [ initialization_type
      simple_random
      | unique_random
      | flat_file STRING
      ]
    [ fitness_type
      linear_rank
      | merit_function
      ]
    [ replacement_type
      random INTEGER
      | chc INTEGER
      | elitist INTEGER
      [ new_solutions_generated INTEGER ]
      ]
    [ crossover_rate REAL ]
    [ crossover_type
      two_point
      | blend
      | uniform
      ]
```

```
    [ mutation_rate REAL ]
    [ mutation_type
      replace_uniform
      |
      ( offset_normal
        | offset_cauchy
        | offset_uniform
        [ mutation_scale REAL ]
        [ mutation_range INTEGER ]
        )
      [ non_adaptive ]
      ]
    [ constraint_penalty REAL ]
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER > 0 ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( coliny_beta
    beta_solver_name STRING
    [ solution_target ALIAS solution_accuracy REAL ]
    [ seed INTEGER > 0 ]
    [ show_misc_options ]
    [ misc_options STRINGLIST ]
    )
  |
  ( nl2sol
    [ function_precision REAL ]
    [ absolute_conv_tol REAL ]
    [ x_conv_tol REAL ]
    [ singular_conv_tol REAL ]
    [ singular_radius REAL ]
    [ false_conv_tol REAL ]
    [ initial_trust_radius REAL ]
    [ covariance INTEGER ]
    [ regression_diagnostics ]
    )
  |
  ( nonlinear_cg
    [ misc_options STRINGLIST ]
    )
  |
  ( surrogate_based_global
    approx_method_name STRING
    | approx_method_pointer STRING
    [ replace_points ]
    )
  |
  ( efficient_global
    [ gaussian_process ALIAS kriging
      surfpack
      | dakota
      ]
    [ use_derivatives ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ seed INTEGER > 0 ]
    )
  |
```

```
( polynomial_chaos ALIAS nond_polynomial_chaos
  [ p_refinement
    uniform
    |
    ( dimension_adaptive
      sobol
      | decay
      | generalized
      )
    ]
  [ askey
  | wiener ]
  ( quadrature_order INTEGERLIST
    [ dimension_preference REALLIST ]
    [ nested
    | non_nested ]
    )
  |
  ( sparse_grid_level INTEGERLIST
    [ restricted
    | unrestricted ]
    [ dimension_preference REALLIST ]
    [ nested
    | non_nested ]
    )
  | cubature_integrand INTEGER
  |
  ( collocation_points INTEGER >= 0
    | collocation_ratio REAL
    [ ratio_order REAL ]
    [ ( least_squares
        [ svd
        | equality_constrained ]
        )
    |
    ( orthogonal_matching_pursuit ALIAS omp
      [ noise_tolerance REALLIST ]
      ]
    | basis_pursuit ALIAS bp
    |
    ( basis_pursuit_denoising ALIAS bpdn
      [ noise_tolerance REALLIST ]
      ]
    |
    ( least_angle_regression ALIAS lars
      [ noise_tolerance REALLIST ]
      ]
    |
    ( least_absolute_shrinkage ALIAS lasso
      [ noise_tolerance REALLIST ]
      [ l2_penalty REAL ]
      ]
    [ cross_validation ]
    [ use_derivatives ]
    [ tensor_grid ]
    [ reuse_points ALIAS reuse_samples ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    expansion_order INTEGERLIST
    )
```

```
|
( expansion_samples INTEGER >= 0
  [ reuse_points ALIAS reuse_samples ]
  [ incremental_lhs ]
  [ points_file STRING
    [ annotated
    | freeform ]
    ]
  expansion_order INTEGERLIST
  )
|
( expansion_import_file STRING
  expansion_order INTEGERLIST
  )
[ variance_based_decomp
  [ univariate_effects ]
  [ drop_tolerance REAL ]
  ]
[ diagonal_covariance
| full_covariance ]
[ sample_refinement
  import
  | adapt_import
  | mm_adapt_import
  ]
[ sample_type
  lhs
  | random
  ]
[ fixed_seed ]
[ reliability_levels REALLIST
  [ num_reliability_levels INTEGERLIST ]
  ]
[ response_levels REALLIST
  [ num_response_levels INTEGERLIST ]
  [ compute
    probabilities
    | reliabilities
    | gen_reliabilities
    [ system
      series
      | parallel
      ]
    ]
  ]
[ distribution
  cumulative
  | complementary
  ]
[ probability_levels REALLIST
  [ num_probability_levels INTEGERLIST ]
  ]
[ gen_reliability_levels REALLIST
  [ num_gen_reliability_levels INTEGERLIST ]
  ]
[ rng
  mt19937
  | rnum2
  ]
[ samples INTEGER ]
[ seed INTEGER > 0 ]
)
```

```
|
( stoch_collocation ALIAS nond_stoch_collocation
  [ ( p_refinement
      uniform
      |
      ( dimension_adaptive
        sobol
        | generalized
        )
      )
  |
  ( h_refinement
    uniform
    |
    ( dimension_adaptive
      sobol
      | generalized
      )
    | local_adaptive
    ]
  [ piecewise
  | askey
  | wiener ]
  quadrature_order INTEGERLIST
  |
  ( sparse_grid_level INTEGERLIST
    [ restricted
    | unrestricted ]
    [ nodal
    | hierarchical ]
    )
  [ dimension_preference REALLIST ]
  [ use_derivatives ]
  [ nested
  | non_nested ]
  [ variance_based_decomp
    [ univariate_effects ]
    [ drop_tolerance REAL ]
    ]
  [ diagonal_covariance
  | full_covariance ]
  [ sample_refinement
    import
    | adapt_import
    | mm_adapt_import
    ]
  [ sample_type
    lhs
    | random
    ]
  [ fixed_seed ]
  [ reliability_levels REALLIST
    [ num_reliability_levels INTEGERLIST ]
    ]
  [ response_levels REALLIST
    [ num_response_levels INTEGERLIST ]
    [ compute
      probabilities
      | reliabilities
      | gen_reliabilities
      [ system
        series
```

```
          | parallel
          ]
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( sampling ALIAS nond_sampling
    [ sample_type
      random
      | lhs
      |
      ( incremental_lhs
        | incremental_random
        previous_samples INTEGER
        )
      ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ fixed_seed ]
    [ reliability_levels REALLIST
      [ num_reliability_levels INTEGERLIST ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | reliabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
```

```
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( importance_sampling ALIAS nond_importance_sampling
    [ import
    | adapt_import
    | mm_adapt_import ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( gpais ALIAS gaussian_process_adaptive_importance_sampling
    [ emulator_samples INTEGER ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ distribution
```

```
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( adaptive_sampling ALIAS nond_adaptive_sampling
    [ emulator_samples INTEGER ]
    [ fitness_metric
      predicted_variance
      | distance
      | gradient
      ]
    [ batch_selection
      naive
      | distance_penalty
      | topology
      | constant_liar
      ]
    [ batch_size INTEGER ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ misc_options STRINGLIST ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
```

```
    [ seed INTEGER > 0 ]
    )
  |
  ( efficient_subspace ALIAS nond_efficient_subspace
    [ emulator_samples INTEGER ]
    [ batch_size INTEGER ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( global_evidence ALIAS nond_global_evidence
    [ sbo
      | ego
      [ gaussian_process ALIAS kriging
        surfpack
        | dakota
        ]
      [ use_derivatives ]
      ]
    | ea
    | lhs ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ rng
      mt19937
```

```
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( global_interval_est ALIAS nond_global_interval_est
    [ sbo
      | ego
      [ gaussian_process ALIAS kriging
        surfpack
        | dakota
        ]
      [ use_derivatives ]
      ]
    | ea
    | lhs ]
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ rng
      mt19937
      | rnum2
      ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( bayes_calibration ALIAS nond_bayes_calibration
    ( queso
      [ emulator
        ( gaussian_process ALIAS kriging
          surfpack
          | dakota
          [ emulator_samples INTEGER ]
          [ points_file STRING
            [ annotated
            | freeform ]
            ]
          )
        |
        ( pce
          [ sparse_grid_level INTEGERLIST ]
          )
        |
        ( sc
          [ sparse_grid_level INTEGERLIST ]
          )
        ]
      )
    |
    ( gpmsa
      emulator_samples INTEGER
      )
    [ rejection
      standard
      | delayed
      ]
    [ metropolis
      hastings
      | adaptive
```

```
      ]
    [ use_derivatives ]
    [ rng
      mt19937
      | rnum2
      ]
    [ proposal_covariance_scale REALLIST ]
    [ likelihood_scale REAL ]
    [ calibrate_sigma ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( dace
    grid
    | random
    | oas
    | lhs
    | oa_lhs
    | box_behnken
    | central_composite
    [ main_effects ]
    [ quality_metrics ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ fixed_seed ]
    [ symbols INTEGER ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( fsu_cvt
    [ latinize ]
    [ quality_metrics ]
    [ variance_based_decomp
      [ drop_tolerance REAL ]
      ]
    [ fixed_seed ]
    [ trial_type
      grid
      | halton
      | random
      ]
    [ num_trials INTEGER ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( psuade_moat
    [ partitions INTEGERLIST ]
    [ samples INTEGER ]
    [ seed INTEGER > 0 ]
    )
  |
  ( ncsu_direct
    [ solution_target ALIAS solution_accuracy REAL ]
    [ min_boxsize_limit REAL ]
    [ volume_boxsize_limit REAL ]
    )
  |
  ( local_evidence ALIAS nond_local_evidence
```

```
    [ sqp
    | nip ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    )
  |
  ( local_interval_est ALIAS nond_local_interval_est
    [ sqp
    | nip ]
    )
  |
  ( local_reliability ALIAS nond_local_reliability
    [ mpp_search
      x_taylor_mean
      | u_taylor_mean
      | x_taylor_mpp
      | u_taylor_mpp
      | x_two_point
      | u_two_point
      | no_approx
      [ sqp
      | nip ]
      [ integration
        first_order
        | second_order
        [ sample_refinement
          import
          | adapt_import
          | mm_adapt_import
          [ samples INTEGER ]
          [ seed INTEGER > 0 ]
          ]
        ]
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | reliabilities
        | gen_reliabilities
        [ system
          series
          | parallel
```

```
          ]
        ]
      ]
    [ reliability_levels REALLIST
      [ num_reliability_levels INTEGERLIST ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    )
  |
  ( global_reliability ALIAS nond_global_reliability
    ( x_gaussian_process ALIAS x_kriging
      | u_gaussian_process ALIAS u_kriging
      [ surfpack
      | dakota ]
      )
    [ points_file STRING
      [ annotated
      | freeform ]
      ]
    [ use_derivatives ]
    [ seed INTEGER > 0 ]
    [ rng
      mt19937
      | rnum2
      ]
    [ response_levels REALLIST
      [ num_response_levels INTEGERLIST ]
      [ compute
        probabilities
        | gen_reliabilities
        [ system
          series
          | parallel
          ]
        ]
      ]
    [ distribution
      cumulative
      | complementary
      ]
    [ probability_levels REALLIST
      [ num_probability_levels INTEGERLIST ]
      ]
    [ gen_reliability_levels REALLIST
      [ num_gen_reliability_levels INTEGERLIST ]
      ]
    )
  |
  ( fsu_quasi_mc
    halton
    | hammersley
    [ latinize ]
    [ quality_metrics ]
```

```
          [ variance_based_decomp
            [ drop_tolerance REAL ]
            ]
          [ samples INTEGER ]
          [ fixed_sequence ]
          [ sequence_start INTEGERLIST ]
          [ sequence_leap INTEGERLIST ]
          [ prime_base INTEGERLIST ]
          )
        |
        ( vector_parameter_study
          final_point REALLIST
          | step_vector REALLIST
          num_steps INTEGER
          )
        |
        ( list_parameter_study
          list_of_points REALLIST
          )
        |
        ( centered_parameter_study
          step_vector REALLIST
          steps_per_variable ALIAS deltas_per_variable INTEGERLIST
          )
        |
        ( multidim_parameter_study
          partitions INTEGERLIST
          )
        |
        ( richardson_extrap
          estimate_order
          | converge_order
          | converge_qoi
          [ refinement_rate REAL ]
          )

KEYWORD model
        [ id_model STRING ]
        [ variables_pointer STRING ]
        [ responses_pointer STRING ]
        ( single
          [ interface_pointer STRING ]
          )
        |
        ( surrogate
          [ id_surrogates INTEGERLIST ]
          ( global
            ( gaussian_process ALIAS kriging
              ( dakota
                [ point_selection ]
                [ trend
                  constant
                  | linear
                  | reduced_quadratic
                  ]
                )
              |
              ( surfpack
                [ trend
                  constant
                  | linear
                  | reduced_quadratic
```

```
      | quadratic
      ]
    [ optimization_method STRING ]
    [ max_trials INTEGER > 0 ]
    [ correlation_lengths REALLIST ]
    )
  )
|
( mars
  [ max_bases INTEGER ]
  [ interpolation
    linear
    | cubic
    ]
  )
|
( moving_least_squares
  [ poly_order INTEGER ]
  [ weight_function INTEGER ]
  )
|
( neural_network
  [ nodes INTEGER ]
  [ range REAL ]
  [ random_weight INTEGER ]
  )
|
( radial_basis
  [ bases INTEGER ]
  [ max_pts INTEGER ]
  [ min_partition INTEGER ]
  [ max_subsets INTEGER ]
  )
|
( polynomial
  linear
  | quadratic
  | cubic
  )
[ total_points INTEGER
| minimum_points
| recommended_points ]
[ dace_method_pointer STRING ]
[ reuse_points ALIAS reuse_samples
  all
  | region
  | none
  ]
[ points_file ALIAS samples_file STRING
  [ annotated
  | freeform ]
  ]
[ use_derivatives ]
[ correction
  zeroth_order
  | first_order
  | second_order
  additive
  | multiplicative
  | combined
  ]
[ diagnostics STRINGLIST ]
```

```
            )
          |
          ( multipoint
            tana
            actual_model_pointer STRING
            )
          |
          ( local
            taylor_series
            actual_model_pointer STRING
            )
          |
          ( hierarchical
            low_fidelity_model_pointer STRING
            high_fidelity_model_pointer STRING
            ( correction
              zeroth_order
              | first_order
              | second_order
              additive
              | multiplicative
              | combined
              )
            )
          )
        |
        ( nested
          [ optional_interface_pointer STRING
            [ optional_interface_responses_pointer STRING ]
            ]
          ( sub_method_pointer STRING
            [ primary_variable_mapping STRINGLIST ]
            [ secondary_variable_mapping STRINGLIST ]
            [ primary_response_mapping REALLIST ]
            [ secondary_response_mapping REALLIST ]
            )
          )

KEYWORD12 variables
        [ id_variables STRING ]
        [ active
          all
          | design
          | uncertain
          | aleatory
          | epistemic
          | state
          ]
        [ mixed
        | relaxed ]
        [ continuous_design INTEGER > 0
          [ initial_point ALIAS cdv_initial_point REALLIST ]
          [ lower_bounds ALIAS cdv_lower_bounds REALLIST ]
          [ upper_bounds ALIAS cdv_upper_bounds REALLIST ]
          [ scale_types ALIAS cdv_scale_types STRINGLIST ]
          [ scales ALIAS cdv_scales REALLIST ]
          [ descriptors ALIAS cdv_descriptors STRINGLIST ]
          ]
        [ discrete_design_range INTEGER > 0
          [ initial_point ALIAS ddv_initial_point INTEGERLIST ]
          [ lower_bounds ALIAS ddv_lower_bounds INTEGERLIST ]
          [ upper_bounds ALIAS ddv_upper_bounds INTEGERLIST ]
```

```
    [ descriptors ALIAS ddv_descriptors STRINGLIST ]
    ]
  [ discrete_design_set_integer INTEGER > 0
    [ initial_point INTEGERLIST ]
    [ num_set_values INTEGERLIST ]
    set_values INTEGERLIST
    [ descriptors STRINGLIST ]
    ]
  [ discrete_design_set_real INTEGER > 0
    [ initial_point REALLIST ]
    [ num_set_values INTEGERLIST ]
    set_values REALLIST
    [ descriptors STRINGLIST ]
    ]
  [ normal_uncertain INTEGER > 0
    means ALIAS nuv_means REALLIST
    std_deviations ALIAS nuv_std_deviations REALLIST
    [ lower_bounds ALIAS nuv_lower_bounds REALLIST ]
    [ upper_bounds ALIAS nuv_upper_bounds REALLIST ]
    [ descriptors ALIAS nuv_descriptors STRINGLIST ]
    ]
  [ lognormal_uncertain INTEGER > 0
    ( lambdas ALIAS lnuv_lambdas REALLIST
      zetas ALIAS lnuv_zetas REALLIST
      )
    |
    ( means ALIAS lnuv_means REALLIST
      std_deviations ALIAS lnuv_std_deviations REALLIST
      | error_factors ALIAS lnuv_error_factors REALLIST
      )
    [ lower_bounds ALIAS lnuv_lower_bounds REALLIST ]
    [ upper_bounds ALIAS lnuv_upper_bounds REALLIST ]
    [ descriptors ALIAS lnuv_descriptors STRINGLIST ]
    ]
  [ uniform_uncertain INTEGER > 0
    lower_bounds ALIAS uuv_lower_bounds REALLIST
    upper_bounds ALIAS uuv_upper_bounds REALLIST
    [ descriptors ALIAS uuv_descriptors STRINGLIST ]
    ]
  [ loguniform_uncertain INTEGER > 0
    lower_bounds ALIAS luuv_lower_bounds REALLIST
    upper_bounds ALIAS luuv_upper_bounds REALLIST
    [ descriptors ALIAS luuv_descriptors STRINGLIST ]
    ]
  [ triangular_uncertain INTEGER > 0
    modes ALIAS tuv_modes REALLIST
    lower_bounds ALIAS tuv_lower_bounds REALLIST
    upper_bounds ALIAS tuv_upper_bounds REALLIST
    [ descriptors ALIAS tuv_descriptors STRINGLIST ]
    ]
  [ exponential_uncertain INTEGER > 0
    betas ALIAS euv_betas REALLIST
    [ descriptors ALIAS euv_descriptors STRINGLIST ]
    ]
  [ beta_uncertain INTEGER > 0
    alphas ALIAS buv_alphas REALLIST
    betas ALIAS buv_betas REALLIST
    lower_bounds ALIAS buv_lower_bounds REALLIST
    upper_bounds ALIAS buv_upper_bounds REALLIST
    [ descriptors ALIAS buv_descriptors STRINGLIST ]
    ]
  [ gamma_uncertain INTEGER > 0
```

```
      alphas ALIAS gauv_alphas REALLIST
      betas ALIAS gauv_betas REALLIST
      [ descriptors ALIAS gauv_descriptors STRINGLIST ]
      ]
    [ gumbel_uncertain INTEGER > 0
      alphas ALIAS guuv_alphas REALLIST
      betas ALIAS guuv_betas REALLIST
      [ descriptors ALIAS guuv_descriptors STRINGLIST ]
      ]
    [ frechet_uncertain INTEGER > 0
      alphas ALIAS fuv_alphas REALLIST
      betas ALIAS fuv_betas REALLIST
      [ descriptors ALIAS fuv_descriptors STRINGLIST ]
      ]
    [ weibull_uncertain INTEGER > 0
      alphas ALIAS wuv_alphas REALLIST
      betas ALIAS wuv_betas REALLIST
      [ descriptors ALIAS wuv_descriptors STRINGLIST ]
      ]
    [ histogram_bin_uncertain INTEGER > 0
      [ num_pairs ALIAS huv_num_bin_pairs INTEGERLIST ]
      abscissas ALIAS huv_bin_abscissas REALLIST
      ordinates ALIAS huv_bin_ordinates REALLIST
      | counts ALIAS huv_bin_counts REALLIST
      [ descriptors ALIAS huv_bin_descriptors STRINGLIST ]
      ]
    [ poisson_uncertain INTEGER > 0
      lambdas REALLIST
      [ descriptors STRINGLIST ]
      ]
    [ binomial_uncertain INTEGER > 0
      probability_per_trial ALIAS prob_per_trial REALLIST
      num_trials INTEGERLIST
      [ descriptors STRINGLIST ]
      ]
    [ negative_binomial_uncertain INTEGER > 0
      probability_per_trial ALIAS prob_per_trial REALLIST
      num_trials INTEGERLIST
      [ descriptors STRINGLIST ]
      ]
    [ geometric_uncertain INTEGER > 0
      probability_per_trial ALIAS prob_per_trial REALLIST
      [ descriptors STRINGLIST ]
      ]
    [ hypergeometric_uncertain INTEGER > 0
      total_population INTEGERLIST
      selected_population INTEGERLIST
      num_drawn INTEGERLIST
      [ descriptors STRINGLIST ]
      ]
    [ histogram_point_uncertain INTEGER > 0
      [ num_pairs ALIAS huv_num_point_pairs INTEGERLIST ]
      abscissas ALIAS huv_point_abscissas REALLIST
      counts ALIAS huv_point_counts REALLIST
      [ descriptors ALIAS huv_point_descriptors STRINGLIST ]
      ]
    [ uncertain_correlation_matrix REALLIST ]
    [ continuous_interval_uncertain ALIAS interval_uncertain INTEGER > 0
      [ num_intervals ALIAS iuv_num_intervals INTEGERLIST ]
      [ interval_probabilities ALIAS interval_probs ALIAS iuv_interval_probs
REALLIST ]
      lower_bounds REALLIST
```

```
          upper_bounds REALLIST
          [ descriptors ALIAS iuv_descriptors STRINGLIST ]
          ]
        [ discrete_interval_uncertain ALIAS discrete_uncertain_range INTEGER > 0
          [ num_intervals INTEGERLIST ]
          [ interval_probabilities ALIAS interval_probs ALIAS range_probabilities
       ALIAS range_probs REALLIST ]
          lower_bounds INTEGERLIST
          upper_bounds INTEGERLIST
          [ descriptors STRINGLIST ]
          ]
        [ discrete_uncertain_set_integer INTEGER > 0
          [ num_set_values INTEGERLIST ]
          set_values INTEGERLIST
          [ set_probabilities ALIAS set_probs REALLIST ]
          [ descriptors STRINGLIST ]
          ]
        [ discrete_uncertain_set_real INTEGER > 0
          [ num_set_values INTEGERLIST ]
          set_values REALLIST
          [ set_probabilities ALIAS set_probs REALLIST ]
          [ descriptors STRINGLIST ]
          ]
        [ continuous_state INTEGER > 0
          [ initial_state ALIAS csv_initial_state REALLIST ]
          [ lower_bounds ALIAS csv_lower_bounds REALLIST ]
          [ upper_bounds ALIAS csv_upper_bounds REALLIST ]
          [ descriptors ALIAS csv_descriptors STRINGLIST ]
          ]
        [ discrete_state_range INTEGER > 0
          [ initial_state ALIAS dsv_initial_state INTEGERLIST ]
          [ lower_bounds ALIAS dsv_lower_bounds INTEGERLIST ]
          [ upper_bounds ALIAS dsv_upper_bounds INTEGERLIST ]
          [ descriptors ALIAS dsv_descriptors STRINGLIST ]
          ]
        [ discrete_state_set_integer INTEGER > 0
          [ initial_state INTEGERLIST ]
          [ num_set_values INTEGERLIST ]
          set_values INTEGERLIST
          [ descriptors STRINGLIST ]
          ]
        [ discrete_state_set_real INTEGER > 0
          [ initial_state REALLIST ]
          [ num_set_values INTEGERLIST ]
          set_values REALLIST
          [ descriptors STRINGLIST ]
          ]

KEYWORD12 interface
        [ id_interface STRING ]
        [ algebraic_mappings STRING ]
        [ analysis_drivers STRINGLIST
          [ analysis_components STRINGLIST ]
          [ input_filter STRING ]
          [ output_filter STRING ]
          ( system
            | fork
            [ parameters_file STRING ]
            [ results_file STRING ]
            [ allow_existing_results ]
            [ verbatim ]
            [ aprepro ]
```

```
            [ file_tag ]
            [ file_save ]
            [ work_directory
              [ named STRING ]
              [ directory_tag ALIAS dir_tag ]
              [ directory_save ALIAS dir_save ]
              [ template_directory STRING
                | template_files STRINGLIST
                [ copy ]
                [ replace ]
                ]
              ]
            )
          |
          ( direct
            [ processors_per_analysis INTEGER > 0 ]
            )
          | matlab
          |
          ( python
            [ numpy ]
            )
          | scilab
          | grid
          [ failure_capture
            abort
            | retry INTEGER
            | recover REALLIST
            | continuation
            ]
          [ deactivate
            [ active_set_vector ]
            [ evaluation_cache ]
            [ restart_file ]
            ]
          ]
        [ asynchronous
          [ evaluation_concurrency INTEGER > 0 ]
          [ local_evaluation_self_scheduling
          | local_evaluation_static_scheduling ]
          [ analysis_concurrency INTEGER > 0 ]
          ]
        [ evaluation_servers INTEGER > 0 ]
        [ evaluation_self_scheduling
        | evaluation_static_scheduling ]
        [ analysis_servers INTEGER > 0 ]
        [ analysis_self_scheduling
        | analysis_static_scheduling ]

KEYWORD12 responses
        [ id_responses STRING ]
        [ descriptors ALIAS response_descriptors STRINGLIST ]
        ( objective_functions ALIAS num_objective_functions INTEGER >= 0
          [ sense STRINGLIST ]
          [ primary_scale_types ALIAS objective_function_scale_types STRINGLIST ]

          [ primary_scales ALIAS objective_function_scales REALLIST ]
          [ weights ALIAS multi_objective_weights REALLIST ]
          [ nonlinear_inequality_constraints ALIAS num_nonlinear_inequality_const
      raints INTEGER >= 0
            [ lower_bounds ALIAS nonlinear_inequality_lower_bounds REALLIST ]
            [ upper_bounds ALIAS nonlinear_inequality_upper_bounds REALLIST ]
```

```
      [ scale_types ALIAS nonlinear_inequality_scale_types STRINGLIST ]
      [ scales ALIAS nonlinear_inequality_scales REALLIST ]
      ]
    [ nonlinear_equality_constraints ALIAS num_nonlinear_equality_constrain
ts INTEGER >= 0
      [ targets ALIAS nonlinear_equality_targets REALLIST ]
      [ scale_types ALIAS nonlinear_equality_scale_types STRINGLIST ]
      [ scales ALIAS nonlinear_equality_scales REALLIST ]
      ]
    )
  |
  ( calibration_terms ALIAS least_squares_terms ALIAS num_least_squares_ter
ms INTEGER >= 0
    [ primary_scale_types ALIAS calibration_term_scale_types ALIAS least_sq
uares_term_scale_types STRINGLIST ]
    [ primary_scales ALIAS calibration_term_scales ALIAS least_squares_term
_scales REALLIST ]
    [ weights ALIAS calibration_weights ALIAS least_squares_weights REALLIS
T ]
    [ calibration_data_file ALIAS least_squares_data_file STRING
      [ num_experiments INTEGER >= 0 ]
      [ annotated
      | freeform ]
      [ num_config_variables INTEGER >= 0 ]
      [ num_std_deviations INTEGER >= 0 ]
      ]
    [ nonlinear_inequality_constraints ALIAS num_nonlinear_inequality_const
raints INTEGER >= 0
      [ lower_bounds ALIAS nonlinear_inequality_lower_bounds REALLIST ]
      [ upper_bounds ALIAS nonlinear_inequality_upper_bounds REALLIST ]
      [ scale_types ALIAS nonlinear_inequality_scale_types STRINGLIST ]
      [ scales ALIAS nonlinear_inequality_scales REALLIST ]
      ]
    [ nonlinear_equality_constraints ALIAS num_nonlinear_equality_constrain
ts INTEGER >= 0
      [ targets ALIAS nonlinear_equality_targets REALLIST ]
      [ scale_types ALIAS nonlinear_equality_scale_types STRINGLIST ]
      [ scales ALIAS nonlinear_equality_scales REALLIST ]
      ]
    )
  | response_functions ALIAS num_response_functions INTEGER >= 0
  no_gradients
  | analytic_gradients
  |
  ( mixed_gradients
    id_numerical_gradients INTEGERLIST
    id_analytic_gradients INTEGERLIST
    [ method_source ]
    [ ( dakota
        [ ignore_bounds ]
        [ relative
        | absolute
        | bounds ]
        )
    | vendor ]
    [ interval_type ]
    [ forward
    | central ]
    [ fd_step_size ALIAS fd_gradient_step_size REALLIST := 0.001 ]
    )
  |
  ( numerical_gradients
```

```
    [ method_source ]
    [ ( dakota
        [ ignore_bounds ]
        [ relative
        | absolute
        | bounds ]
        )
    | vendor ]
    [ interval_type ]
    [ forward
    | central ]
    [ fd_step_size ALIAS fd_gradient_step_size REALLIST := 0.001 ]
    )
  no_hessians
  |
  ( numerical_hessians
    [ fd_step_size ALIAS fd_hessian_step_size REALLIST ]
    [ relative
    | absolute
    | bounds ]
    [ forward
    | central ]
    )
  |
  ( quasi_hessians
    ( bfgs
      [ damped ]
      )
    | sr1
    )
  | analytic_hessians
  |
  ( mixed_hessians
    [ id_numerical_hessians INTEGERLIST
      [ fd_step_size ALIAS fd_hessian_step_size REALLIST ]
      ]
    [ relative
    | absolute
    | bounds ]
    [ forward
    | central ]
    [ id_quasi_hessians INTEGERLIST
      ( bfgs
        [ damped ]
        )
      | sr1
      ]
    [ id_analytic_hessians INTEGERLIST ]
    )
```