SANDIA REPORT

SAND2006-4055
Unlimited Release
Printed October 2006

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

Version 4.0 Reference Manual

Michael S. Eldred, Anthony A. Giunta, Shannon L. Brown, Brian M. Adams, Daniel M. Dunlavy, John P. Eddy, David M. Gay, Josh D. Griffin, William E. Hart, Patty D. Hough, Tammy G. Kolda, Monica L. Martinez-Canales, Laura P. Swiler, Jean-Paul Watson, Pamela J. Williams

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

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

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

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

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

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

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

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

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2006-4055
Unlimited Release
Printed October 2006

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

Version 4.0 Reference Manual

Michael S. Eldred, Shannon L. Brown, Brian M. Adams, Daniel M. Dunlavy,
David M. Gay, Laura P. Swiler
Optimization and Uncertainty Estimation Department

Anthony A. Giunta
Validation and Uncertainty Quantification Processes Department

William E. Hart, Jean-Paul Watson
Discrete Algorithms and Math Department

John P. Eddy
System Sustainment and Readiness Technologies Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Josh D. Griffin, Patty D. Hough, Tammy G. Kolda, Monica L. Martinez-Canales,
Pamela J. Williams
Computational Sciences and Mathematics Research 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 finite element 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

Chapter 1

DAKOTA Reference Manual

Author:
Michael S. Eldred, Anthony A. Giunta, Shannon L. Brown, Brian M. Adams, Daniel M. Dunlavy, John P. Eddy, David M. Gay, Josh D. Griffin, William E. Hart, Patty D. Hough, Tamara G. Kolda, Monica L. Martinez-Canales, Laura P. Swiler, Jean-Paul Watson, Pamela J. Williams

1.1 Introduction

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

The Reference Manual focuses on documentation of the various input commands for the DAKOTA system. It follows closely the structure of dakota.input.spec, the master input specification. For information on software structure, refer to the `Developers Manual`, and for a tour of DAKOTA features and capabilities, refer to the Users Manual [Eldred et al., 2006].

1.2 Input Specification Reference

In the DAKOTA system, 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://endo.sandia.gov/DAKOTA` with software specifics and documentation pointers provided at `http://endo.sandia.gov/DAKOTA/software.html`, and a list of publications provided at `http://endo.sandia.gov/DAKOTA/references.html`

Chapter 2

DAKOTA File Documentation

2.1 dakota.input.spec File Reference

File containing the input specification for DAKOTA.

2.1.1 Detailed Description

File containing the input specification for DAKOTA.

This file is used in the generation of parser system files which are compiled into the DAKOTA executable. Therefore, this file is 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 are enclosed in `{}`, optional individual specifications are enclosed in [], required group specifications are enclosed in (), optional group specifications are enclosed in [], and either-or relationships are denoted by the | symbol. These symbols only appear in dakota.input.spec; they must not appear in actual user input files.

- Keyword specifications (i.e., `strategy`, `method`, `model`, `variables`, `interface`, and `responses`) are delimited by newline characters, both in the input specification and in user input files. Therefore, to continue a keyword specification onto multiple lines, the back-slash character (\) is needed at the end of a line in order to escape the newline. Continuation onto multiple lines is not required; however, it is commonly used to enhance readability.

- Each of the six keywords in the input specification begins with a

```
<KEYWORD = name>, <FUNCTION = handler_name>
```

header which names the keyword and provides the binding to the keyword handler within DAKOTA's problem description database. In a user input file, only the name of the keyword appears (e.g., `variables`).

- Some of the keyword components within the input specification indicate that the user must supply <INTEGER>, <REAL>, <STRING>, <LISTof><INTEGER>, <LISTof><REAL>, or <LISTof><STRING> data as part of the specification. In a user input file, the "=" is optional, the <LISTof> data can be separated by commas or whitespace, and the <STRING> data are enclosed in single quotes (e.g., 'text_book').

- In user input files, input is 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.spec file used in DAKOTA V3.3 is:

```
# DO NOT CHANGE THIS FILE UNLESS YOU UNDERSTAND THE COMPLETE UPDATE PROCESS
#
# Any changes made to the input specification require the manual merging
# of code fragments generated by IDR into the DAKOTA code.  If this manual
# merging is not performed, then libidr.a and the Dakota src files
# (ProblemDescDB.C, keywordtable.C) will be out of synch which will cause
# errors that are difficult to track.  Please be sure to consult the
# documentation in Dakota/docs/SpecChange.dox before you modify the input
# specification or otherwise change the IDR subsystem.
#
<KEYWORD = strategy>, <FUNCTION = strategy_kwhandler>                    \
        [graphics]                                                       \
        [ {tabular_graphics_data} [tabular_graphics_file = <STRING>] ]   \
        [iterator_servers = <INTEGER>]                                   \
        [iterator_self_scheduling] [iterator_static_scheduling]          \
        ( {multi_level}                                                  \
            ( {uncoupled}                                                \
                [ {adaptive} {progress_threshold = <REAL>} ]             \
                {method_list = <LISTof><STRING>} )                       \
            |                                                            \
            ( {coupled}                                                  \
                {global_method_pointer = <STRING>}                       \
                {local_method_pointer = <STRING>}                        \
                [local_search_probability = <REAL>] ) )                  \
        |                                                                \
        ( {surrogate_based_opt}                                          \
                {opt_method_pointer = <STRING>}                          \
                [max_iterations = <INTEGER>]                             \
                [convergence_tolerance = <REAL>]                         \
                [soft_convergence_limit = <INTEGER>]                     \
                [truth_surrogate_bypass]                                 \
                [ {trust_region} [initial_size = <REAL>]                 \
                  [ {constraint_relax} {homotopy} ]                      \
                  [minimum_size = <REAL>]                                \
```

```
                  [contract_threshold = <REAL>]                          \
                  [expand_threshold = <REAL>]                            \
                  [contraction_factor = <REAL>]                          \
                  [expansion_factor = <REAL>] ] )                        \
        |                                                                \
#       ( {branch_and_bound}                                             \
#               {opt_method_pointer = <STRING>}                          \
#               [num_samples_at_root = <INTEGER>]                        \
#               [num_samples_at_node = <INTEGER>] )                      \
#       |                                                                \
        ( {multi_start}                                                  \
                {method_pointer = <STRING>}                              \
                [ {random_starts = <INTEGER>} [seed = <INTEGER>] ]       \
                [starting_points = <LISTof><REAL>] )                     \
        |                                                                \
        ( {pareto_set}                                                   \
                {opt_method_pointer = <STRING>}                          \
                [ {random_weight_sets = <INTEGER>} [seed = <INTEGER>] ] \
                [multi_objective_weight_sets = <LISTof><REAL>] )         \
        |                                                                \
        ( {single_method}                                                \
                [method_pointer = <STRING>] )

<KEYWORD = method>, <FUNCTION = method_kwhandler>                        \
        [id_method = <STRING>]                                           \
        [model_pointer = <STRING>]                                       \
        [ {output} {debug} | {verbose} | {quiet} | {silent} ]            \
        [max_iterations = <INTEGER>]                                     \
        [max_function_evaluations = <INTEGER>]                           \
        [speculative]                                                    \
        [convergence_tolerance = <REAL>]                                 \
        [constraint_tolerance = <REAL>]                                  \
        [scaling]                                                        \
        [linear_inequality_constraint_matrix = <LISTof><REAL>]           \
        [linear_inequality_lower_bounds = <LISTof><REAL>]                \
        [linear_inequality_upper_bounds = <LISTof><REAL>]                \
        [linear_inequality_scales = <LISTof><REAL>]                      \
        [linear_equality_constraint_matrix = <LISTof><REAL>]             \
        [linear_equality_targets = <LISTof><REAL>]                       \
        [linear_equality_scales = <LISTof><REAL>]                        \
        ( {dot_frcg}                                                     \
                [ {optimization_type} {minimize} | {maximize} ] )        \
        |                                                                \
        ( {dot_mmfd}                                                     \
                [ {optimization_type} {minimize} | {maximize} ] )        \
        |                                                                \
        ( {dot_bfgs}                                                     \
                [ {optimization_type} {minimize} | {maximize} ] )        \
        |                                                                \
        ( {dot_slp}                                                      \
                [ {optimization_type} {minimize} | {maximize} ] )        \
        |                                                                \
        ( {dot_sqp}                                                      \
                [ {optimization_type} {minimize} | {maximize} ] )        \
        |                                                                \
        ( {conmin_frcg} )                                                \
        |                                                                \
        ( {conmin_mfd}  )                                                \
        |                                                                \
        ( {npsol_sqp}                                                    \
                [verify_level = <INTEGER>]                               \
                [function_precision = <REAL>]                            \
```

```
        [linesearch_tolerance = <REAL>] )                       \
|                                                               \
( {nlssol_sqp}                                                  \
        [verify_level = <INTEGER>]                              \
        [function_precision = <REAL>]                           \
        [linesearch_tolerance = <REAL>] )                       \
|                                                               \
( {nlpql_sqp} )                                                 \
|                                                               \
( {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] )     \
|                                                               \
( {reduced_sqp} )                                               \
|                                                               \
( {optpp_cg}                                                    \
        [max_step = <REAL>] [gradient_tolerance = <REAL>] )     \
|                                                               \
( {optpp_q_newton}                                              \
        [ {search_method} {value_based_line_search} |           \
          {gradient_based_line_search} | {trust_region} |       \
          {tr_pds} ]                                            \
        [max_step = <REAL>] [gradient_tolerance = <REAL>]       \
        [merit_function = <STRING>] [central_path = <STRING>]   \
        [steplength_to_boundary = <REAL>]                       \
        [centering_parameter = <REAL>] )                        \
|                                                               \
( {optpp_fd_newton}                                             \
        [ {search_method} {value_based_line_search} |           \
          {gradient_based_line_search} | {trust_region} |       \
          {tr_pds} ]                                            \
        [max_step = <REAL>] [gradient_tolerance = <REAL>]       \
        [merit_function = <STRING>] [central_path = <STRING>]   \
        [steplength_to_boundary = <REAL>]                       \
        [centering_parameter = <REAL>] )                        \
|                                                               \
( {optpp_g_newton}                                              \
        [ {search_method} {value_based_line_search} |           \
          {gradient_based_line_search} | {trust_region} |       \
          {tr_pds} ]                                            \
        [max_step = <REAL>] [gradient_tolerance = <REAL>]       \
        [merit_function = <STRING>] [central_path = <STRING>]   \
        [steplength_to_boundary = <REAL>]                       \
        [centering_parameter = <REAL>] )                        \
|                                                               \
( {optpp_newton}                                                \
        [ {search_method} {value_based_line_search} |           \
          {gradient_based_line_search} | {trust_region} |       \
          {tr_pds} ]                                            \
        [max_step = <REAL>] [gradient_tolerance = <REAL>]       \
        [merit_function = <STRING>] [central_path = <STRING>]   \
        [steplength_to_boundary = <REAL>]                       \
        [centering_parameter = <REAL>] )                        \
|                                                               \
( {optpp_pds}                                                   \
        [search_scheme_size = <INTEGER>] )                      \
|                                                               \
( {coliny_apps}                                                 \
```

```
        [solution_accuracy = <REAL>] [seed = <INTEGER>]         \
        {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
        [contraction_factor = <REAL>]                           \
        [ {synchronization} {blocking} | {nonblocking} ]        \
        [constraint_penalty = <REAL>]                           \
        [show_misc_options] [misc_options = <LISTof><STRING>] ) \
|                                                               \
( {coliny_cobyla}                                               \
        [solution_accuracy = <REAL>] [seed = <INTEGER>]         \
        {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
        [show_misc_options] [misc_options = <LISTof><STRING>] ) \
|                                                               \
( {coliny_direct}                                               \
        [solution_accuracy = <REAL>] [seed = <INTEGER>]         \
        [ {division} {major_dimension} | {all_dimensions} ]     \
        [global_balance_parameter = <REAL>]                     \
        [local_balance_parameter = <REAL>]                      \
        [max_boxsize_limit = <REAL>]                            \
        [min_boxsize_limit = <REAL>]                            \
        [constraint_penalty = <REAL>]                           \
        [show_misc_options] [misc_options = <LISTof><STRING>] ) \
|                                                               \
( {coliny_pattern_search}                                       \
        [solution_accuracy = <REAL>] [seed = <INTEGER>]         \
        {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
        [contraction_factor = <REAL>]                           \
        [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} ]        \
        [constraint_penalty = <REAL>] [constant_penalty]        \
        [show_misc_options] [misc_options = <LISTof><STRING>] ) \
|                                                               \
( {coliny_solis_wets}                                           \
        [solution_accuracy = <REAL>] [seed = <INTEGER>]         \
        {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
        [no_expansion] [expand_after_success = <INTEGER>]       \
        [contract_after_failure = <INTEGER>]                    \
        [contraction_factor = <REAL>]                           \
        [constraint_penalty = <REAL>] [constant_penalty]        \
        [show_misc_options] [misc_options = <LISTof><STRING>] ) \
|                                                               \
( {coliny_ea}                                                   \
        [solution_accuracy = <REAL>] [seed = <INTEGER>]         \
        [population_size = <INTEGER>]                           \
        [ {initialization_type} {simple_random} |               \
          {unique_random} | {flat_file = <STRING>} ]            \
        [ {fitness_type} {linear_rank} | {merit_function} ]     \
        [ {replacement_type} {random = <INTEGER>} |             \
          {chc = <INTEGER>} | {elitist = <INTEGER>}             \
          [new_solutions_generated = <INTEGER>] ]               \
        [crossover_rate = <REAL>]                               \
        [ {crossover_type} {two_point} | {blend} | {uniform} ]  \
        [mutation_rate = <REAL>]                                \
        [ {mutation_type} {replace_uniform} |                   \
          ( {offset_normal}  [mutation_scale = <REAL>]          \
                             [mutation_range = <INTEGER>] ) |   \
          ( {offset_cauchy}  [mutation_scale = <REAL>]          \
                             [mutation_range = <INTEGER>] ) |   \
```

```
                  ( {offset_uniform} [mutation_scale = <REAL>]          \
                                     [mutation_range = <INTEGER>] )     \
                  [non_adaptive] ]                                      \
                [constraint_penalty = <REAL>]                           \
                [show_misc_options] [misc_options = <LISTof><STRING>] ) \
#       |                                                               \
#       ( {coliny_misc_solver}                                          \
#               [show_misc_options] [misc_options = <LISTof><STRING>] ) \
        |                                                               \
        ( {moga}                                                        \
                [seed = <INTEGER>] [population_size = <INTEGER>]        \
                [log_file = <STRING>] [print_each_pop]                  \
                [ {initialization_type} {simple_random} |               \
                  {unique_random} | {flat_file = <STRING>} ]            \
                [ {crossover_type} {multi_point_binary = <INTEGER>} |   \
                  {multi_point_parameterized_binary = <INTEGER>}    |   \
                  {multi_point_real = <INTEGER>}                    |   \
                  ( {shuffle_random} [num_parents = <INTEGER>]          \
                    [num_offspring = <INTEGER>] )                       \
                  [crossover_rate = <REAL>] ]                           \
                [ {mutation_type} {bit_random} | {replace_uniform} |    \
                   ( {offset_normal}  [mutation_scale = <REAL>] )  |    \
                   ( {offset_cauchy}  [mutation_scale = <REAL>] )  |    \
                   ( {offset_uniform} [mutation_scale = <REAL>] )       \
                   [mutation_rate = <REAL>] ]                           \
                [ {fitness_type} {layer_rank} | {domination_count} ]    \
                [ {replacement_type} {roulette_wheel} |                 \
                  {unique_roulette_wheel}  |                            \
                  ({below_limit = <REAL>}                               \
                  [shrinkage_percentage = <REAL>])]                     \
                [ {niching_type} {radial = <LISTof><REAL>}]             \
                [ {convergence_type}                                    \
                  ( {metric_tracker} [percent_change = <REAL>]          \
                    [num_generations = <INTEGER>] )])                   \
        |                                                               \
        ( {soga}                                                        \
                [seed = <INTEGER>] [population_size = <INTEGER>]        \
                [log_file = <STRING>] [print_each_pop]                  \
                [ {initialization_type} {simple_random} |               \
                  {unique_random} | {flat_file = <STRING>} ]            \
                [ {crossover_type} {multi_point_binary = <INTEGER>} |   \
                  {multi_point_parameterized_binary = <INTEGER>}    |   \
                  {multi_point_real = <INTEGER>}                    |   \
                  ( {shuffle_random} [num_parents = <INTEGER>]          \
                    [num_offspring = <INTEGER>] )                       \
                  [crossover_rate = <REAL>] ]                           \
                [ {mutation_type} {bit_random} | {replace_uniform} |    \
                   ( {offset_normal}  [mutation_scale = <REAL>] )  |    \
                   ( {offset_cauchy}  [mutation_scale = <REAL>] )  |    \
                   ( {offset_uniform} [mutation_scale = <REAL>] )       \
                   [mutation_rate = <REAL>] ]                           \
                [ {fitness_type}({merit_function}                       \
                       [constraint_penalty = <REAL>])]                  \
                [ {replacement_type} {favor_feasible} |                 \
                    {roulette_wheel} | {unique_roulette_wheel}]         \
                [ {convergence_type}                                    \
                  ( {best_fitness_tracker} [percent_change = <REAL>]    \
                    [num_generations = <INTEGER>] ) |                   \
                  ( {average_fitness_tracker} [percent_change = <REAL>] \
                    [num_generations = <INTEGER>] ) ] )                 \
        |                                                               \
        ( {nond_polynomial_chaos}                                       \
```

```
        {expansion_terms = <INTEGER>} |                          \
        {expansion_order = <INTEGER>}                            \
        [seed = <INTEGER>] [fixed_seed] [samples = <INTEGER>]    \
        [ {sample_type} {random} | {lhs} ]                       \
        [ {distribution} {cumulative} | {complementary} ]        \
        [ {response_levels = <LISTof><REAL>}                     \
          [num_response_levels = <LISTof><INTEGER>]              \
          [ {compute} {probabilities} | {reliabilities} ] ]      \
        [ {probability_levels = <LISTof><REAL>}                  \
          [num_probability_levels = <LISTof><INTEGER>] ]         \
        [ {reliability_levels = <LISTof><REAL>}                  \
          [num_reliability_levels = <LISTof><INTEGER>] ] )       \
|                                                                \
( {nond_sampling}                                                \
        [seed = <INTEGER>] [fixed_seed] [samples = <INTEGER>]    \
        [ {sample_type} {random} | {lhs} ] [all_variables]       \
        [ {distribution} {cumulative} | {complementary} ]        \
        [ {response_levels = <LISTof><REAL>}                     \
          [num_response_levels = <LISTof><INTEGER>]              \
          [ {compute} {probabilities} | {reliabilities} ] ]      \
        [ {probability_levels = <LISTof><REAL>}                  \
          [num_probability_levels = <LISTof><INTEGER>] ]         \
        [ {reliability_levels = <LISTof><REAL>}                  \
          [num_reliability_levels = <LISTof><INTEGER>] ]         \
        [variance_based_decomp] )                                \
|                                                                \
( {nond_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} ]         \
        [ {distribution} {cumulative} | {complementary} ]        \
        [ {response_levels = <LISTof><REAL>}                     \
          [num_response_levels = <LISTof><INTEGER>]              \
          [ {compute} {probabilities} | {reliabilities} ] ]      \
        [ {probability_levels = <LISTof><REAL>}                  \
          [num_probability_levels = <LISTof><INTEGER>] ]         \
        [ {reliability_levels = <LISTof><REAL>}                  \
          [num_reliability_levels = <LISTof><INTEGER>] ] )       \
|                                                                \
( {nond_evidence}                                                \
        [seed = <INTEGER>] [samples = <INTEGER>] )               \
|                                                                \
( {dace}                                                         \
        {grid} | {random} | {oas} | {lhs} | {oa_lhs} |           \
        {box_behnken} | {central_composite}                      \
        [main_effects] [quality_metrics]                         \
        [variance_based_decomp]                                  \
        [seed = <INTEGER>] [fixed_seed]                          \
        [samples = <INTEGER>] [symbols = <INTEGER>] )            \
                                                                 \
|                                                                \
( {fsu_quasi_mc}                                                 \
        {halton} | {hammersley}                                  \
        [latinize] [quality_metrics] [variance_based_decomp]     \
        [samples = <INTEGER>] [fixed_sequence]                   \
        [sequence_start = <LISTof><INTEGER>]                     \
        [sequence_leap = <LISTof><INTEGER>]                      \
        [prime_base = <LISTof><INTEGER>] )                       \
|                                                                \
( {fsu_cvt}                                                      \
```

```
                [latinize] [quality_metrics] [variance_based_decomp]    \
                [seed = <INTEGER>] [fixed_seed]                         \
                [samples = <INTEGER>]                                   \
                [ {trial_type} {grid} | {halton} | {random} ]           \
                [num_trials = <INTEGER>] )                              \
        |                                                               \
        ( {vector_parameter_study}                                      \
                ( {final_point = <LISTof><REAL>}                        \
                  {step_length = <REAL>} | {num_steps = <INTEGER>} )    \
                |                                                       \
                ( {step_vector = <LISTof><REAL>}                        \
                  {num_steps = <INTEGER>} ) )                           \
        |                                                               \
        ( {list_parameter_study}                                        \
                {list_of_points = <LISTof><REAL>} )                     \
        |                                                               \
        ( {centered_parameter_study}                                    \
                {percent_delta = <REAL>}                                \
                {deltas_per_variable = <INTEGER>} )                     \
        |                                                               \
        ( {multidim_parameter_study}                                    \
                {partitions = <LISTof><INTEGER>} )

<KEYWORD = model>, <FUNCTION = model_kwhandler>                         \
        [id_model = <STRING>]                                           \
        [variables_pointer = <STRING>]                                  \
        [responses_pointer = <STRING>]                                  \
        ( {single}                                                      \
                [interface_pointer = <STRING>] )                        \
        |                                                               \
        ( {surrogate}                                                   \
                [id_surrogates = <LISTof><INTEGER>]                     \
                ( {global}                                              \
                  {mars} | {hermite} |{neural_network} |               \
                  {gaussian_process} |                                  \
                  ( {polynomial} {linear} | {quadratic} | {cubic} ) |   \
                  ( {kriging} [correlations = <LISTof><REAL>] )         \
                  [dace_method_pointer = <STRING>]                      \
                  [ {reuse_samples} {all} | {region} |                  \
                    {samples_file = <STRING>} ] [use_gradients]         \
#                 [ {rebuild} {inactive_all} | {inactive_region} ]      \
                  [ {correction}                                        \
                    {zeroth_order} | {first_order} | {second_order}     \
                    {additive} |  {multiplicative} | {combined} ] )     \
                |                                                       \
                ( {multipoint} {tana}                                   \
                  {actual_model_pointer = <STRING>} )                   \
                |                                                       \
                ( {local} {taylor_series}                               \
                  {actual_model_pointer = <STRING>} )                   \
                |                                                       \
                ( {hierarchical}                                        \
                  {low_fidelity_model_pointer  = <STRING>}              \
                  {high_fidelity_model_pointer = <STRING>}              \
#                 {model_hierarchy_pointers = <LISTof><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>}                       \
#               ( {sub_method_pointers = <LISTof><STRING>}              \
                  [primary_variable_mapping   = <LISTof><STRING>]       \
                  [secondary_variable_mapping = <LISTof><STRING>]       \
                  [primary_response_mapping   = <LISTof><REAL>]         \
                  [secondary_response_mapping = <LISTof><REAL>] ) )

<KEYWORD = variables>, <FUNCTION = variables_kwhandler>                 \
        [id_variables = <STRING>]                                       \
        [ {continuous_design = <INTEGER>}                               \
                [cdv_initial_point = <LISTof><REAL>]                    \
                [cdv_lower_bounds = <LISTof><REAL>]                     \
                [cdv_upper_bounds = <LISTof><REAL>]                     \
                [cdv_scales = <LISTof><REAL>]                           \
                [cdv_descriptors = <LISTof><STRING>] ]                  \
        [ {discrete_design = <INTEGER>}                                 \
                [ddv_initial_point = <LISTof><INTEGER>]                 \
                [ddv_lower_bounds = <LISTof><INTEGER>]                  \
                [ddv_upper_bounds = <LISTof><INTEGER>]                  \
                [ddv_descriptors = <LISTof><STRING>] ]                  \
        [ {normal_uncertain = <INTEGER>}                                \
                {nuv_means = <LISTof><REAL>}                            \
                {nuv_std_deviations = <LISTof><REAL>}                   \
                [nuv_lower_bounds = <LISTof><REAL>]                     \
                [nuv_upper_bounds = <LISTof><REAL>]                     \
                [nuv_descriptors = <LISTof><STRING>] ]                  \
        [ {lognormal_uncertain = <INTEGER>}                             \
                {lnuv_means = <LISTof><REAL>}                           \
                {lnuv_std_deviations = <LISTof><REAL>}                  \
              | {lnuv_error_factors = <LISTof><REAL>}                   \
                [lnuv_lower_bounds = <LISTof><REAL>]                    \
                [lnuv_upper_bounds = <LISTof><REAL>]                    \
                [lnuv_descriptors = <LISTof><STRING>] ]                 \
        [ {uniform_uncertain = <INTEGER>}                               \
                {uuv_lower_bounds = <LISTof><REAL>}                     \
                {uuv_upper_bounds = <LISTof><REAL>}                     \
                [uuv_descriptors = <LISTof><STRING>] ]                  \
        [ {loguniform_uncertain = <INTEGER>}                            \
                {luuv_lower_bounds = <LISTof><REAL>}                    \
                {luuv_upper_bounds = <LISTof><REAL>}                    \
                [luuv_descriptors = <LISTof><STRING>] ]                 \
        [ {triangular_uncertain = <INTEGER>}                            \
                {tuv_modes = <LISTof><REAL>}                            \
                {tuv_lower_bounds = <LISTof><REAL>}                     \
                {tuv_upper_bounds = <LISTof><REAL>}                     \
                [tuv_descriptors = <LISTof><STRING>] ]                  \
        [ {beta_uncertain = <INTEGER>}                                  \
                {buv_alphas = <LISTof><REAL>}                           \
                {buv_betas = <LISTof><REAL>}                            \
                {buv_lower_bounds = <LISTof><REAL>}                     \
                {buv_upper_bounds = <LISTof><REAL>}                     \
                [buv_descriptors = <LISTof><STRING>] ]                  \
        [ {gamma_uncertain = <INTEGER>}                                 \
                {gauv_alphas = <LISTof><REAL>}                          \
                {gauv_betas = <LISTof><REAL>}                           \
                [gauv_descriptors = <LISTof><STRING>] ]                 \
        [ {gumbel_uncertain = <INTEGER>}                                \
                {guuv_alphas = <LISTof><REAL>}                          \
                {guuv_betas = <LISTof><REAL>}                           \
                [guuv_descriptors = <LISTof><STRING>] ]                 \
        [ {frechet_uncertain = <INTEGER>}                               \
                {fuv_alphas = <LISTof><REAL>}                           \
```

```
                {fuv_betas = <LISTof><REAL>}                            \
                [fuv_descriptors = <LISTof><STRING>] ]                  \
        [ {weibull_uncertain = <INTEGER>}                               \
                {wuv_alphas = <LISTof><REAL>}                           \
                {wuv_betas = <LISTof><REAL>}                            \
                [wuv_descriptors = <LISTof><STRING>] ]                  \
        [ {histogram_uncertain = <INTEGER>}                             \
                [ {huv_num_bin_pairs = <LISTof><INTEGER>}               \
                  {huv_bin_pairs = <LISTof><REAL>} ]                    \
                [ {huv_num_point_pairs = <LISTof><INTEGER>}             \
                  {huv_point_pairs = <LISTof><REAL>} ]                  \
                [huv_descriptors = <LISTof><STRING>] ]                  \
        [ {interval_uncertain = <INTEGER>}                              \
                {iuv_num_intervals = <LISTof><INTEGER>}                 \
                {iuv_interval_probs = <LISTof><REAL>}                   \
                {iuv_interval_bounds = <LISTof><REAL>}                  \
                [iuv_descriptors = <LISTof><STRING>] ]                  \
        [uncertain_correlation_matrix = <LISTof><REAL>]                 \
        [ {continuous_state = <INTEGER>}                                \
                [csv_initial_state = <LISTof><REAL>]                    \
                [csv_lower_bounds = <LISTof><REAL>]                     \
                [csv_upper_bounds = <LISTof><REAL>]                     \
                [csv_descriptors = <LISTof><STRING>] ]                  \
        [ {discrete_state = <INTEGER>}                                  \
                [dsv_initial_state = <LISTof><INTEGER>]                 \
                [dsv_lower_bounds = <LISTof><INTEGER>]                  \
                [dsv_upper_bounds = <LISTof><INTEGER>]                  \
                [dsv_descriptors = <LISTof><STRING>] ]

<KEYWORD = interface>, <FUNCTION = interface_kwhandler>                 \
        [id_interface = <STRING>]                                       \
        [algebraic_mappings = <STRING>]                                 \
        [ {analysis_drivers = <LISTof><STRING>}                         \
          [analysis_components = <LISTof><STRING>]                      \
          [input_filter = <STRING>]                                     \
          [output_filter = <STRING>]                                    \
          ( {system} [analysis_usage = <STRING>]                        \
                [parameters_file = <STRING>]                            \
                [results_file = <STRING>]                               \
                [aprepro] [file_tag] [file_save] )                      \
          |                                                             \
          ( {fork}                                                      \
                [parameters_file = <STRING>]                            \
                [results_file = <STRING>]                               \
                [aprepro] [file_tag] [file_save] )                      \
          |                                                             \
          ( {direct}                                                    \
#               [processors_per_analysis = <LISTof><INTEGER>]           \
                [processors_per_analysis = <INTEGER>] )                 \
          |                                                             \
          {grid}                                                        \
          [ {failure_capture} {abort} | {retry = <INTEGER>} |           \
            {recover = <LISTof><REAL>} | {continuation} ]               \
          [ {deactivate} [active_set_vector] [evaluation_cache]         \
                         [restart_file] ] ]                             \
        [ {asynchronous} [evaluation_concurrency = <INTEGER>]           \
                         [analysis_concurrency = <INTEGER>] ]           \
        [evaluation_servers = <INTEGER>]                                \
        [evaluation_self_scheduling] [evaluation_static_scheduling]     \
        [analysis_servers = <INTEGER>]                                  \
        [analysis_self_scheduling]   [analysis_static_scheduling]
```

```
<KEYWORD = responses>, <FUNCTION = responses_kwhandler>                \
        [id_responses = <STRING>]                                      \
        [response_descriptors = <LISTof><STRING>]                      \
        ( {num_objective_functions = <INTEGER>}                        \
          [objective_function_scales = <LISTof><REAL>]                 \
          [multi_objective_weights = <LISTof><REAL>]                   \
          [ {num_nonlinear_inequality_constraints = <INTEGER>}         \
            [nonlinear_inequality_lower_bounds = <LISTof><REAL>]       \
            [nonlinear_inequality_upper_bounds = <LISTof><REAL>]       \
            [nonlinear_inequality_scales = <LISTof><REAL> ] ]          \
          [ {num_nonlinear_equality_constraints = <INTEGER>}           \
            [nonlinear_equality_targets = <LISTof><REAL>]              \
            [nonlinear_equality_scales = <LISTof><REAL>] ] )           \
        |                                                              \
        ( {num_least_squares_terms = <INTEGER>}                        \
          [least_squares_term_scales = <LISTof><REAL>]                 \
          [ {num_nonlinear_inequality_constraints = <INTEGER>}         \
            [nonlinear_inequality_lower_bounds = <LISTof><REAL>]       \
            [nonlinear_inequality_upper_bounds = <LISTof><REAL>]       \
            [nonlinear_inequality_scales = <LISTof><REAL> ] ]          \
          [ {num_nonlinear_equality_constraints = <INTEGER>}           \
            [nonlinear_equality_targets = <LISTof><REAL>]              \
            [nonlinear_equality_scales = <LISTof><REAL>] ] )           \
        |                                                              \
        {num_response_functions = <INTEGER>}                           \
        {no_gradients}                                                 \
        |                                                              \
        ( {numerical_gradients}                                        \
                [ {method_source} {dakota} | {vendor} ]                \
                [ {interval_type} {forward} | {central} ]              \
                [fd_gradient_step_size = <LISTof><REAL>] )             \
        |                                                              \
        {analytic_gradients}                                           \
        |                                                              \
        ( {mixed_gradients}                                            \
                {id_numerical_gradients = <LISTof><INTEGER>}           \
                  [ {method_source} {dakota} | {vendor} ]              \
                  [ {interval_type} {forward} | {central} ]            \
                  [fd_gradient_step_size = <LISTof><REAL>]             \
                {id_analytic_gradients = <LISTof><INTEGER>} )          \
        {no_hessians}                                                  \
        |                                                              \
        ( {numerical_hessians}                                         \
                [fd_hessian_step_size = <LISTof><REAL>] )              \
        |                                                              \
        ( {quasi_hessians} ( {bfgs} [damped] ) | {sr1} )               \
        |                                                              \
        {analytic_hessians}                                            \
        |                                                              \
        ( {mixed_hessians}                                             \
                [ {id_numerical_hessians = <LISTof><INTEGER>}          \
                  [fd_hessian_step_size = <LISTof><REAL>] ]            \
                [ {id_quasi_hessians = <LISTof><INTEGER>}              \
                  ( {bfgs} [damped] ) | {sr1} ]                        \
                [id_analytic_hessians = <LISTof><INTEGER>] )
```

Chapter 3

Introduction

3.1 Overview

In the DAKOTA system, a *strategy* creates and manages *iterators* and *models*. A model, generally speaking, contains a set of *variables*, an *interface*, and a set of *responses*, and the iterator operates on the model to map the variables into responses using the interface. Each of these 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. The number of strategies which can be invoked during a DAKOTA execution is limited to one. This strategy, however, may invoke multiple methods. Furthermore, each method may have its own model, consisting of (generally speaking) its own set of variables, its own interface, and its own set of responses. Thus, there may be multiple specifications of the method, model, variables, interface, and responses sections.

The syntax of DAKOTA specification is governed by the Input Deck Reader (IDR) parsing system [Weatherby et al., 1996], which uses the dakota.input.spec file to describe the allowable inputs to the system. This input specification file, then, provides a quick reference to the allowable system inputs from which a particular input file (e.g., `dakota.in`) can be derived.

This Reference Manual focuses on providing complete details for the allowable specifications in an input file 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 [Eldred et al., 2006].

3.2 IDR Input Specification File

DAKOTA input is governed by the IDR input specification file. This file (dakota.input.spec) is used by a code generator to create parsing system components which are compiled into the DAKOTA executable (refer to **Instructions for Modifying DAKOTA's Input Specification** for additional information). Therefore, dakota.input.spec is the definitive source for input syntax, capability options, and optional and required capability sub-parameters. Beginning users may find this file more confusing than helpful and, in this case, adaptation of example input files to a particular problem may be a more effective approach. However, advanced users can master all of the various

input specification possibilities once the structure of the input specification file is understood.

Refer to the dakota.input.spec documentation for a listing of the current version and discussion of specification features. From this file listing, it can be seen that the main structure of the strategy specification is that of several required group specifications separated by logical OR's: either multi-level OR surrogate-based optimization 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 simply that of a set of optional method-independent settings followed by a long list of possible methods appearing as required group specifications (containing a variety of method-dependent settings) separated by OR's. The model keyword reflects a structure of three required group specifications separated by OR's. Within the surrogate model type, the type of approximation must be specified with either a global OR multipoint OR local OR hierarchical required group specification. The structure of the variables keyword is that of optional group specifications for continuous and discrete design variables, a number of different uncertain variable distribution types, and continuous and discrete state variables. Each of these specifications can either appear or not appear as a group. Next, the interface keyword allows the specification of either algebraic mappings, simulation-based analysis driver mappings, or both. Within the analysis drivers specification, a system OR fork OR direct OR grid group specification must be selected. Finally, within the responses keyword, the primary structure is the required specification of the function set (either optimization functions OR least squares 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 IDR specification logic and rules, refer to [Weatherby et al., 1996].

3.3 Common Specification Mistakes

Spelling and omission of required parameters are the most common errors. Less obvious errors include:

- Since keywords are terminated with the newline character, care must be taken to avoid following the backslash character with any white space since the newline character will not be properly escaped, resulting in parsing errors due to the truncation of the keyword specification.

- Care must be taken to include newline escapes when embedding comments within a keyword specification. That is, newline characters will signal the end of a keyword specification even if they are part of a comment line. For example, the following specification will be truncated because one of the embedded comments neglects to escape the newline:

```
# No error here: newline need not be escaped since comment is not embedded
responses,                                          \
# No error here: newline is escaped                 \
        num_objective_functions = 1                 \
# Error here: this comment must escape the newline
        analytic_gradients                          \
        no_hessians
```

- Documentation of new capability sometimes lags the use of new capability in executables (especially experimental VOTD executables from nightly builds). When parsing errors occur which 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.

In most cases, the IDR system provides helpful error messages which will help the user isolate the source of the parsing problem.

3.4 Sample dakota.in Files

A DAKOTA input file is a collection of the fields allowed in the dakota.input.spec specification file which describe the problem to be solved by the DAKOTA system. Several examples follow.

3.4.1 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 in the test directory as `Dakota/test/dakota_textbook.in`.

```
strategy,\
single_method

method,\
dot_mmfd \
  max_iterations = 50 \
  convergence_tolerance = 1e-4 \
  output verbose

model,\
single

variables,\
continuous_design = 2 \
  cdv_initial_point    0.9    1.1 \
  cdv_upper_bounds     5.8    2.9 \
  cdv_lower_bounds     0.5   -2.9 \
  cdv_descriptor       'x1'   'x2'

interface,\
system \
  analysis_driver = 'text_book'\
  parameters_file = 'text_book.in'\
  results_file    = 'text_book.out'\
  file_tag file_save

responses,\
num_objective_functions = 1 \
num_nonlinear_inequality_constraints = 2 \
analytic_gradients \
no_hessians
```

3.4.2 Sample 2: Least Squares

The following sample input file shows a nonlinear least squares solution of the Rosenbrock Example using the NL2SOL method. A similar file is available in the test directory as `Dakota/test/dakota_rosenbrock.in`.

```
strategy,\
single_method

method,                                          \
nl2sol                    \
  max_iterations = 50 \
  convergence_tolerance = 1e-4

model,\
single

variables,\
continuous_design = 2 \
  cdv_initial_point   -1.2 1.0 \
  cdv_lower_bounds    -2.0     -2.0 \
  cdv_upper_bounds     2.0 2.0 \
  cdv_descriptor       'x1'     'x2'

interface,\
system \
  analysis_driver = 'rosenbrock'

responses,                                       \
num_least_squares_terms = 2 \
analytic_gradients \
no_hessians
```

3.4.3 Sample 3: Nondeterministic Analysis

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

```
strategy,\
single_method graphics

method,\
        nond_sampling \
  samples = 100 seed = 12345 \
  sample_type lhs                                        \
  response_levels = 3.6e+11 6.e+04 3.5e+05

model,\
single

variables,\
normal_uncertain = 2 \
  nuv_means             =  248.89, 593.33 \
  nuv_std_deviations    =   12.4,   29.7 \
  nuv_descriptor        =  'TF1n'   'TF2n'\
uniform_uncertain = 2 \
  uuv_dist_lower_bounds =  199.3,  474.63 \
  uuv_dist_upper_bounds =  298.5,  712. \
  uuv_descriptor        =  'TF1u'   'TF2u'\
weibull_uncertain = 2 \
  wuv_alphas            =   12.,    30. \
  wuv_betas             =  250.,   590. \
  wuv_descriptor        =  'TF1w'   'TF2w'
```

```
interface,\
system asynch evaluation_concurrency = 5 \
  analysis_driver = 'text_book'

responses,\
num_response_functions = 3 \
no_gradients \
no_hessians
```

3.4.4 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). A similar file is available in the test directory as `Dakota/test/dakota_pstudy`.in.

```
method,\
       vector_parameter_study \
  step_vector = .1 .1 .1 \
  num_steps = 4

variables,\
continuous_design = 3 \
  cdv_initial_point       1.0 1.0 1.0

interface,\
system asynchronous                  \
  analysis_driver = 'text_book'

responses,\
num_objective_functions = 1 \
num_nonlinear_inequality_constraints = 2 \
analytic_gradients \
analytic_hessians
```

3.4.5 Sample 5: Multilevel Hybrid Strategy

The following sample input file shows a multilevel 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 in the test directory as `Dakota/test/dakota_multilevel`.in.

```
strategy,\
graphics \
multi_level uncoupled \
  method_list = 'GA' 'PS' 'NLP'

method,\
id_method = 'GA'\
model_pointer = 'M1'\
coliny_ea \
```

```
  population_size = 10 \
  output verbose

method,\
id_method = 'PS'\
model_pointer = 'M1'\
coliny_pattern_search stochastic \
  output verbose \
  initial_delta = 0.1 \
  threshold_delta = 1.e-4 \
  solution_accuracy = 1.e-10 \
  exploratory_moves best_first

method,\
id_method = 'NLP'\
model_pointer = 'M2'\
        optpp_newton \
  gradient_tolerance = 1.e-12 \
  convergence_tolerance = 1.e-15

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 \
  cdv_initial_point    0.6    0.7 \
  cdv_upper_bounds     5.8    2.9       \
  cdv_lower_bounds     0.5   -2.9       \
  cdv_descriptor       'x1'   'x2'

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

responses,\
id_responses = 'R1'\
num_objective_functions = 1 \
no_gradients \
no_hessians

responses,\
id_responses = 'R2'\
num_objective_functions = 1 \
analytic_gradients \
analytic_hessians
```

Additional example input files, as well as the corresponding output and graphics, are provided in the Getting Started chapter of the Users Manual [Eldred et al., 2006].

3.5 Tabular descriptions

In the following discussions of keyword specifications, tabular formats (Tables 4.1 through 9.10) are used to present a short description of the specification, the keyword used in the specification, the type of data associated with the keyword, the status of the specification (required, optional, required group, or optional group), and the default for an optional specification.

It can be difficult to capture in a simple tabular format the complex relationships that can occur when specifications are nested within multiple groupings. For example, in the model keyword, the `actual_model_pointer` specification is a required specification within the `multipoint` and `local` required group specifications, which are separated from each other and from other required group specifications (`global` and `hierarchical`) by logical OR's. The selection between the `global`, `multipoint`, `local`, or `hierarchical` required groups is contained within another required group specification (`surrogate`), which is separated from the `single` and `nested` required group specifications by logical OR's. Rather than unnecessarily proliferate the number of tables in attempting to capture all of these inter-relationships, a balance is sought, since some inter-relationships are more easily discussed in the associated text. The general structure of the following sections is to present the outermost specification groups first (e.g., `single`, `surrogate`, or `nested` in Table 6.1), followed by lower levels of group specifications (e.g., `global`, `multipoint`, `local`, or `hierarchical` surrogates in Table 6.3), followed by the components of each group (e.g., Tables 6.4 through 6.8) in succession.

Chapter 4

Strategy Commands

4.1 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. Five strategies currently exist: `multi_level`, `surrogate_based_opt`, `multi_start`, `pareto_set`, and `single_method`. These algorithms are implemented within the **Strategy** "Strategy" class hierarchy in the **MultilevelOptStrategy**, **SurrBasedOptStrategy**, **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 [Eldred et al., 2006].

In a multi-level hybrid optimization strategy (`multi_level`), a list of methods is specified which will be used synergistically in seeking an optimal design. The goal here is to exploit the strengths of different optimization algorithms through different stages of the optimization 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 surrogate-based optimization (`surrogate_based_opt`), optimization occurs using an approximation, i.e., a surrogate model, that is 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. A trust region strategy is used to manage the optimization 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 optimization runs performed on the surrogate model and bounded by the trust region. At the end of each approximate optimization, the candidate optimum point found 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, an optimization run, and another test for sufficient decrease in the truth model. This cycle continues until convergence is attained. The goals of surrogate-based optimization 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.

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 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 a "fall through" 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.

Each of the strategy specifications identifies one or more method pointers (e.g., `method_list`, `opt_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 `multi_level` example is:

```
strategy,  \
multi_level uncoupled         \
  method_list = 'GA', 'PS', 'NLP'
```

A `surrogate_based_opt` example specification is:

```
strategy,  \
        graphics                                \
surrogate_based_opt \
  opt_method_pointer = 'NLP1'        \
  trust_region initial_size = 0.10
```

A `multi_start` example specification is:

```
strategy,  \
```

```
multi_start \
  method_pointer = 'NLP1'               \
  random_starts = 10
```

A `pareto_set` example specification is:

```
strategy,  \
pareto_set \
  opt_method_pointer = 'NLP1'           \
          random_weight_sets = 10
```

And finally, a `single_method` example specification is:

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

4.2 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: `multi_level`, `surrogate_based_opt`, `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.spec, 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 (`multi_level` OR `surrogate_based_opt` 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.

4.3 Strategy Independent Controls

The strategy independent controls include `graphics`, `tabular_graphics_data`, `tabular_graphics_file`, `iterator_servers`, `iterator_self_scheduling`, and `iterator_static_scheduling`. 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. For applications utilizing approximations over 2 variables, a 3D graphics window containing a surface plot of the approximation will also be activated. 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 3rd 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. 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`. Table 4.1 summarizes the strategy independent controls.

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`
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 4.1: Specification detail for strategy independent controls

4.4 Multilevel Hybrid Optimization Commands

The multi-level hybrid optimization strategy has `uncoupled`, `uncoupled adaptive`, and `coupled` approaches (see the Users Manual for more information on the algorithms employed). In the two uncoupled approaches, 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. The uncoupled adaptive approach may be specified by turning on the `adaptive` flag. If this flag in specified, then `progress_threshold` must also be specified since it is a required part of adaptive specification. In the nonadaptive case, method switching is managed through the separate convergence controls of each method. In the adaptive case, however, method

switching occurs when the internal progress metric (normalized between 0.0 and 1.0) falls below the user specified `progress_threshold`. Table 4.2 summarizes the uncoupled multi-level strategy inputs.

Description	Keyword	Associated Data	Status	Default
Multi-level hybrid strategy	`multi_level`	none	Required group (1 of 6 selections)	N/A
Uncoupled hybrid	`uncoupled`	none	Required group (1 of 2 selections)	N/A
Adaptive flag	`uncoupled`	none	Optional group	nonadaptive hybrid
Adaptive progress threshold	`progress_-threshold`	real	Required	N/A
List of methods	`method_list`	list of strings	Required	N/A

Table 4.2: Specification detail for uncoupled multi-level strategies

In the `coupled` approach, global and local method strings supplied with the `global_method_pointer` and `local_method_pointer` specifications identify the two methods to be used. The `local_search_-probability` setting is an optional specification for supplying the probability (between 0.0 and 1.0) of employing local search to improve estimates within the global search. Table 4.3 summarizes the coupled multi-level strategy inputs.

Description	Keyword	Associated Data	Status	Default
Multi-level hybrid strategy	`multi_level`	none	Required group (1 of 6 selections)	N/A
Coupled hybrid	`coupled`	none	Required group (1 of 2 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 4.3: Specification detail for coupled multi-level strategies

4.5 Surrogate-based Optimization (SBO) Commands

The `surrogate_based_opt` strategy must specify an optimization method using `opt_method_pointer`. The method specification identified by `opt_method_pointer` is then responsible for using its `model_-pointer` (see Method Independent Controls) to select a `surrogate` model (see Surrogate Model Controls). SBO algorithm controls include `max_iterations` (the maximum number of SBO cycles allowed),

`convergence_tolerance` (the relative tolerance used in internal SBO convergence assessments), `soft_convergence_limit` (a soft convergence control for the SBO 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). In addition, 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 SBO iteration. The command `expand_threshold` determines the trust region value above which the trust region will expand for the next SBO iteration. 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. Tables 4.4 and 4.5 summarize the surrogate based optimization strategy inputs.

Description	Keyword	Associated Data	Status	Default
Surrogate-based optimization strategy	`surrogate_based_opt`	none	Required group (1 of 6 selections)	N/A
Optimization method pointer	`opt_method_pointer`	string	Required	N/A
Maximum number of SBO iterations	`max_iterations`	integer	Optional	100
Convergence tolerance for SBO iterations	`convergence_tolerance`	real	Optional	1.e-4
Soft convergence limit for SBO iterations	`soft_convergence_limit`	integer	Optional	5
Flag for bypassing lower level surrogates in truth verifications	`truth_surrogate_bypass`	none	Optional	no bypass

Table 4.4: Specification detail for surrogate based optimization strategies

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
Constraint relaxation method for infeasible iterates	constraint_-relax	homotopy	Optional	no relaxation

Table 4.5: Specification detail for trust region controls in surrogate based optimization strategies

4.6 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 4.7 summarizes the multi-start strategy inputs.

Description	Keyword	Associated Data	Status	Default
Multi-start iteration strategy	`multi_start`	none	Required group (1 of 6 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 4.6: Specification detail for multi-start strategies

4.7 Pareto Set Optimization Commands

The `pareto_set` strategy must specify an optimization method using `opt_method_pointer`. This optimizer is responsible for computing a set of optimal solutions from a set of multiobjective weightings. 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 `multi_objective_weight_sets`, in which the weighting sets are specified in a list, or (3) using both `random_weight_sets` and `multi_objective_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 4.8 summarizes the pareto set strategy inputs.

4.8 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`

Description	Keyword	Associated Data	Status	Default
Pareto set optimization strategy	pareto_set	none	Required group (1 of 6 selections)	N/A
Optimization method pointer	opt_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	multi_objective_weight_sets	list of reals	Optional	no user-specified weighting sets

Table 4.7: Specification detail for pareto set strategies

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 4.9 summarizes the single method strategy inputs.

Description	Keyword	Associated Data	Status	Default
Single method strategy	single_method	string	Required group (1 of 6 selections)	N/A
Method pointer	method_pointer	string	Optional	use of last method parsed

Table 4.8: Specification detail for single method strategies

Chapter 5

Method Commands

5.1 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 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 for strategy algorithm descriptions).

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

```
method,        \
dot_sqp
```

This example uses all of the defaults for this method.

A more sophisticated example would be

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

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

DOT method independent controls for DOT usage of these controls). The optimization_type control is a method dependent control, in that it is only meaningful for DOT methods (see DOT method dependent controls).

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

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

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

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

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

The last example shows a specification for a parameter study iterator 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
```

5.2 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, reduced_sqp, optpp_cg, optpp_q_newton, optpp_fd_newton, optpp_g_newton, optpp_newton, optpp_pds, coliny_apps, coliny_cobyla, coliny_direct, coliny_pattern_search, coliny_solis_wets, coliny_ea, moga, soga, nond_polynomial_chaos, nond_sampling, nond_reliability, nond_evidence, dace, fsu_quasi_mc, fsu_cvt, 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.spec, 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.

5.3 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 Coliny) 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 Coliny 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.). This control is not currently used within the uncertainty quantification, design of experiments, and parameter study iterator branches, and in the case of optimization and least squares, 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).

Continuous design variable, function, and constraint scaling can be turned on for optimizers and least squares minimizers by providing the `scaling` keyword. 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 choose to specify no, one, or a vector of scale values through each of the `cdv_scales` (see Variables Commands); `objective_function_scales`, `least_squares_term_scales`, `nonlinear_inequality_scales`, `nonlinear_equality_scales` (see Function Specification); `linear_inequality_scales`, and `linear_equality_scales` (see Method Independent Controls below) specifications. Discrete variable scaling is not supported. If a single value is specified using any of these keywords it will apply to each component of the relevant vector, e.g., `cdv_scales = 3.0` will apply a characteristic scaling value of `3.0` to each continuous design variable. Valid entries in `*_scales` vectors include positive characteristic values (user-specified scale factors), `1.0` to exempt a component from scaling, or `0.0` for automatic scaling, if available for that component. Negative scale values are not currently permitted. When the `scaling` keyword is omitted, all `*_scales` specifications are ignored in the method, variables, and responses sections. When scaling is enabled, the following progression will be used to determine the type of scaling used on each component of a variables or response vector:

1. When a strictly positive characteristic value is specified, the quantity will be scaled by it.
2. If a zero or no characteristic value is specified, automatic scaling will be attempted according to the following scheme:
   - two-sided bounds scaled into the interval [0,1];
   - one-sided bound or targets scaled by the absolute value of the characteristic value, moving the bound or target to -1 or +1.
   - no bounds or targets: no automatic scaling possible, therefore no scaling for this component

Automatic scaling is not available for objective functions or least squares terms since they do not have bound constraints. Futher, when automatically scaled, linear constraints are scaled by characteristic values only, not affinely scaled into [0,1]. *Caution:* The scaling hierarchy is followed for all problem variables and constraints when the `scaling` keyword is specified, so one must note the default scaling behavior for each component and manually exempt components with a scale value of `1.0`, if necessary.

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

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` | `verbose` | `debug`	Optional	`normal`
Maximum iterations	`max_iterations`	integer	Optional	optimization/least squares: 100, `fsu_cvt`: 30, `nond_reliability`: 10
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
Scaling flag	`scaling`	none	Optional	no scaling

Table 5.1: Specification detail for the method independent controls: identifiers, pointers, tolerances, limits, output verbosity, speculative gradients, and scaling

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

$$a_l \leq Ax \leq a_u$$

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

$$Ax \leq 0.0$$

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

$$Ax = a_t$$

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

$$Ax = 0.0$$

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

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

$$x_1 + x_2 + x_3 \geq 2.0$$

$$x_1 + x_2 - x_3 = 1.0$$

which is not to be confused with something like

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

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

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

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

The `linear_inequality_scales` or `linear_equality_scales` specifications provide nonnegative real values for scaling of 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 these vectors may be a user-specified characteristic value, `0.` for automatic scaling, or `1.` for no scaling. If a single real value is specified it will apply to each constraint component. Scaling for linear constraints specified through `linear_inequality_scales` or `linear_equality_scales` is applied *after* any continuous variable scaling. For example, for variable scaling on continuous design variables x:

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

we have the following system for linear inequality constraints

$$a_L \leq A_i x \leq a_U$$

$$a_L \leq A_i \left(\text{diag}(x_M)\tilde{x} + x_O\right) \leq a_U$$

$$a_L - A_i x_O \leq A_i \text{diag}(x_M)\tilde{x} \leq a_U - A_i x_O$$

$$\tilde{a}_L \leq \tilde{A}_i \tilde{x} \leq \tilde{a}_U$$

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

Table 5.2 provides the specification detail for the method independent controls involving linear constraints.

Description	Keyword	Associated Data	Status	Default
Linear inequality coefficient matrix	`linear_inequality_constraint_matrix`	list of reals	Optional	no linear inequality constraints
Linear inequality lower bounds	`linear_inequality_lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Linear inequality upper bounds	`linear_inequality_upper_bounds`	list of reals	Optional	vector values = `0`.
Linear inequality scales	`linear_inequality_scales`	list of reals	Optional	no scaling (vector values = `1`.)
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 scales	`linear_equality_scales`	list of reals	Optional	no scaling (vector values = `1`.)

Table 5.2: Specification detail for the method independent controls: linear inequality and equality constraints

5.4 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.

5.4.1 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). In addition, if speculative is specified, then gradients (dakota numerical or analytic gradients) will be computed on each line search evaluation in order to balance the load and lower the total run time in parallel optimization studies. Lastly, specialized handling of linear constraints is supported with DOT; linear constraint coefficients, bounds, and targets can be provided to DOT at start-up and tracked internally. Specification detail for these method independent controls is provided in Tables 5.1 through 5.2.

5.4.2 DOT method dependent controls

DOT's only method dependent control is optimization_type which may be either minimize or maximize. DOT provides the only set of methods within DAKOTA which support this control; to convert a maximization problem into the minimization formulation assumed by other methods, simply change the sign on the objective function (i.e., multiply by -1). Table 5.3 provides the specification detail for the DOT methods and their method dependent controls.

Description	Keyword	Associated Data	Status	Default
Optimization type	optimization_type	minimize | maximize	Optional group	minimize

Table 5.3: Specification detail for the DOT methods

5.5 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.

5.5.1 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). An important related observation is the fact that NPSOL uses two different line searches depending on how gradients are computed. For either `analytic_gradients` or `numerical_gradients` with `method_source dakota`, NPSOL is placed in user-supplied gradient mode (NPSOL's "Derivative Level" is set to 3) and it uses a gradient-based line search (the assumption is that user-supplied gradients are inexpensive). On the other hand, if `numerical_gradients` are selected with `method_source vendor`, then NPSOL is computing finite differences internally and it will use a value-based line search (the assumption is that finite differencing on each line search evaluation is too expensive). The ramifications of this are: (1) performance will vary between `method_source dakota` and `method_source vendor` for `numerical_gradients`, and (2) gradient speculation is unnecessary when performing optimization in parallel since the gradient-based line search in user-supplied gradient mode is already load balanced for parallel execution. Therefore, a `speculative` specification will be ignored by NPSOL, and optimization with numerical gradients should select `method_source dakota` for load balanced parallel operation and `method_source vendor` for efficient serial operation.

Lastly, NPSOL supports specialized handling of linear inequality and equality constraints. By specifying the coefficients and bounds of the linear inequality constraints and the coefficients and targets of the linear equality constraints, this information can be provided to NPSOL at initialization and tracked internally, removing the need for the user to provide the values of the linear constraints on every function evaluation. Refer to Method Independent Controls for additional information and to Tables 5.1 through 5.2 for method independent control specification detail.

5.5.2 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 5.4: Specification detail for the NPSOL SQP method

5.6 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.

5.6.1 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.

5.6.2 NLPQL method dependent controls

NLPQL does not currently support any method dependent controls.

5.7 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.

5.7.1 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.

5.7.2 CONMIN method dependent controls

CONMIN does not currently support any method dependent controls.

5.8 OPT++ Methods

The OPT++ library [Meza, 1994] 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.

5.8.1 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 uncon-

strained and optpp_pds can be, at most, bound-constrained. Specification detail for the method independent controls is provided in Tables 5.1 through 5.2.

5.8.2 OPT++ method dependent controls

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

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

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

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

see [El-Bakry et al., 1996].

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

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

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

The `central_path` specification represents a measure of proximity to the central path and specifies an update strategy for the perturbation parameter mu. Refer to [Argaez et al., 2002] for a detailed discussion on proximity measures to the central region. Valid options are, again, "el_bakry", "argaez_tapia", or "van_shanno", where user input is not case sensitive. The default value for `central_path` is the value of `merit_function` (either user-selected or default). The `steplength_to_boundary` specification is a parameter (between 0 and 1) that controls how close to the boundary of the feasible region the algorithm is allowed to move. A value of 1 means that the algorithm is allowed to take steps that may reach the boundary of the feasible region. If the user wishes to maintain strict feasibility of the design parameters this value should be less than 1. Default values are .8, .99995, and .95 for the "el_bakry", "argaez_tapia", and "van_shanno" merit functions, respectively. The `centering_parameter` specification is a parameter (between 0 and 1) that controls how closely the algorithm should follow the "central path". See [Wright] for the definition of central path. The larger the value, the more closely the algorithm follows the central path, which results in small steps. A value of 0 indicates that the algorithm will take a pure Newton step. Default values are .2, .2, and .1 for the "el_bakry", "argaez_tapia", and "van_shanno" merit functions, respectively.

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

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.

5.9 SGOPT Methods

As of DAKOTA Version 4.0, the SGOPT Library has been deprecated and the methods have been incorporated into Coliny optimizers.

5.10 Coliny Methods

Coliny is a collection of nongradient-based optimizers that support the Common Optimization Library INterface (COLIN). These methods supersede the SGOPT methods which have been deprecated as of DAKOTA Version

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

Table 5.6: Specification detail for OPT++ Newton-based optimization methods

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 5.7: Specification detail for the OPT++ PDS method

4.0. Coliny optimizers currently include coliny_apps, coliny_cobyla, coliny_direct, coliny_ea, coliny_pattern_search and coliny_solis_wets. Additional Coliny information is available from http://software.sandia.gov/Acro/Coliny/.

Coliny solvers now support bound constraints and general nonlinear constraints. Supported nonlinear constraints include both equality and two-sided inequality constraints. Coliny solvers do not yet support linear constraints. Most Coliny 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.)

5.10.1 Coliny 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 Coliny 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 Coliny: the silent, quiet, and normal settings correspond to minimal reporting from Coliny, whereas the verbose setting corresponds to a higher level of information, and debug outputs method initialization and a variety of internal Coliny diagnostics. The majority of Coliny'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 Coliny. Specification detail for method independent controls is provided in Tables 5.1 through 5.2.

5.10.2 Coliny method dependent controls

All Coliny 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 Coliny, 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 Coliny 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 Coliny and understand any differences that exist between those specifications and the ones available through DAKOTA.

Each of the Coliny methods supports the solution_accuracy control, which defines a convergence criterion in which the optimizer will terminate if it finds an objective function value lower than the specified accuracy. Specification detail for method dependent controls for all Coliny methods is provided in Table 5.8.

Each Coliny method supplements the settings of Table 5.8 with controls which are specific to its particular class of method.

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 accuracy	solution_accuracy	real	Optional	-DBL_MAX

Table 5.8: Specification detail for Coliny method dependent controls

5.10.3 APPS

The asynchronous parallel pattern search (APPS) algorithm [Hough et al., 2000] 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. By default, it utilizes the nonblocking schedulers in DAKOTA (synchronization nonblocking). APPS is currently interfaced to DAKOTA as part of Coliny (method coliny_apps). APPS-specific software documentation is available from http://software.sandia.gov/appspack/.

The only method independent control currently mapped to APPS is the output verbosity control. The APPS internal "debug" level is mapped to the DAKOTA debug, verbose, normal, quiet, and silent settings as follows:

- DAKOTA "debug": APPS debug level = 7
- DAKOTA "verbose": APPS debug level = 4
- DAKOTA "normal": APPS debug level = 3
- DAKOTA "quiet": APPS debug level = 2
- DAKOTA ""silent": APPS debug level = 1

The APPS method is invoked using a coliny_apps group specification. The method dependent controls are a subset of the Coliny controls for coliny_pattern_search described in Pattern Search. In particular, APPS supports initial_delta, threshold_delta, and contraction_factor, and the APPS step lengths are dynamically rescaled like the steps in coliny_pattern_search. Coliny specifications such as pattern_basis, total_pattern_size, and no_expansion are not supported since 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. Table 5.9 summarizes the APPS specification.

5.10.4 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 con-

Description	Keyword	Associated Data	Status	Default
APPS method	coliny_apps	none	Required group	N/A
Initial offset value	initial_delta	real	Required	N/A
Threshold for offset values	threshold_delta	real	Required	N/A
Pattern contraction factor	contraction_factor	real	Optional	0.5
Evaluation synchronization	synchronization	blocking | nonblocking	Optional	nonblocking
Constraint penalty	constraint_penalty	real	Optional	1000.0

Table 5.9: Specification detail for the APPS method

straint 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_accuracy specifications. The search performed by COBYLA is currently not parallelized.

Table 5.10 summarizes the COBYLA specification.

Description	Keyword	Associated Data	Status	Default
COBYLA method	coliny_cobyla	none	Required group	N/A
Initial offset value	initial_delta	real	Required	N/A
Threshold for offset values	threshold_delta	real	Required	N/A

Table 5.10: Specification detail for the COBYLA method

5.10.5 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.

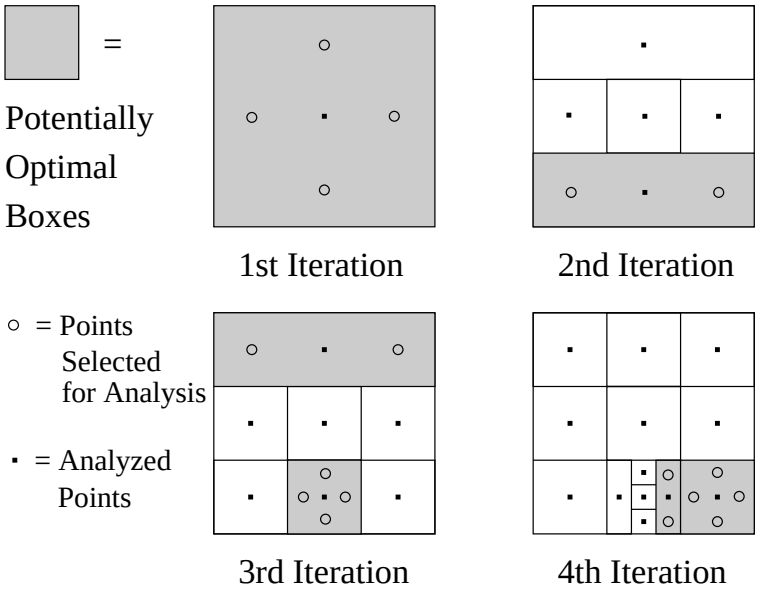

Figure 5.1: Design space partitioning with DIRECT

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_accuracy`, `constraint_penalty` and `show_misc_options` specifications that are described in Coliny 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_accuracy` 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 5.11: Specification detail for the DIRECT method

5.10.6 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 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.

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 5.12: Specification detail for the Coliny EA method dependent controls: seed, initialization, selection, and replacement

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

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

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

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

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

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

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

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 5.13: Specification detail for the Coliny EA method: crossover and mutation

generate a new offspring. The Coliny 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 Coliny EA does not utilize bit representations of variable values, the crossover points only occur on coordinate boundaries, never within the bits of a particular coordinate. Uniform crossover creates offspring through random combination of coordinates from the two parents. Blend crossover generates a new individual randomly along the multidimensional vector connecting the two parents.

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

The Coliny 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.

5.10.7 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 Coliny 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 Coliny 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.

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. 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 Coliny:

- The `basic_pattern` case is the simple pattern search approach, which uses the same pattern in each iteration.
- The `multi_step` case examines each trial step in the pattern in turn. If a successful step is found, the pattern search continues examining trial steps about this new point. In this manner, the effects of multiple successful steps are cumulative within a single iteration. This option does not support any parallelism and will result in a serial pattern search.
- The `adaptive_pattern` case invokes a pattern search technique that adaptively rescales the different search directions to maximize the number of redundant function evaluations. See [Hart et al., 2001] for details of this method. In preliminary experiments, this method had more robust performance than the standard `basic_pattern` case in serial tests. This option supports a limited degree of parallelism. After successful iterations (where the step length is not contracted), a parallel search will be performed. After unsuccessful iterations (where the step length is contracted), only a single evaluation is performed.

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

In general, pattern search methods can expand and contract their step lengths. Coliny 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 Coliny pattern search method and its method dependent controls.

Description	Keyword	Associated Data	Status	Default
Coliny pattern search method	coliny_pattern_search	none	Required group	N/A
Stochastic pattern search	stochastic	none	Optional group	N/A
Random seed for stochastic pattern search	seed	integer	Optional	randomly generated seed
Initial offset value	initial_delta	real	Required	N/A
Threshold for offset values	threshold_delta	real	Required	N/A
Constraint penalty	constraint_penalty	real	Optional	1.0
Control of dynamic penalty	constant_penalty	none	Optional	algorithm dynamically adapts the constraint penalty

Table 5.14: Specification detail for the Coliny 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	1
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 5.15: Specification detail for the Coliny pattern search method: pattern controls

5.10.8 Solis-Wets

DAKOTA's implementation of Coliny 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.

Description	Keyword	Associated Data	Status	Default
Coliny Solis-Wets method	`coliny_solis_wets`	none	Required group	N/A
Random seed for stochastic pattern search	`seed`	integer	Optional	randomly generated seed
Initial offset value	`initial_delta`	real	Required	N/A
Threshold for offset values	`threshold_delta`	real	Required	N/A
No expansion flag	`no_expansion`	none	Optional	algorithm may expand pattern size
Number of consecutive improvements before expansion	`expand_after_success`	integer	Optional	5
Number of consecutive failures before contraction	`contract_after_failure`	integer	Optional	3
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 5.16: Specification detail for the Coliny Solis-Wets method

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.

5.11 JEGA Methods

The JEGA (John Eddy's Genetic Algorithms) 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 Sustainment and Readiness group 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.

5.11.1 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 Table 5.20 below).

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

5.11.2 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 allowed)
4. Perform mutation (several mutation types are allowed)
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 populations. Choice of fitness assessor operators is strongly dependent on the type of algorithm being used and can have a profound effect on the choice of selectors. 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 recomended that you use 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` selector together. If using `SOGA`, the only fitness assessor is the `merit_function` fitness assessor which currently uses an

exterior penalty function formulation to assign fitnesses. Any of the selectors can be used in conjunction with this fitness assessment scheme.

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 `replacement_type` of `favor_feasible` is specific to a SOGA. This replacement operator will always take a feasible design over an infeasible one. Beyond that, it favors solutions based on an assigned fitness value which must have been installed by some fitness assessor.

8. Apply niche pressure to the population. This step is specific to the MOGA and is new to JEGA v2.0. Technically, the step is carried out during runs of the SOGA but only the `null_niching` operator is avaiable for use with SOGA. In MOGA, the `radial_niching` operator 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. 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. The final step in the iterator loop is to assess the convergence of the algorithm. 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 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. In 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.

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 5.17 and 5.18 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 to JEGA v2.0 is the introduction of the `log_file` specification. JEGA v2.0 uses a logging library to output messages and status to the user. JEGA can be configured at build time to log to both standard error 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 if 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 vari-

able 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 must be delimited with a tab in the input file. The input file will continue to 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 $10^{\wedge}6$ 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, 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.

5.11.3 Multi-objective Evolutionary Algorithms

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

The initialization, crossover, and mutation controls were all described in the preceding section. There are no MOGA specific aspects to these controls. The fitness_type for a MOGA may be domination_count or layer_rank. Both have been specifically designed to avoid problems with aggregating and scaling objective function values and transforming them into a single objective. Instead, the domination_count fitness assessor works by ordering population members by the negative of the number of designs that dominate them. The values are negated in keeping with the convention that higher fitness is better. The layer_rank fitness assessor works by assigning all non-dominated designs a layer of 0, then from what remains, assigning all the non-dominated a layer of -1, and so on until all designs have been assigned a layer. Again, the values are negated for the higher-is-better fitness convention. Use of the below_limit selector with the domination_count fitness assessor has the effect of keeping all designs that are dominated by fewer then a limiting number of other designs subject

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
Population Output	print_each_pop	none	Optional	N/A
Output verbosity	output	silent | quiet | verbose | debug	Optional	normal
Initialization type	initialization_type	simple_random | unique_random | flat_file	Required	unique_random
Mutation type	mutation_type	replace_uniform | bit_random | offset_cauchy | offset_uniform | offset_normal	Optional group	None
Mutation scale	mutation_scale	real	Optional	0.15
Mutation rate	mutation_rate	real	Optional	0.08
Replacement type	replacement_type	below_limit | roulette_wheel | unique_roulette_wheel	Required group	None
Below limit selection	below_limit	real	Optional	6
Shrinkage percentage in below limit selection	shrinkage_percentage	real	Optional	0.9

Table 5.17: 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	none
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 5.18: Specification detail for JEGA method dependent controls: crossover

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 a minimum.

New to 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 operator available is the `radial` nicher. This niche pressure applicator works by enforcing a minimum 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 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 niche pressure control consists of two options. The first is the `null_niching` option which specifies that no niche pressure is to be applied. The second is the `radial_niching` option which specifies that the radial niching algorithm is to be use. 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 distance defined by these percentages.

Aslo new to 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 5.19 below. Note that MOGA and SOGA create additional output files during execution. "finaldata.dat" is a file that holds the Pareto members of the population in the final generation. "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 multi-level 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.

Description	Keyword	Associated Data	Status	Default
Fitness type	fitness_type	layer_rank | domination_count	Required group	None
Niche pressure type	niching_type	radial_niching	Optional group	No niche pressure
Radial niching	radial_niching	list of real	Optional	0.01 for all objectives
Convergence type	metric_tracker	none	Optional group	Stopping criteria only
Percent change limit for metric_tracker converger	percent_change	real	Optional	0.1
Number generations for metric_tracker converger	num_generations	integer	Optional	10

Table 5.19: Specification detail for MOGA method controls

5.11.4 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, or favor_feasible. The favor_feasible replacement type always takes a feasible design over an infeasible one. Beyond that, it selects designs based on a fitness value. As of JEGA v2.0, the fitness assessment operator must be specified with SOGA although the merit_function is currently the only one. 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. Both also respect the stopping criteria.

The SOGA specific controls are described in Table 5.20 below.

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

Table 5.20: Specification detail for SOGA method controls

5.12 Least Squares Methods

DAKOTA's least squares branch currently contains three methods for solving nonlinear least squares problems: NL2SOL, a trust-region method that adaptively chooses between two Hessian approximations (Gauss-Newton and Gauss-Newton plus a quasi-Newton approximation to the rest of the Hessian), NLSSOL, a sequential quadratic programming (SQP) approach that is from the same algorithm family as NPSOL, and Gauss-Newton, which supplies the Gauss-Newton Hessian approximation to the full-Newton optimizers from OPT++.

The important difference of these algorithms from general-purpose optimization methods is that the response set is defined by least squares terms, 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 Least squares terms and constraint functions (least squares data set) for additional information on the least squares response data set.

5.12.1 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 5.21 provides the specification detail for the NL2SOL method dependent controls.

5.12.2 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.

5.12.3 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.)

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 5.21: Specification detail for NL2SOL method dependent controls.

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`.

5.13 Nondeterministic Methods

DAKOTA's nondeterministic methods do not make use of many method independent controls. Only the `x_taylor_mpp`, `u_taylor_mpp`, `x_two_point`, and `u_two_point` methods within `nond_reliability` use method independent convergence controls (see Reliability methods). As such, the nondeterministic branch documentation which follows is primarily limited to the method dependent controls for the sampling, reliability, and polynomial chaos expansion methods.

Each of these techniques supports `response_levels`, `probability_levels`, and `reliability_levels` specifications along with optional `num_response_levels`, `num_probability_levels`, and `num_reliability_levels` keys. 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 and/or reliabilities (forward mapping). The selection of probability or reliability results for the forward mapping can be performed with the `compute` keyword followed by either `probabilities` or `reliabilities`. Conversely, the `probability_levels` and `reliability_levels` specifications provide target levels for which response values will be computed (inverse mapping). The mapping results (probabilities or 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. The selection of a CDF or CCDF can be performed with the `distribution` keyword followed by either `cumulative` for the CDF option or `complementary` for the CCDF option. Table 5.22 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 probability distributions. Table 5.23 summarizes the uncertain variables that are available for use by the different methods, where a "C" denotes that correlations are supported between uncertain input variables of this type, a "U" means the uncertain input variables of this type must be uncorrelated, and a "-" indicates that the distribution is not supported by the method. Note that we have two versions of LHS available in DAKOTA, listed as old and new. The new LHS is preferred and is the default; however, it requires the presence of a Fortran 90 compiler which is not available on all platforms. The old version does not support correlations and has limited distribution support, but is still maintained due to portability. Additional details include: (1) old LHS lognormal distributions require `error_factor` specifications and do not support `std_deviation` specifications, and (2) reliability methods require

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	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

Table 5.22: Specification detail for forward/inverse level mappings

the presence of the GSL library within the build configuration in order to support Beta, Gamma, Frechet, and Weibull distributions.

Distribution Type	Old LHS	New LHS	Reliability	Evidence
Normal	U	C	C	-
Lognormal	U	C	C	-
Uniform	U	C	C	-
Loguniform	U	C	U	-
Triangular	-	C	U	-
Beta	-	C	U	-
Gamma	-	C	C	-
Gumbel	-	-	C	-
Frechet	-	-	C	-
Weibull	U	C	C	-
Histogram	U	C	-	-
Interval	-	-	-	U

Table 5.23: Summary of Distribution Types supported by Nondeterministic Methods

5.13.1 Nondeterministic sampling method

The nondeterministic sampling iterator is selected using the `nond_sampling` specification. This iterator 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 currently 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 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. 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.

The nondeterministic sampling iterator also supports a design of experiments mode through the `all_-`

variables flag. Normally, nond_sampling generates samples only for the uncertain variables, and treats any design or state variables as constants. The all_variables flag alters this behavior by instructing the sampling algorithm to treat any continuous design or continuous state variables as parameters with uniform probability distributions between their upper and lower bounds. Samples are then generated over all of the continuous variables (design, uncertain, and state) in the variables specification. This is similar to the behavior of the design of experiments methods described in Design of Computer Experiments Methods, since they will also generate samples over all continuous design, uncertain, and state variables in the variables specification. However, the design of experiments methods will treat all variables as being uniformly distributed between their upper and lower bounds, whereas the nond_sampling iterator will sample the uncertain variables within their specified probability distributions.

Finally, the nondeterministic sampling iterator 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, 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, Si, 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]. 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 5.24 provides details of the nondeterministic sampling specifications beyond those of Table 5.22.

Description	Keyword	Associated Data	Status	Default
Nondeterministic sampling iterator	nond_sampling	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
Number of samples	samples	integer	Optional	minimum required
Sampling type	sample_type	random | lhs	Optional group	lhs
All variables flag	all_variables	none	Optional	sampling only over uncertain variables
Variance based decomposition	variance_based_decomp	none	Optional	No variance_based_decomp

Table 5.24: Specification detail for nondeterministic sampling method

5.13.2 Reliability methods

Reliability methods are selected using the `nond_reliability` specification and are implemented within the **NonDReliability** class. These methods compute approximate response function distribution statistics based on specified uncertain variable probability distributions. Each of the reliability methods can compute the probabilities/reliabilities corresponding to specified response levels and the response levels corresponding to specified probability/reliability levels. Moreover, specifications of `response_levels`, `probability_levels`, and `reliability_levels` may be combined within the calculations for each response function.

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 response-probability-reliability 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). The distance of the MPP from the origin in the transformed standard normal space ("u-space") defines the reliability index. The reliability can then be converted to a probability using either first- or second-order integration. The forward reliability analysis algorithm of computing 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 [Breitung, 1984] (the approach of [Hohenbichler and Rackwitz, 1988] 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). Additional details on these methods are available in [Eldred et al., 2004b] and [Eldred and Bichon, 2006].

Table 5.25 provides details of the reliability method specifications beyond those of Table 5.28.

Description	Keyword	Associated Data	Status	Default
Reliability method	nond_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

Table 5.25: Specification detail for reliability methods

5.13.3 Polynomial chaos expansion method

The polynomial chaos expansion (PCE) method is a general framework for the approximate representation of random response functions in terms of finite-dimensional series expansions in standard unit Gaussian random variables. An important distinguishing feature of the methodology is that the solution series expansions are expressed as random processes, not merely as statistics as in the case of many nondeterministic methodologies. DAKOTA currently provides access to PCE methods through the combination of the **NonDSampling** base class and the **NonDPCESampling** derived class.

The method requires either the expansion_terms or the expansion_order specification in order to specify the number of terms in the expansion or the highest order of Gaussian variable appearing in the expansion. The number of terms, P, in a complete polynomial chaos expansion of arbitrary order, p, for a response function involving n uncertain input variables is given by

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

One must be careful when using the expansion_terms specification, as the satisfaction of the above equation for some order p is not rigidly enforced. As a result, in some cases, only a subset of terms of a certain order will be included in the series while others of the same order will be omitted. This omission of terms can increase the efficacy of the methodology for some problems but have extremely deleterious effects for others. The method outputs either the first expansion_terms coefficients of the series or the coefficients of all

terms up to order `expansion_order` in the series depending on the specification. Additional specifications include the level mappings described in Nondeterministic Methods and the `seed`, `fixed_seed`, `samples`, and `sample_type` specifications described in Nondeterministic sampling method. Table 5.26 provides details of the polynomial chaos expansion specifications beyond those of Table 5.22.

Description	Keyword	Associated Data	Status	Default
Polynomial chaos expansion iterator	`nond_polynomial_chaos`	none	Required group	N/A
Expansion terms	`expansion_terms`	integer	Required	N/A
Expansion order	`expansion_order`	integer	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
Number of samples	`samples`	integer	Optional	minimum required
Sampling type	`sample_type`	`random` | `lhs`	Optional group	`lhs`

Table 5.26: Specification detail for polynomial chaos expansion method

5.13.4 Epistemic Uncertainty Methods

Epistemic uncertainty is also referred to as subjective uncertainty, reducible uncertainty, model form uncertainty, or uncertainty due to lack of knowledge. It is different from aleatory uncertainty, which refers to inherent variability, irreducible uncertainty, or uncertainty due to chance. An example of aleatory uncertainty is the distribution of height in a population. 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. In DAKOTA, epistemic uncertainty analysis is performed using Dempster-Shafer theory of evidence. In this approach, one does not assign a probability distribution to each uncertain input variable. Rather, one divides each uncertain input variable into one or more intervals. The input parameters are only known to occur within intervals: nothing more is assumed. Each interval is defined by its upper and lower bounds, and a Basic Probability Assignment (BPA) associated with that interval. The BPA represents a probability of that uncertain variable being located within that interval. The intervals and BPAs are used to construct uncertainty measures on the outputs called "belief" and "plausibility." Belief represents the smallest possible probability that is consistent with the evidence, while plausibility represents the largest possible probability that is consistent with the evidence. For more information about the Dempster-Shafer theory of evidence, see Oberkampf and Helton, 2003 and Helton and Oberkampf, 2004.

Table 5.27 provides the specification for the `nond_evidence` method. At this point, the method specification is very simple and just involves specifying an optional seed and number of samples. Note that to calculate the plausibility and belief complementary cumulative distribution functions, which is what the `nond_evidence` method returns, 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 plausi-

bility, respectively. In terms of implementation, this method creates a large sample of 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 `nond_evidence` performs its sampling and calculations on the surrogate and not on the original model.

Description	Keyword	Associated Data	Status	Default
Nondeterministic evidence method	`nond_evidence`	none	Required group	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Number of samples	`samples`	integer	Optional	10,000

Table 5.27: Specification detail for epistemic uncertainty method

5.14 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. Two libraries are provided for performing these studies: DDACE and FSUDace. The design of experiments methods do not currently make use of any of the method independent controls.

5.14.1 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 `nond_sampling` can support this with its `all_variables` option (see Nondeterministic sampling method). DAKOTA provides access to the DDACE library through the **DDACEDesignCompExp** class.

In terms of method dependent controls, the specification structure is straightforward. First, there is a set of design of experiments algorithm selections separated by logical OR's (`grid` or `random` or `oas` or `lhs` or `oa_lhs` or `box_behnken` or `central_composite`). Second, there are optional specifications for the random seed to use in generating the sample set (`seed`), for fixing the seed (`fixed_seed`) among multiple sample sets (see Nondeterministic sampling method for discussion), for the number of samples to perform (`samples`), and for the number of symbols to use (`symbols`). The `seed` control is used to make sample sets repeatable, and

the `symbols` control is related to the number of replications in the sample set (a larger number of symbols equates to more stratification and fewer replications). The `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 5.28.

Description	Keyword	Associated Data	Status	Default
Design of experiments iterator	`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
Number of samples	`samples`	integer	Optional	minimum required
Number of symbols	`symbols`	integer	Optional	default for sampling algorithm
Quality metrics	`quality_metrics`	none	Optional	No quality_metrics
Variance based decomposition	`variance_based_decomp`	none	Optional	No variance_based_decomp

Table 5.28: Specification detail for design of experiments methods

5.14.2 FSUDace

The Florida State University Design and Analysis of Computer Experiments (FSUDace) library provides the following DACE techniques: quasi-Monte Carlo sampling (`fsu_quasi_mc`) based on the Halton sequence (`halton`) or the Hammersley sequence (`hammersley`), and Centroidal Voronoi Tessellation (`fsu_cvt`). All three methods generate sets of uniform random variables on the interval [0,1]. If the user specifies lower and upper bounds for a variable, the [0,1] samples are mapped to the [lower, upper] interval. The quasi-Monte Carlo and CVT methods are designed with the goal of low discrepancy. Discrepancy refers to the nonuniformity of the sample points within the hypercube. Discrepancy is defined as the difference between the actual number and the expected number of points one would expect in a particular set B (such as a hyper-rectangle within the unit hypercube), maximized over all such sets. Low discrepancy sequences tend to cover the unit hypercube reasonably

uniformly. Quasi-Monte Carlo methods produce low discrepancy sequences, especially if one is interested in the uniformity of projections of the point sets onto lower dimensional faces of the hypercube (usually 1-D: how well do the marginal distributions approximate a uniform?) CVT does very well volumetrically: it spaces the points fairly equally throughout the space, so that the points cover the region and are isotropically distributed with no directional bias in the point placement. There are various measures of volumetric uniformity which take into account the distances between pairs of points, regularity measures, etc. Note that CVT does not produce low-discrepancy sequences in lower dimensions, however: the lower-dimension (such as 1-D) projections of CVT can have high discrepancy.

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

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

The specification for specifying quasi-Monte Carlo (`fsu_quasi_mc`) is given below in Table 5.29. The user must specify if the sequence is (`halton`) or (`hammersley`). The user must also specify the number of samples to generate for each variable (`samples`). Then, there are three optional lists the user may specify. The first list determines where in the sequence the user wants to start. For example, for the Halton sequence in base 2, if the user specifies sequence_start = 2, the sequence would not include 0.5 and 0.25, but instead would start at 0.75. The default `sequence_start` is a vector with 0 for each variable, specifying that each sequence start with the first term. The `sequence_leap` control is similar but controls the "leaping" of terms in the sequence. The default is 1 for each variable, meaning that each term in the sequence be returned. If the user specifies a sequence_leap of 2 for a variable, the points returned would be every other term from the QMC sequence. The advantage to using a leap value greater than one is mainly for high-dimensional sets of random deviates. In this case, setting a leap value to the next prime number larger than the largest prime base can help maintain uniformity when generating sample sets for high dimensions. For more information about the efficacy of leaped Halton sequences, see [Robinson and Atcitty, 1999]. The final specification for the QMC sequences is the prime base. It is recommended that the user not specify this and use the default values. For the Halton sequence, the default bases are primes in increasing order, starting with 2, 3, 5, etc. For the Hammersley sequence, the user specifies (s-1) primes if one is generating an s-dimensional set of random variables.

The `fixed_sequence` control is similar to `fixed_seed` for other sampling methods. If `fixed_sequence` is specified, the user will get the same sequence (meaning the same set of samples) for subsequent calls of the QMC sampling method (for example, this might be used in a surrogate based optimization method or a parameter study where one wants to fix the uncertain variables). The `latinize` command takes the QMC sequence and "latinizes" it, meaning that each original sample is moved so that it falls into one strata or bin in each dimension as in Latin Hypercube sampling. The default setting is NOT to latinize a QMC sample. However, one may be interested in doing this in situations where one wants better discrepancy of the 1-dimensional projections (the marginal distributions). The `variance_based_decomp` control is also available. This control enables the calculation of sensitivity indices which indicate how important the uncertainty in each input variable is in contributing to the

output variance. More details on variance based decomposition are given in Nondeterministic sampling method.

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

Description	Keyword	Associated Data	Status	Default
FSU Quasi-Monte Carlo	`fsu_quasi_mc`	none	Required group	N/A
Sequence type	`halton` | `hammersley`	none	Required group	N/A
Number of samples	`samples`	integer	Optional	(0) for standalone sampling, (minimum required) for surrogates
Sequence starting indices	`sequence_-start`	integer list (one integer per variable)	Optional	Vector of zeroes
Sequence leaping indices	`sequence_-leap`	integer list (one integer per variable)	Optional	Vector of ones
Prime bases for sequences	`prime_base`	integer list (one integer per variable)	Optional	Vector of the first s primes for s-dimensions in Halton, First (s-1) primes for Hammersley
Fixed sequence flag	`fixed_-sequence`	none	Optional	sequence not fixed: sampling patterns are variable
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 5.29: Specification detail for FSU Quasi-Monte Carlo sequences

The FSU CVT method (`fsu_cvt`) produces a set of sample points that are (approximately) a Centroidal Voronoi Tessellation. The primary feature of such a set of points is that they have good volumetric spacing; the points

tend to arrange themselves in a pattern of cells that are roughly the same shape. To produce this set of points, an almost arbitrary set of initial points is chosen, and then an internal set of iterations is carried out. These iterations repeatedly replace the current set of sample points by an estimate of the centroids of the corresponding Voronoi subregions. [Du, Faber, and Gunzburger, 1999].

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

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

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

The specification detail for the FSU CVT method is given in Table 5.30.

5.15 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. 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.

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
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 5.30: Specification detail for FSU Centroidal Voronoi Tesselation sampling

5.15.1 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). 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.spec). First, a final_point specification, when combined with the initial values from the variables specification (see cdv_initial_point, ddv_initial_point, csv_initial_state, and dsv_initial_state in Variables Commands), uniquely defines an n-dimensional vector's direction and magnitude through its start and end points. The intervals along this vector may either be specified with a step_length or a num_steps specification. In the former case, steps of equal length (Cartesian distance) are taken from the initial values up to (but not past) the final_point. The study will terminate at the last full step which does not go beyond the final_point. In the latter num_steps case, the distance between the initial values and the final_point is broken into num_steps intervals of equal length. This study performs function evaluations at both ends, making the total number of evaluations equal to num_steps+1. The final_point specification detail is given in Table 5.31.

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. This 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 5.32.

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	Required group	N/A
Step length along vector	step_length	real	Required	N/A
Number of steps along vector	num_steps	integer	Required	N/A

Table 5.31: final_point specification detail for the vector parameter study

Description	Keyword	Associated Data	Status	Default
Vector parameter study	vector_parameter_study	none	Required group	N/A
Step vector	step_vector	list of reals	Required group	N/A
Number of steps along vector	num_steps	integer	Required	N/A

Table 5.32: step_vector specification detail for the vector parameter study

5.15.2 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 continuous variables 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. The list parameter study specification detail is given in Table 5.33

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	Required	N/A

Table 5.33: Specification detail for the list parameter study

5.15.3 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). It is selected using the `centered_parameter_study` method specification followed by `percent_delta` and `deltas_per_variable` specifications, where `percent_delta` specifies the size of the increments in percent and `deltas_per_variable` specifies the number of increments per variable in each of the plus and minus directions. The centered parameter study specification detail is given in Table 5.34.

Description	Keyword	Associated Data	Status	Default
Centered parameter study	`centered_parameter_study`	none	Required group	N/A
Interval size in percent	`percent_delta`	real	Required	N/A
Number of +/- deltas per variable	`deltas_per_variable`	integer	Required	N/A

Table 5.34: Specification detail for the centered parameter study

5.15.4 Multidimensional parameter study

DAKOTA's multidimensional parameter study computes response data sets for an n-dimensional grid of points. Each continuous variable is partitioned into equally spaced intervals between its upper and lower bounds, 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 continuous variable. Therefore, the number of entries in the partitions list must be equal to the total number of continuous variables contained in the variables specification. 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 5.35.

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 5.35: Specification detail for the multidimensional parameter study

Chapter 6

Model Commands

6.1 Model Description

The model specification in a DAKOTA input file specifies 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. In all cases, however, the model provides the logical unit for determining how a set of variables is mapped into a set of responses in support of an iterative method.

Several examples follow. The first example shows a minimal specification for a single model.

```
model,        \
single
```

This example does not provide any pointers and therefore relies on the default behavior of constructing the model with the last variables, interface, and responses specifications parsed. This is also the default model specification, for the case where no model specifications are provided by the user.

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

```
model,        \
id_model = 'M1'\
variables_pointer = 'V1'\
responses_pointer = 'R1'\
surrogate global \
  quadratic polynomial \
  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. \
    secondary_response_mapping = 0. 0. 0. 1. 3. 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).

6.2 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.spec, 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.

6.3 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, interface, and responses specifications. This top-down flow specifies all of the object interrelationships.

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

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

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

Table 6.1: Specification detail for the model independent controls: identifiers, model types, and pointers

6.4 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 6.2: Specification detail for single models

6.5 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 6.3: 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 strategy (see **SurrBasedOptStrategy** and Surrogate-based Optimization (SBO) Commands), 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 in any way to optimization techniques, and in fact, the uncertainty quantification methods and optimization under uncertainty strategy are other primary users.

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

6.5.1 Global approximations

The global surrogate model specification requires the specification of one of the following approximation methods: `neural_network`, `polynomial`, `mars`, `hermite`, `gaussian_process`, or `kriging`. These specifications invoke a layered perceptron artificial neural network approximation, a polynomial regression approximation, a multivariate adaptive regression spline approximation, a hermite polynomial approximation, a gaussian process approximation, or a kriging interpolation approximation, respectively. In the polynomial case, the order of the polynomial (linear, quadratic, or cubic) must be specified, and in the kriging case, a vector of correlations can be optionally specified in order to bypass the internal kriging calculations of correlation coefficients. Note that the gaussian process approximation is new, and currently always invokes an internal optimization procedure to determine the correlation coefficients.

For each of the global surrogates, `dace_method_pointer`, `reuse_samples`, `correction`, and `use_-`

gradients can be optionally specified. The dace_method_pointer specification points to a design of experiments iterator which can be 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 samples_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. The combination of new build data from dace_method_pointer and old build data from reuse_samples must be sufficient for building the global approximation. If not enough data is available, the system will abort with an error message. 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.

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].

Finally, the use_gradients flag specifies a future capability for the use of gradient data in the global approximation builds. This capability is currently supported in **SurrBasedOptStrategy**, **SurrogateDataPoint**, and **Approximation::build()**, but is not yet supported in any global approximation derived class redefinitions of **Approximation::find_coefficients()**. Tables 6.4 and 6.5 summarizes the global approximation specifications.

6.5.2 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 6.6 summarizes the multipoint approximation specifications.

Description	Keyword	Associated Data	Status	Default
Global approximations	global	none	Required group (1 of 4 selections)	N/A
Artificial neural network	neural_network	none	Required (1 of 6 selections)	N/A
Polynomial	polynomial	linear | quadratic | cubic	Required group (1 of 6 selections)	N/A
Multivariate adaptive regression splines	mars	none	Required (1 of 6 selections)	N/A
Hermite polynomial	hermite	none	Required (1 of 6 selections)	N/A
Gaussian process	gaussian_process	none	Required (1 of 6 selections)	N/A
Kriging interpolation	kriging	none	Required group (1 of 6 selections)	N/A
Kriging correlations	correlations	list of reals	Optional	internally computed correlations

Table 6.4: Specification detail for global approximations: global approximation type

Description	Keyword	Associated Data	Status	Default
Design of experiments method pointer	dace_method_pointer	string	Optional	no design of experiments data
Sample reuse in global approximation builds	reuse_samples	all | region | samples_file	Optional group	no sample reuse
Surrogate correction approach	correction	additive or multiplicative or combined, zeroth_order or first_order or second_order	Optional group	no surrogate correction
Use of gradient data in global approximation builds	use_gradients	none	Optional	gradient data not used in global approximation builds

Table 6.5: 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 6.6: Specification detail for multipoint approximations

6.5.3 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 6.7 summarizes the local approximation interface specifications.

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

Table 6.7: Specification detail for local approximations

6.5.4 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. Table 6.8 summarizes the hierarchical approximation specifications.

Description	Keyword	Associated Data	Status	Default
Hierarchical approximation	`hierarchical`	none	Required group (1 of 4 selections)	N/A
Pointer to the low fidelity model specification	`low_fidelity_model_pointer`	string	Required	N/A
Pointer to the high fidelity model specification	`high_fidelity_model_pointer`	string	Required	N/A
Surrogate correction approach	`correction`	`additive` or `multiplicative` or `combined`, `zeroth_order` or `first_order` or `second_order`	Required group	N/A

Table 6.8: Specification detail for hierarchical approximations

6.6 Nested Model Controls

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 6.9 provides the specification detail for nested model pointers.

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 top-level variables into the sub-model variables, and in the latter case, the `primary_response_mapping` and `secondary_response_mapping` specifications are

Description	Keyword	Associated Data	Status	Default
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 6.9: Specification detail for nested models

used to map from the sub-model responses back to the top-level responses. For the variable mappings, the primary and secondary specifications provide lists of strings which are used to target active sub-model variables and their distribution parameters, respectively. The primary strings are matched to variable labels such as 'cdv_1' (either user-supplied or default labels), and the secondary strings are matched to distribution parameters such as 'mean' or 'std_deviation' (the singular form of the uncertain distribution parameter keywords, lacking the prepended distribution type identifier). Both specifications are optional, which is designed to support 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-model distribution parameter (the secondary mapping) for the identified active 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 active sub-model variable value (the primary mapping).
3. If a primary mapping is not specified, then an active top-level variable value will be added as an inactive sub-model variable, augmenting the active sub-model variables (note: if a secondary mapping is specified in this case, it will be ignored).

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 number of active top-level variables).

For the response mappings, the primary and secondary specifications provide real-valued multipliers to be applied to sub-iterator response results. The sub-iterator response results are defined as follows for different sub-iterator types:

- optimization: the final objective function(s) and nonlinear constraints
- nonlinear least squares: the final least squares terms and nonlinear constraints
- uncertainty quantification: for each response function, a mean statistic, a standard deviation statistic, and all probability/reliability/response level results for any user-specified response_levels, probability_levels, and/or reliability_levels, in that order.
- parameter studies and design of experiments: for optimization and least squares response data sets, the best point found (lowest constraint violation if infeasible, lowest composite objective function if feasible). For

generic response data sets, a best point metric is not defined, so sub-iterator response results are not defined in this case.

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

An example of variable and response mappings is provided below:

```
primary_variable_mapping   = ''  ''  'X'     'Y'\
secondary_variable_mapping = ''  ''  'mean'  'mean'\
primary_response_mapping   = 1. 0. 0. 0. 0. 0. 0. 0. \
secondary_response_mapping = 0. 0. 0. 1. 3. 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 augment the active sub-model variables as inactive sub-model variables (option 3 above) and the latter two of which are inserted into the mean distribution parameters of active sub-model variables `'X'` and `'Y'` (option 1 above). The response mappings correspond to 9 sub-iterator response functions (e.g., a set of UQ final statistics for 3 response functions, each with a mean, a standard deviation, and a reliability level). The primary response mapping maps the first sub-iterator response function (mean) into a single objective function, least squares term, or generic response function (as dictated by the top-level response specification), and the secondary response mapping maps the fourth sub-iterator response function plus 3 times the fifth sub-iterator response function (mean plus 3 standard deviations) into one top-level nonlinear constraint and the seventh sub-iterator response function plus 3 times the eighth sub-iterator response function (mean plus 3 standard deviations) into another top-level nonlinear constraint (these top-level nonlinear constraints may be inequality or equality, as dictated by the top-level response specification).

Table 6.10 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	augmentation of sub-model variables (no insertion)
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 6.10: Specification detail for the model independent controls: nested model mappings

Chapter 7

Variables Commands

7.1 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, a lower bound, an upper bound, and a descriptive tag.

Uncertain variables are continuous variables which are characterized by probability distributions. The distribution type can be normal, lognormal, uniform, loguniform, triangular, beta, gamma, gumbel, frechet, weibull, or histogram. In addition to the uncertain variables defined by probability distributions, we have an uncertain variable type called interval, where the uncertainty in a variable is described by one or more interval values in which the variable may lie. The interval uncertain variable type is used in epistemic uncertainty calculations, specifically Dempster-Shafer theory of evidence.

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 and interval variables from the extreme values within the bin/point/interval specifications. When used with design of experiments and multidimensional parameter studies, distribution bounds are also inferred for gamma, gumbel, frechet, and weibull ([0, $\mu + 3\sigma$] for gamma, frechet, and weibull, [$\mu - 3\sigma$, $\mu + 3\sigma$] for gumbel). In addition to tags and bounds specifications, normal variables include mean and standard deviation specifications, lognormal variables include mean and either standard deviation or error factor specifications, triangular variables include mode specifications, beta, gamma, gumbel, frechet, and weibull variables include alpha and beta specifications, histogram variables include bin pairs and point pairs 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, lower and upper bounds, and descriptors. 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.

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

```
variables,\
continuous_design = 2 \
  cdv_initial_point    0.9    1.1 \
  cdv_upper_bounds     5.8    2.9 \
  cdv_lower_bounds     0.5   -2.9 \
  cdv_descriptors   'radius' 'location'
```

In the next example, defaults are employed. In this case, `cdv_initial_point` will default to a vector of `0`. values, `cdv_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), `cdv_lower_bounds` will default to a vector of `-DBL_MAX` values, and `cdv_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             \
          nuv_means            =  1.0           \
          nuv_std_deviations   =  1.0           \
          nuv_descriptors      =  'TF1n'        \
        lognormal_uncertain    =  1             \
          lnuv_means           =  2.0           \
          lnuv_std_deviations  =  0.5           \
          lnuv_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                      \
          csv_initial_state       4.0             \
          csv_lower_bounds        0.0             \
          csv_upper_bounds        8.0             \
          csv_descriptors        'CS1'            \
        discrete_state = 1                        \
          dsv_initial_state       104             \
          dsv_lower_bounds        100             \
          dsv_upper_bounds        110             \
          dsv_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 \
```

```
  cdv_initial_point    0.9    1.1 \
  cdv_upper_bounds     5.8    2.9        \
  cdv_lower_bounds     0.5   -2.9        \
  cdv_descriptors   'radius' 'location'\
discrete_design = 1 \
  ddv_initial_point    2 \
  ddv_upper_bounds     1        \
  ddv_lower_bounds     3        \
  ddv_descriptors   'material'\
normal_uncertain = 2 \
  nuv_means             =  248.89, 593.33 \
  nuv_std_deviations    =   12.4,   29.7 \
  nuv_descriptors       =  'TF1n'   'TF2n'\
uniform_uncertain = 2 \
  uuv_lower_bounds =  199.3,  474.63 \
  uuv_upper_bounds =  298.5,  712. \
  uuv_descriptors       =  'TF1u'   'TF2u'\
continuous_state = 2 \
  csv_initial_state = 1.e-4  1.e-6 \
  csv_descriptors = 'EPSIT1' 'EPSIT2'\
discrete_state = 1 \
  dsv_initial_state = 100 \
  dsv_descriptors = 'load_case'
```

Refer to the DAKOTA Users Manual [Eldred et al., 2006] for discussion on how different iterators view these mixed variable sets.

7.2 Variables Specification

The variables specification has the following structure:

```
variables,  \
<set identifier>\
<continuous design variables specification>\
<discrete design variables specification>\
<normal uncertain variables specification>\
<lognormal uncertain variables specification>\
<uniform uncertain variables specification>\
<loguniform uncertain variables specification>\
<triangular uncertain variables specification>\
<beta uncertain variables specification>\
<gamma uncertain variables specification>\
<gumbel uncertain variables specification>\
<frechet uncertain variables specification>\
<weibull uncertain variables specification>\
<histogram uncertain variables specification>\
<interval uncertain variables specification>\
<uncertain correlation specification>       \
<continuous state variables specification>\
<discrete state variables specification>
```

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

The optional status of the different variable type specifications allows the user to specify only those variables which are present (rather than explicitly specifying that the number of a particular type of variables = 0). However, at least one type of variables must have nonzero size or an input error message will result. The following sections describe each of these specification components in additional detail.

7.3 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 7.1: Specification detail for set identifier

7.4 Design Variables

Within the optional continuous design variables specification group, the number of continuous design variables is a required specification and the initial point, lower bounds, upper bounds, scaling factors, and variable names are optional specifications. Likewise, within the optional discrete design variables specification group, the number of discrete design variables is a required specification and the initial guess, lower bounds, upper bounds, and variable names are optional specifications. Table 7.2 summarizes the details of the continuous design variable specification and Table 7.3 summarizes the details of the discrete design variable specification.

The `cdv_initial_point` and `ddv_initial_point` specifications provide the point in design space from which an iterator is started for the continuous and discrete design variables, respectively. The `cdv_lower_bounds`, `ddv_lower_bounds`, `cdv_upper_bounds` and `ddv_upper_bounds` restrict the size of the feasible design space and are frequently used to prevent nonphysical designs. The `cdv_scales` specification provides nonnegative real values for scaling of 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 `cdv_scales` may be a user-specified characteristic value, `0.` for automatic scaling, or `1.` for no scaling. If a single real value is specified it will apply to all components of the continuous design variables vector. The

Description	Keyword	Associated Data	Status	Default
Continuous design variables	continuous_design	integer	Optional group	no continuous design variables
Initial point	cdv_initial_point	list of reals	Optional	vector values = 0.
Lower bounds	cdv_lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Upper bounds	cdv_upper_bounds	list of reals	Optional	vector values = +DBL_MAX
Scales	cdv_scales	list of reals	Optional	no scaling (vector values = 1.)
Descriptors	cdv_descriptors	list of strings	Optional	vector of 'cdv_i' where i = 1,2,3...

Table 7.2: Specification detail for continuous design variables

Description	Keyword	Associated Data	Status	Default
Discrete design variables	discrete_design	integer	Optional group	no discrete design variables
Initial point	ddv_initial_point	list of integers	Optional	vector values = 0
Lower bounds	ddv_lower_bounds	list of integers	Optional	vector values = INT_MIN
Upper bounds	ddv_upper_bounds	list of integers	Optional	vector values = INT_MAX
Descriptors	ddv_descriptors	list of strings	Optional	vector of 'ddv_i' where i = 1,2,3,...

Table 7.3: Specification detail for discrete design variables

cdv_descriptors and ddv_descriptors specifications supply strings which will be replicated through the DAKOTA output to help identify the numerical values for these parameters. Default values for optional specifications are zeros for initial values, 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), and numbered strings for descriptors. 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 design variables, 1.e+9 for discrete design 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), 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).

7.5 Uncertain Variables

Uncertain variables involve one of several supported probability distribution specifications, including normal, lognormal, uniform, loguniform, triangular, beta, gamma, gumbel, frechet, weibull, or histogram distributions. Each of these specifications is an optional group specification. There also is an uncertain variable type called an interval variable. This is not a probability distribution, but is used in specifying the inputs necessary for an epistemic uncertainty analysis using Dempster-Shafer theory of evidence.

The inclusion of lower and upper distribution bounds for all uncertain variable types (either explicitly defined, implicitly defined, or inferred; see Variables Description) allows the use of these variables with methods that rely on a bounded region to define a set of function evaluations (i.e., design of experiments and some parameter study methods). In addition, distribution bounds can be used to truncate the tails of distributions for normal and lognormal uncertain variables (see "bounded normal", "bounded lognormal", and "bounded lognormal-n" distribution types in [Wyss and Jorgensen, 1998]). Default upper and lower bounds are positive and negative machine limits (+/- DBL_MAX from the float.h system header file), respectively, for non-logarithmic distributions and positive machine limits and zeros, respectively, for logarithmic distributions. The uncertain variable descriptors provide strings which will be replicated through the DAKOTA output to help identify the numerical values for these parameters. Default values for descriptors are numbered strings. Tables 7.4 through 7.14 summarize the details of the uncertain variable specifications.

7.5.1 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. This can result in

the mean and the standard deviation of the sample data being different from the mean and standard deviation of the underlying distribution. For example, if you are sampling from a normal distribution with a mean of 5 and a standard deviation of 3, but you specify bounds of 1 and 7, the resulting mean of the samples will be around 4.3 and the resulting standard deviation will be around 1.6. This is because you have bounded the original distribution significantly, and asymetrically, since 7 is closer to the original mean than 1.

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

Table 7.4: Specification detail for normal uncertain variables

7.5.2 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.

For the lognormal variables, DAKOTA's uncertainty quantification methods standardize on the use of statistics of the actual lognormal distribution, as opposed to statistics of the underlying normal distribution. This approach diverges from that of [Wyss and Jorgensen, 1998], which assumes that a specification of means and standard deviations provides parameters of the underlying normal distribution, whereas a specification of means and error factors provides statistics of the actual lognormal distribution. By binding the mean, standard deviation, and error factor parameters consistently to the actual lognormal distribution, inputs are more intuitive and require fewer conversions in most user applications. The conversion equations from lognormal mean μ_{LN} and either lognormal error factor ϵ_{LN} or lognormal standard deviation σ_{LN} to the mean μ_N and standard deviation σ_N of the underlying normal distribution are as follows:

$$\sigma_N = \frac{ln(\epsilon_{LN})}{1.645}$$

$$\sigma_N^2 = ln(\frac{\sigma_{LN}^2}{\mu_{LN}^2} + 1.)$$

$$\mu_N = ln(\mu_{LN}) - \frac{\sigma_N^2}{2}$$

Conversions from μ_N and σ_N back to μ_{LN} and ϵ_{LN} or σ_{LN} are as follows:

$$\mu_{LN} = e^{\mu_N + \frac{\sigma_N^2}{2}}$$
$$\sigma_{LN}^2 = e^{2\mu_N + \sigma_N^2}(e^{\sigma_N^2} - 1.)$$
$$\epsilon_{LN} = e^{1.645\sigma_N}$$

The density function for the lognormal distribution is:

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

Description	Keyword	Associated Data	Status	Default
lognormal uncertain variables	`lognormal_uncertain`	integer	Optional group	no lognormal uncertain variables
lognormal uncertain means	`lnuv_means`	list of reals	Required	N/A
lognormal uncertain standard deviations	`lnuv_std_deviations`	list of reals	Required (1 of 2 selections)	N/A
lognormal uncertain error factors	`lnuv_error_factors`	list of reals	Required (1 of 2 selections)	N/A
Distribution lower bounds	`lnuv_lower_bounds`	list of reals	Optional	vector values = `0.`
Distribution upper bounds	`lnuv_upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`lnuv_descriptors`	list of strings	Optional	vector of `'lnuv_i'` where `i = 1,2,3,...`

Table 7.5: Specification detail for lognormal uncertain variables

7.5.3 Uniform Distribution

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

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

where U_U and L_U are the upper and lower bounds of the uniform distribution, respectively. The mean of the uniform distribution is $\frac{U_U + L_U}{2}$ and the variance is $\frac{(U_U - L_U)^2}{12}$.

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

Table 7.6: Specification detail for uniform uncertain variables

7.5.4 Loguniform Distribution

If the logarithm of an uncertain variable X has a uniform distribution, that is $\log X \sim U(U_U, L_U)$, 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	`luuv_lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`luuv_upper_bounds`	list of reals	Required	N/A
Descriptors	`luuv_descriptors`	list of strings	Optional	vector of `'luuv_i'` where `i = 1,2,3,...`

Table 7.7: Specification detail for loguniform uncertain variables

7.5.5 Triangular Distribution

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

The density function for the triangular distribution is:

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

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

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

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

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

Table 7.8: Specification detail for triangular uncertain variables

7.5.6 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:

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

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

Description	Keyword	Associated Data	Status	Default
beta uncertain variables	beta_uncertain	integer	Optional group	no beta uncertain variables
beta uncertain alphas	buv_alphas	list of reals	Required	N/A
beta uncertain betas	buv_betas	list of reals	Required	N/A
Distribution lower bounds	buv_lower_bounds	list of reals	Required	N/A
Distribution upper bounds	buv_upper_bounds	list of reals	Required	N/A
Descriptors	buv_descriptors	list of strings	Optional	vector of 'buv_i' where i = 1,2,3,...

Table 7.9: Specification detail for beta uncertain variables

7.5.7 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.

The density function for the gamma distribution is given by:

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

where $\mu_{GA} = \alpha\beta$ and $\sigma^2_{GA} = \alpha\beta^2$. Note that the gamma distribution with parameters α = 1 and β is the same as an exponential with parameter β.

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

Table 7.10: Specification detail for gamma uncertain variables

7.5.8 Gumbel Distribution

Within the gumbel optional uncertain group specification, the number of gumbel uncertain variables, and the alpha and beta parameters are required specifications. The Gumbel distribution is also referred to as the Type I Largest Extreme Value distribution. The distribution of maxima in sample sets from a population with a normal distribution will asymptotically converge to this distribution. It is commonly used to model demand variables such as wind loads and flood levels.

The density function for the Gumbel distribution is given by:

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

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

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

Table 7.11: Specification detail for gumbel uncertain variables

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

7.5.10 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

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

Table 7.12: Specification detail for frechet uncertain variables

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	wuv_alphas	list of reals	Required	N/A
weibull uncertain betas	wuv_betas	list of reals	Required	N/A
Descriptors	wuv_descriptors	list of strings	Optional	vector of 'wuv_i' where i = 1,2,3,...

Table 7.13: Specification detail for weibull uncertain variables

7.5.11 Histogram

Within the histogram uncertain optional group specification, the number of histogram uncertain variables is a required specification, the bin pairs and point pairs are optional group specifications, and the variable descriptors is an optional specification. When using a histogram variable, one must define at least one set of bin pairs or point pairs. Also note that the total number of histogram variables must be equal to the number of variables defined by bin pairs and point pairs.

For the histogram uncertain variable specification, the bin pairs and point pairs specifications provide sets of (x,y) pairs for each histogram variable. The distinction between the two types is that the former specifies counts for bins of non-zero width, whereas the latter specifies counts for individual point values, which can be thought of as

bins with zero width. In the terminology of LHS [Wyss and Jorgensen, 1998], the former is a "continuous linear histogram" and the latter is a "discrete histogram" (although the points are real-valued, the number of possible values is finite). To fully specify a bin-based histogram with `n` bins where the bins can be of unequal width, `n+1` (x,y) pairs must be specified with the following features:

- `x` is the parameter value for the left boundary of a histogram bin and `y` is the corresponding count for that bin.
- the final pair specifies the right end of the last bin and must have a `y` value of zero.
- the `x` values must be strictly increasing.
- all `y` values must be positive, except for the last which must be zero.
- a minimum of two (x,y) pairs must be specified for each bin-based histogram.

Similarly, to specify a point-based histogram with `n` points, `n` (x,y) pairs must be specified with the following features:

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

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

```
histogram_uncertain = 3 \
  huv_num_bin_pairs     = 3 4 \
  huv_bin_pairs         = 5 17 8 21 10 0 .1 12 .2 24 .3 12 .4 0 \
  huv_num_point_pairs   = 2 \
  huv_point_pairs       = 3 1 4 1
```

`huv_num_bin_pairs` associates the first 3 pairs from `huv_bin_pairs` ((5,17),(8,21),(10,0)) with one bin-based histogram variable and the following set of 4 pairs ((.1,12),(.2,24),(.3,12),(.4,0)) with a second bin-based histogram variable. Likewise, `huv_num_point_pairs` associates both of the (x,y) pairs from `huv_point_pairs` ((3,1),(4,1)) with a single point-based histogram variable. Finally, the total number of bin-based variables and point-based variables must add to the total number of histogram variables specified (3 in this example).

7.5.12 Interval Uncertain Variable

The interval uncertain variable is NOT a probability distribution. Although it may seem similar to a histogram, the interpretation of this uncertain variable is different. It is used in epistemic uncertainty analysis, where one is trying to model uncertainty due to lack of knowledge. In DAKOTA, epistemic uncertainty analysis is performed using Dempster-Shafer theory of evidence. In this approach, one does not assign a probability distribution to each uncertain input variable. Rather, one divides each uncertain input variable into one or more intervals. The

Description	Keyword	Associated Data	Status	Default
histogram uncertain variables	histogram_uncertain	integer	Optional group	no histogram uncertain variables
number of (x,y) pairs for each bin-based histogram variable	huv_num_bin_pairs	list of integers	Optional group	no bin-based histogram uncertain variables
(x,y) pairs for all bin-based histogram variables	huv_bin_pairs	list of reals	Optional group	no bin-based histogram uncertain variables
number of (x,y) pairs for each point-based histogram variable	huv_num_point_pairs	list of integers	Optional group	no point-based histogram uncertain variables
(x,y) pairs for all point-based histogram variables	huv_point_pairs	list of reals	Optional group	no point-based histogram uncertain variables
Descriptors	huv_descriptors	list of strings	Optional	vector of 'huv_i' where i = 1,2,3,...

Table 7.14: Specification detail for histogram uncertain variables

input parameters are only known to occur within intervals: nothing more is assumed. Each interval is defined by its upper and lower bounds, and a Basic Probability Assignment (BPA) associated with that interval. The BPA represents a probability of that uncertain variable being located within that interval. The intervals and BPAs are used to construct uncertainty measures on the outputs called "belief" and "plausibility." Belief represents the smallest possible probability that is consistent with the evidence, while plausibility represents the largest possible probability that is consistent with the evidence. For more information about the Dempster-Shafer approach, see the nondeterministic evidence method, nond_evidence, in the Methods section of this Reference manual. As an example, in the following specification:

```
interval_uncertain = 2 \
  iuv_num_intervals    = 3 2 \
  iuv_interval_probs   = 0.2 0.5 0.3 0.4 0.6 \
  iuv_interval_bounds  = 2 2.5 4 5 4.5 6 1.0 5.0 3.0 5.0 \
```

there are 2 interval uncertain variables. The first one is defined by three intervals, and the second by two intervals. The three intervals for the first variable have basic probability assignments of 0.2, 0.5, and 0.3, respectively, while the basic probability assignments for the two intervals for the second variable are 0.4 and 0.6. The basic probability assignments for each interval variable must sum to one. The interval bounds for the first variable are [2, 2.5], [4, 5], and [4.5, 6]. Note that the lower bound must always come first in the bound pair. Also note that the intervals can be overlapping. The interval bounds for the second variable are [1.0, 5.0] and [3.0, 5.0]. Table 7.15 summarizes the specification details for the interval_uncertain variable.

Description	Keyword	Associated Data	Status	Default
interval uncertain variables	`interval_uncertain`	integer	Optional group	no interval uncertain variables
number of intervals defined for each interval variable	`iuv_num_intervals`	list of integers	Required group	None
basic probability assignments per interval	`iuv_interval_probs`	list of reals	Required group. Note that the probabilities per variable must sum to one.	None
bounds per interval	`iuv_interval_bounds`	list of reals	Required group. Specify bounds as (lower, upper) per interval, per variable	None
Descriptors	`iuv_descriptors`	list of strings	Optional	vector of `'iuv_i'` where `i = 1,2,3,...`

Table 7.15: Specification detail for interval uncertain variables

7.5.13 Correlations

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 nondeterministic method in use. When the method is a nondeterministic sampling method (i.e., `nond_sampling`), then the correlation matrix specifies *rank correlations* [Iman and Conover, 1982]. When the method is instead a reliability (i.e., `nond_reliability`) or polynomial chaos (i.e., `nond_polynomial_chaos`) 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 uncertain variables (all normal, lognormal, uniform, loguniform, weibull, and histogram specifications, in that order). Table 7.16 summarizes the specification details:

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

Table 7.16: Specification detail for uncertain correlations

7.6 State Variables

Within the optional continuous state variables specification group, the number of continuous state variables is a required specification and the initial states, lower bounds, upper bounds, and variable descriptors are optional specifications. Likewise, within the optional discrete state variables specification group, the number of discrete state variables is a required specification and the initial states, lower bounds, upper bounds, and variable descriptors are optional specifications. These variables provide a convenient mechanism for managing additional model parameterizations such as mesh density, simulation convergence tolerances, and time step controls. Table 7.17 summarizes the details of the continuous state variable specification and Table 7.18 summarizes the details of the discrete state variable specification.

Description	Keyword	Associated Data	Status	Default
Continuous state variables	`continuous_state`	integer	Optional group	No continuous state variables
Initial states	`csv_initial_state`	list of reals	Optional	vector values = `0`.
Lower bounds	`csv_lower_bounds`	list of reals	Optional	vector values = `-DBL_MAX`
Upper bounds	`csv_upper_bounds`	list of reals	Optional	vector values = `+DBL_MAX`
Descriptors	`csv_descriptors`	list of strings	Optional	vector of `'csv_i'` where `i = 1,2,3,...`

Table 7.17: Specification detail for continuous state variables

Description	Keyword	Associated Data	Status	Default
Discrete state variables	`discrete_state`	integer	Optional group	No discrete state variables
Initial states	`dsv_initial_state`	list of integers	Optional	vector values = `0`
Lower bounds	`dsv_lower_bounds`	list of integers	Optional	vector values = `INT_MIN`
Upper bounds	`dsv_upper_bounds`	list of integers	Optional	vector values = `INT_MAX`
Descriptors	`dsv_descriptors`	list of strings	Optional	vector of `'dsv_i'` where `i = 1,2,3,...`

Table 7.18: Specification detail for discrete state variables

The `csv_initial_state` and `dsv_initial_state` specifications define the initial values for the continuous and discrete state variables which will be passed through to the simulator (e.g., in order to define parame-

terized modeling controls). The csv_lower_bounds, csv_upper_bounds, dsv_lower_bounds, and dsv_upper_bounds restrict the size of the state parameter space and are frequently used to define a region for design of experiments or parameter study investigations. The csv_descriptors and dsv_descriptors specifications provide strings which will be replicated through the DAKOTA output to help identify the numerical values for these parameters. Default values for optional specifications are zeros for initial states, 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), and numbered strings for descriptors.

Chapter 8

Interface Commands

8.1 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 an system call 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 System call interface for more information on the use of these options.

```
interface,                                              \
system \
  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 **DirectFnApplicInterface** 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'\
system \
  analysis_driver = 'text_book'\
  parameters_file = 'tb.in'\
  results_file    = 'tb.out'\
  file_tag \
asynchronous
```

8.2 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.spec, 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.

8.3 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 8.1: 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
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 8.2: 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 for additional information.

8.4 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 8.3: Specification detail for algebraic mappings

8.5 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 8.4: 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 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 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.

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.

8.5.1 System call interface

For system call interfaces, the `parameters_file`, `results_file`, `analysis_usage`, `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). Special analysis command syntax can be entered as a string with the `analysis_usage` specification. This special syntax replaces the normal system call combination of the specified `analysis_drivers` with command line arguments; however, it does not affect the `input_filter` and `output_filter` syntax (if filters are present). Note that if there are multiple analysis drivers, then `analysis_usage` must include the syntax for all analyses in a single string (typically separated by semi-colons). The default is no special syntax, such that the `analysis_drivers` will be used in the standard way as described in the Interfaces chapter of the Users Manual. The format of data in the parameters files can be modified for direct usage with the APREPRO pre-processing tool [Sjaardema, 1992] using the `aprepro` specification (NOTE: the DPrePro pre-processing utility does not require this special formatting). File tagging (appending parameters and results files with the function evaluation number) and file saving (leaving parameters and results files in existence after their use is complete) are controlled with the `file_tag` and `file_save` flags. If these specifications are omitted, the default is no file tagging (no appended function evaluation number) and no file saving (files will be removed after a function evaluation). File tagging is most useful when multiple function evaluations are running simultaneously using files in a shared disk space, and file saving is most useful when debugging the data communication between DAKOTA and the simulation. The additional specifications for system call interfaces are summarized in Table 8.5.

Description	Keyword	Associated Data	Status	Default
System call interface	system	none	Required group (1 of 4 selections)	N/A
Parameters file name	parameters_file	string	Optional	Unix temp files
Results file name	results_file	string	Optional	Unix temp files
Special analysis usage syntax	analysis_usage	string	Optional	standard analysis usage
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

Table 8.5: Additional specifications for system call interfaces

8.5.2 Fork interface

For fork interfaces, the parameters_file, results_file, aprepro, file_tag, and file_save are additional settings within the group specification and have identical meanings to those for the system call interface. The only difference in specifications is that fork interfaces do not support an analysis_usage specification due to limitations in the execvp() function used when forking a process. The additional specifications for fork interfaces are summarized in Table 8.6.

Description	Keyword	Associated Data	Status	Default
Fork interface	fork	none	Required group (1 of 4 selections)	N/A
Parameters file name	parameters_file	string	Optional	Unix temp files
Results file name	results_file	string	Optional	Unix temp files
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

Table 8.6: Additional specifications for fork interfaces

8.5.3 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) 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 8.7.

Description	Keyword	Associated Data	Status	Default
Direct function interface	direct	none	Required group (1 of 4 selections)	N/A
Number of processors per analysis	processors_per_analysis	integer	Optional	no override of auto configure

Table 8.7: Additional specifications for direct function interfaces

DAKOTA supports direct interfaces to a few select simulation codes. One example is ModelCenter, a commercial simulation management framework from Phoenix Integration. To utilize this interface, a user must first define the simulation specifics within a ModelCenter session and then save these definitions to a ModelCenter configuration file. The analysis_components specification provides the means to communicate this configuration file to DAKOTA's ModelCenter interface. A similar direct interface to The Mathworks' (http://www.mathworks.com/) Matlab (specified by analysis_driver = 'matlab') enables a user to employ the analysis_components specification to point to a Matlab m-file containing a function that performs the simulation. This capability is in development and disabled by default in DAKOTA binaries, but interested users may contact the DAKOTA developers for assistance building and using DAKOTA with Matlab simulation support.

Other direct interfaces to simulation codes include Sandia's SALINAS structural dynamics code, Sandia's SIERRA multiphysics framework, and Sandia's SAGE computational fluid dynamics code, which are available within Sandia and supported to varying degrees. In addition to interfaces with simulation codes, a common usage of the direct interface is for invoking internal test functions which are available for performing parameter to response mappings as inexpensively as possible. These problems are compiled directly into the DAKOTA executable as part of the direct function interface class and are used for algorithm testing. Refer to **DirectFnApplicInterface** for currently available testers.

8.5.4 Grid interface

For grid interfaces, no additional specifications are used at this time.

This capability has been used for interfaces with IDEA and JAVASpaces in the past and is currently a placeholder for future work with Condor and/or Globus services. It is not currently operational. The grid interface specification is summarized in Table 8.8.

Description	Keyword	Associated Data	Status	Default
Grid interface	grid	none	Required group (1 of 4 selections)	N/A

Table 8.8: Additional specifications for grid interfaces

Chapter 9

Responses Commands

9.1 Responses Description

The responses specification in a DAKOTA input file specifies the data set that can be recovered from the interface after the completion of a "function evaluation." Here, the term function evaluation is used somewhat 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 made up 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 (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. Lastly, parameter study and design of experiments iterators may be used with any of the response data set types. Within the C++ implementation, the same data structures are reused for each of these cases; only the interpretation of the data varies from iterator branch to iterator branch.

Gradient availability may be described by `no_gradients`, `numerical_gradients`, `analytic_gradients`, or `mixed_gradients`. The `no_gradients` selection means 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 using either the native or one of the vendor finite differencing routines. 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 may be described by `no_hessians`, `numerical_hessians`, `quasi_hessians`, `analytic_hessians`, or `mixed_hessians`. As for the gradient specification, 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 is needed and will be estimated with finite differences using either first-order differences of gradients (for analytic gradients) or second-order differences of function values (for non-analytic gradients). The `quasi_hessians` specification means that Hessian information is needed and will be accumulated over time using quasi-Newton 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.

The responses specification provides a description of the *total* data set that is available for use by the iterator during the course of its iteration. This should be distinguished from the data *subset* described in an active set vector (see DAKOTA File Data Formats in the Users Manual) which describes the particular subset of the response data needed for an individual function evaluation. In other words, the responses specification is a broad description of the data to be used during a study whereas the active set vector describes the particular subset of the available data that is 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,        \
num_objective_functions = 1         \
num_nonlinear_inequality_constraints = 2 \
analytic_gradients         \
no_hessians
```

The next example shows a typical specification for a least squares 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,        \
num_least_squares_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,        \
num_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.

9.2 Responses Specification

The responses specification has the following structure:

```
responses,          \
<set identifier>\
<response descriptors>\
<function specification>\
<gradient specification>\
<Hessian specification>
```

Referring to dakota.input.spec, it is evident from the enclosing brackets that the set identifier and response descriptors are optional. However, the function, gradient, and Hessian specifications are all required specifications, each of which contains several possible specifications separated by logical OR's. The function specification must be one of three types:

- objective and constraint functions
- least squares terms and constraint functions
- generic response functions

The gradient specification must be one of four types:

- no gradients
- numerical gradients
- analytic gradients
- mixed gradients

And the Hessian specification must be one of five types:

- no Hessians
- numerical Hessians
- quasi Hessians
- analytic Hessians
- mixed Hessians

The following sections describe each of these specification components in additional detail.

9.3 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 9.1: Specification detail for set identifier

9.4 Response Labels

The optional response labels specification uses the keyword `response_descriptors` to input a list of strings which will be replicated through the DAKOTA output to help identify the numerical 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	`response_descriptors`	list of strings	Optional	root strings plus numeric identifiers

Table 9.2: Specification detail for response labels

9.5 Function Specification

The function specification must be one of three types: 1) a group containing objective and constraint functions, 2) a group containing 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.

9.5.1 Objective and constraint functions (optimization data set)

An optimization data set is specified using `num_objective_functions` and optionally `objective_function_scales`, `multi_objective_weights`, `num_nonlinear_inequality_constraints`, `nonlinear_inequality_lower_bounds`, `nonlinear_inequality_upper_bounds`, `nonlinear_inequality_scales`, `num_nonlinear_equality_constraints`, `nonlinear_equality_targets`, and `nonlinear_equality_scales`. The `num_objective_functions`, `num_nonlinear_inequality_constraints`, and `num_nonlinear_equality_constraints` inputs specify the number of objective functions, nonlinear inequality constraints, and nonlinear

equality constraints, respectively. The number of objective functions must be 1 or greater, and the number of inequality and equality constraints must be 0 or greater.

The `objective_function_scales` specification provides positive real values for scaling each objective function value in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `objective_function_scales` may be a user-specified characteristic value, or `1.` for no scaling. Automatic scaling is not available for objective functions. If a single real value is specified it will apply to each objective function. If the number of objective functions is greater than 1, then a `multi_objective_weights` specification provides a simple weighted-sum approach to combining multiple objectives:

$$f = \sum_{i=1}^{n} w_i f_i$$

If this is not specified, then each objective function is given equal weighting:

$$f = \sum_{i=1}^{n} \frac{f_i}{n}$$

If scaling is specified, it is applied before multi-objective weighted sums are formed.

The `nonlinear_inequality_lower_bounds` and `nonlinear_inequality_upper_bounds` specifications provide the lower and upper bounds for 2-sided nonlinear inequalities of the form

$$g_l \leq g(x) \leq g_u$$

The defaults for the inequality constraint bounds are selected so that one-sided inequalities of the form

$$g(x) \leq 0.0$$

result when there are no user constraint bounds specifications (this provides backwards compatibility with previous DAKOTA versions). In a user bounds specification, any upper bound values greater than `+bigRealBoundSize` (1.e+30, as defined in **Minimizer**) are treated as +infinity and any lower bound values less than `-bigRealBoundSize` are treated as -infinity. This feature is commonly used to drop one of the bounds in order to specify a 1-sided constraint (just as the default lower bounds drop out since `-DBL_MAX` < `-bigRealBoundSize`). The same approach is used for nonexistent linear inequality bounds as described in Method Independent Controls and for nonexistent design variable bounds as described in Design Variables.

The `nonlinear_equality_targets` specification provides the targets for nonlinear equalities of the form

$$g(x) = g_t$$

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

$$g(x) = 0.0$$

The `nonlinear_inequality_scales` and `nonlinear_equality_scales` specifications provide nonnegative real values for scaling nonlinear inequality and equality constraints, repsectively, in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in these scale vectors may be a user-specified characteristic value, `0.` for automatic scaling or `1.` for no scaling. If a single real value is specified it will apply to each component.

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.

Description	Keyword	Associated Data	Status	Default
Number of objective functions	num_objective_functions	integer	Required group	N/A
Objective function scales	objective_function_scales	list of reals	Optional	no scaling (vector values = 1.)
Multiobjective weightings	multi_objective_weights	list of reals	Optional	equal weightings
Number of nonlinear inequality constraints	num_nonlinear_inequality_constraints	integer	Optional	0
Nonlinear inequality constraint lower bounds	nonlinear_inequality_lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Nonlinear inequality constraint upper bounds	nonlinear_inequality_upper_bounds	list of reals	Optional	vector values = 0.
Nonlinear inequality constraint scales	nonlinear_inequality_scales	list of reals	Optional	no scaling (vector values = 1.)
Number of nonlinear equality constraints	num_nonlinear_equality_constraints	integer	Optional	0
Nonlinear equality constraint targets	nonlinear_equality_targets	list of reals	Optional	vector values = 0.
Nonlinear equality constraint scales	nonlinear_equality_scales	list of reals	Optional	no scaling (vector values = 1.)

Table 9.3: Specification detail for optimization data sets

9.5.2 Least squares terms and constraint functions (least squares data set)

A least squares data set is specified using `num_least_squares_terms` and optionally `least_squares_term_scales`, `num_nonlinear_inequality_constraints`, `nonlinear_inequality_lower_bounds`, `nonlinear_inequality_upper_bounds`, `nonlinear_inequality_scales`, `num_nonlinear_equality_constraints`, `nonlinear_equality_targets`, and `nonlinear_equality_scales`. Each of the least squares 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 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. It is important to realize that, 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 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 involves the values and derivatives of the residual functions, whereas the optimization case involves values and derivatives of the sum of the squares objective function. Switching between the two approaches will likely require different simulation interfaces capable of returning the different granularity of response data required. The `least_squares_term_scales` specification provides positive real values for scaling each least squares residual term in methods that support scaling, when scaling is enabled (see Method Independent Controls for details). Each entry in `least_squares_term_scales` may be a user-specified characteristic value, or `1.` for no scaling. Automatic scaling is not available for least squares terms. If a single real value is specified it will apply to each term.

Table 9.4 summarizes the least squares data set specification.

9.5.3 Response functions (generic data set)

A generic response data set is specified using `num_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 9.5 summarizes the generic response function data set specification.

Description	Keyword	Associated Data	Status	Default
Number of least squares terms	num_least_squares_terms	integer	Required	N/A
Least squares terms scales	least_squares_term_scales	list of reals	Optional	no scaling (vector values = 1.)
Number of nonlinear inequality constraints	num_nonlinear_inequality_constraints	integer	Optional	0
Nonlinear inequality constraint lower bounds	nonlinear_inequality_lower_bounds	list of reals	Optional	vector values = -DBL_MAX
Nonlinear inequality constraint upper bounds	nonlinear_inequality_upper_bounds	list of reals	Optional	vector values = 0.
Nonlinear inequality constraint scales	nonlinear_inequality_scales	list of reals	Optional	no scaling (vector values = 1.)
Number of nonlinear equality constraints	num_nonlinear_equality_constraints	integer	Optional	0
Nonlinear equality constraint targets	nonlinear_equality_targets	list of reals	Optional	vector values = 0.
Nonlinear equality constraint scales	nonlinear_equality_scales	list of reals	Optional	no scaling (vector values = 1.)

Table 9.4: Specification detail for nonlinear least squares data sets

Description	Keyword	Associated Data	Status	Default
Number of response functions	num_response_functions	integer	Required	N/A

Table 9.5: Specification detail for generic response function data sets

9.6 Gradient Specification

The gradient specification must be one of four types: 1) no gradients, 2) numerical gradients, 3) analytic gradients, or 4) mixed gradients.

9.6.1 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.

9.6.2 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). However, the `vendor` setting can be desirable in some cases since certain libraries will modify their algorithm when the finite differencing is performed internally. Since the selection of the `dakota` routine hides the use of finite differencing from the optimizers (the optimizers are configured to accept user-supplied gradients, which some algorithms assume to be of analytic accuracy), the potential exists for the `vendor` setting to trigger the use of an algorithm more optimized for the higher expense and/or lower accuracy of finite-differencing. For example, NPSOL uses gradients in its line search when in user-supplied gradient mode (since it assumes they are inexpensive), but uses a value-based line search procedure when internally finite differencing. The use of a value-based line search will often reduce total expense in serial operations. However, in parallel operations, the use of gradients in the NPSOL line search (user-supplied gradient mode) provides excellent load balancing without need to resort to speculative optimization approaches. In summary, then, the `dakota` routine is preferred for parallel optimization, and the `vendor` routine may be preferred for serial optimization in special cases.

The `interval_type` setting is used to select between `forward` and `central` differences in the numerical gradient calculations. The `dakota`, DOT `vendor`, and OPT++ `vendor` routines have both forward and central differences available, the CONMIN and NL2SOL `vendor` routines support forward differences only, and the NPSOL and NLSSOL `vendor` routines start with forward differences and automatically switch to central differences as the iteration progresses (the user has no control over this). The following forward difference expression

$$\nabla f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

and the following central difference expression

$$\nabla f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}$$

are used to estimate the i^{th} component of the gradient vector.

Lastly, fd_gradient_step_size specifies the relative finite difference step size to be used in the computations. Either a single value may be entered for use with all parameters, or a list of step sizes may be entered, one for each parameter. The latter option of a list of step sizes is only valid for use with the DAKOTA finite differencing routine. For DAKOTA, DOT, CONMIN, and OPT++, the differencing intervals are computed by multiplying the fd_gradient_step_size with the current parameter value. In this case, a minimum absolute differencing interval is needed when the current parameter value is close to zero. This prevents finite difference intervals for the parameter which are too small to distinguish differences in the response quantities being computed. DAKOTA, DOT, CONMIN, and OPT++ all use .01∗fd_gradient_step_size as their minimum absolute differencing interval. With a fd_gradient_step_size = .001, for example, DAKOTA, DOT, CONMIN, and OPT++ will use intervals of .001∗current value with a minimum interval of 1.e-5. NPSOL and NLSSOL use a different formula for their finite difference intervals: fd_gradient_step_size∗(1+|current parameter value|). This definition has the advantage of eliminating the need for a minimum absolute differencing interval since the interval no longer goes to zero as the current parameter value goes to zero. Table 9.6 summarizes the numerical gradient specification.

Description	Keyword	Associated Data	Status	Default
Numerical gradients	numerical_gradients	none	Required group	N/A
Method source	method_source	dakota | vendor	Optional group	dakota
Interval type	interval_type	forward | central	Optional group	forward
Finite difference step size	fd_gradient_step_size	list of reals	Optional	0.001

Table 9.6: Specification detail for numerical gradients

9.6.3 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) for the case of file transfer of data. The analytic_gradients keyword is a complete specification for this case.

9.6.4 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. 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 9.7 summarizes the mixed gradient specification.

Description	Keyword	Associated Data	Status	Default
Mixed gradients	mixed_gradients	none	Required group	N/A
Analytic derivatives function list	id_analytic_gradients	list of integers	Required	N/A
Numerical derivatives function list	id_numerical_gradients	list of integers	Required	N/A
Method source	method_source	dakota | vendor	Optional group	dakota
Interval type	interval_type	forward | central	Optional group	forward
Finite difference step size	fd_gradient_step_size	list of reals	Optional	0.001

Table 9.7: Specification detail for mixed gradients

9.7 Hessian Specification

Hessian availability must be specified with either no_hessians, numerical_hessians, quasi_hessians, analytic_hessians, or mixed_hessians.

9.7.1 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.

9.7.2 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 or for the functions identified by id_analytic_gradients in the case of mixed_gradients) or second-order

function value differencing (all other gradient specifications). In the former case, the following expression

$$\nabla^2 f(\mathbf{x}) \cong \frac{\nabla f(\mathbf{x} + h\mathbf{e}_i) - \nabla f(\mathbf{x})}{h}$$

estimates the i^{th} Hessian column, and in the latter case, the following expression

$$\nabla^2 f(\mathbf{x}) \cong \frac{f(\mathbf{x} + h\mathbf{e}_i + h\mathbf{e}_j) - f(\mathbf{x} + h\mathbf{e}_i - h\mathbf{e}_j) - f(\mathbf{x} - h\mathbf{e}_i + h\mathbf{e}_j) + f(\mathbf{x} - h\mathbf{e}_i - h\mathbf{e}_j)}{4h^2}$$

estimates the ij^{th} Hessian term.

The `fd_hessian_step_size` specifies the relative finite difference step size to be used in these differences. Either a single value may be entered for use with all parameters, or a list of step sizes may be entered, one for each parameter. The differencing intervals are computed by multiplying the `fd_hessian_step_size` with the current parameter value. A minimum absolute differencing interval of `.01`$*$`fd_hessian_step_size` is used when the current parameter value is close to zero. Table 9.8 summarizes the numerical Hessian specification.

Description	Keyword	Associated Data	Status	Default
Numerical Hessians	`numerical_hessians`	none	Required group	N/A
Finite difference step size	`fd_hessian_step_size`	list of reals	Optional	`0.001` (1st-order), `0.002` (2nd-order)

Table 9.8: Specification detail for numerical Hessians

9.7.3 Quasi Hessians

The `quasi_hessians` specification means that Hessian information is needed and will be approximated using quasi-Newton secant updates. Compared to finite difference numerical Hessians, quasi-Newton 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 quasi-Newton 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 k^{th} 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 9.9 summarizes the quasi Hessian specification.

Description	Keyword	Associated Data	Status	Default
Quasi Hessians	`quasi_-hessians`	`bfgs` | `sr1`	Required group	N/A
Numerical safeguarding of BFGS update	`damped`	none	Optional	undamped BFGS

Table 9.9: Specification detail for quasi Hessians

9.7.4 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 File Data Formats in Users Manual) for the case of file transfer of data. The `analytic_hessians` keyword is a complete specification for this case.

9.7.5 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 quasi-Newton 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 quasi-Newton 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` quasi-Newton 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 9.10 summarizes the mixed Hessian specification.

Description	Keyword	Associated Data	Status	Default
Mixed Hessians	mixed_-hessians	none	Required group	N/A
Analytic Hessians function list	id_-analytic_-hessians	list of integers	Required	N/A
Numerical Hessians function list	id_-numerical_-hessians	list of integers	Required	N/A
Finite difference step size	fd_hessian_-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 9.10: Specification detail for mixed Hessians

Chapter 10

Bibliography

- Anderson, G., and Anderson, P., 1986 *The UNIX C Shell Field Guide*, Prentice-Hall, Englewood Cliffs, NJ.

- Argaez, M., Tapia, R. A., and Velazquez, L., 2002. "Numerical Comparisons of Path-Following Strategies for a Primal-Dual Interior-Point Method for Nonlinear Programming", *Journal of Optimization Theory and Applications*, Vol. 114 (2).

- Breitung, K., 1984. "Asymptotic approximation for multinormal integrals," *J. Eng. Mech., ASCE*, Vol. 110, No. 3, pp. 357-366.

- Byrd, R. H., Schnabel, R. B., and Schultz, G. A., 1988. "Parallel quasi-Newton Methods for Unconstrained Optimization," *Mathematical Programming*, 42(1988), pp. 273-306.

- 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.

- 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., Brown, S.∼L., Adams, B.∼M., Dunlavy, D.∼M., Gay, D.∼M., Swiler, L.∼P., Giunta, A.∼A., Hart, W.∼E., Watson, J.-P., Eddy, J.∼P., Griffin, J.∼D., Hough, P.∼D., Kolda, T.∼G., Martinez-Canales, M.∼L., and Williams, P.∼J., 2006. "DAKOTA: A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 4.0 Users Manual," Sandia Technical Report SAND2006-xxxx.

- 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.

- 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.

- 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.

- 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., 2001a. "SGOPT User Manual: Version 2.0," Sandia Technical Report SAND2001-3789.

- Hart, W. E., 2001b. "SGOPT Reference Manual: Version 2.0," Sandia Technical Report SAND2001-XXXX, In Preparation.

- 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.

- Hough, P. D., Kolda, T. G., and Torczon, V. J., 2000. "Asynchronous Parallel Pattern Search for Nonlinear Optimization," Sandia Technical Report SAND2000-8213, Livermore, CA.

- 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.

- 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., 1994. "OPT++: An Object-Oriented Class Library for Nonlinear Optimization," Sandia Report SAND94-8225, Sandia National Laboratories, Livermore, CA.

- More, J., and Thuente, D., 1994. "Line Search Algorithms with Guaranteed Sufficient Decrease," *ACM Transactions on Mathematical Software* 20(3):286-307.

- 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.

- 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.

- 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).

- 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.

- Weatherby, J. R., Schutt, J. A., Peery, J. S., and Hogan, R. E., 1996. "Delta: An Object-Oriented Finite Element Code Architecture for Massively Parallel Computers," Sandia Technical Report SAND96-0473.

- 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.

- Xu, S. and Grandhi, R. V., 1998. "Effective Two-Point Function Approximation for Design Optimization," AIAA Journal, Vol. 36, No. 12, December 1998.

DISTRIBUTION:

10	MS 0828	A. A. Giunta, 1533
10	MS 1318	M. S. Eldred, 1411
2	MS 9018	Central Technical Files, 8944
2	MS 0899	Technical Library, 4536