SAND2001-3515
Unlimited Release
Updated April 2003

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

Version 3.1 Reference Manual

Michael S. Eldred, Anthony A. Giunta, and Bart G. van Bloemen Waanders
Optimization and Uncertainty Estimation Department

Steven F. Wojtkiewicz, Jr.
Structural Dynamics Research Department

William E. Hart
Discrete Algorithms and Math Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0847

Mario P. Alleva
Compaq Federal
Albuquerque, New Mexico 87109-3432

Abstract

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

This report serves as a 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 , Bart G. van Bloemen Waanders , Steven F. Wojtkiewicz, Jr. , William E. Hart , Mario P. Alleva

1.1 Introduction

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

The 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., 2001].

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, variables, interface, and responses keyword specifications. The Reference Manual closely follows this structure, with introductory material followed by detailed documentation of the strategy, method, variables, interface, and responses keyword specifications:

Commands Introduction

Strategy Commands

Method Commands

Variables Commands

Interface Commands

Responses Commands

1.3 Web Resources

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`, `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 five 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 `application` specification within the interface keyword could be abbreviated as `applic`, but should not be abbreviated as `app` since this would be ambiguous with `approximation`.

- In both the input specification and user input files, comments are preceded by #.

The dakota.input.spec file used in DAKOTA V3.1 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 = 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_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_dist_lower_bounds = <LISTof><REAL>]                \
                [nuv_dist_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_dist_lower_bounds = <LISTof><REAL>]               \
                [lnuv_dist_upper_bounds = <LISTof><REAL>]               \
                [lnuv_descriptors = <LISTof><STRING>] ]                 \
        [ {uniform_uncertain = <INTEGER>}                               \
                {uuv_dist_lower_bounds = <LISTof><REAL>}                \
                {uuv_dist_upper_bounds = <LISTof><REAL>}                \
                [uuv_descriptors = <LISTof><STRING>] ]                  \
        [ {loguniform_uncertain = <INTEGER>}                            \
                {luuv_dist_lower_bounds = <LISTof><REAL>}               \
                {luuv_dist_upper_bounds = <LISTof><REAL>}               \
                [luuv_descriptors = <LISTof><STRING>] ]                 \
        [ {weibull_uncertain = <INTEGER>}                               \
                {wuv_alphas = <LISTof><REAL>}                           \
                {wuv_betas = <LISTof><REAL>}                            \
                [wuv_dist_lower_bounds = <LISTof><REAL>]                \
                [wuv_dist_upper_bounds = <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>] ]                  \
        [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>]                                       \
        ( {application}                                                 \
                {analysis_drivers = <LISTof><STRING>}                   \
                [input_filter = <STRING>]                               \
                [output_filter = <STRING>]                              \
                ( {system}                                              \
                  [parameters_file = <STRING>]                          \
                  [results_file = <STRING>]                             \
                  [analysis_usage = <STRING>]                           \
                  [aprepro] [file_tag] [file_save] )                    \
                |                                                       \
                ( {fork}                                                \
                  [parameters_file = <STRING>]                          \
                  [results_file = <STRING>]                             \
                  [aprepro] [file_tag] [file_save] )                    \
                |                                                       \
                ( {direct}                                              \
                  [processors_per_analysis = <INTEGER>]                 \
#                 [processors_per_analysis = <LISTof><INTEGER>]         \
                  [modelcenter_file = <STRING>] )                       \
                |                                                       \
                ( {grid}                                                \
                  {hostnames = <LISTof><STRING>}                        \
                  [processors_per_host = <LISTof><INTEGER>] )           \
                [ {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]                            \
                [ {failure_capture} {abort} | {retry = <INTEGER>} |     \
                  {recover = <LISTof><REAL>} | {continuation} ]         \
                [ {deactivate} [active_set_vector] [evaluation_cache]   \
                               [restart_file] ] )                       \
        |                                                               \
        ( {approximation}                                               \
                ( {global}                                              \
                  {neural_network} |                                    \
                  ( {polynomial} {linear} | {quadratic} | {cubic} ) |   \
                  {mars}           | {hermite}    |                     \
                  ( {kriging} [correlations = <LISTof><REAL>] )         \
                  [dace_method_pointer = <STRING>]                      \
                  [ {reuse_samples} {all} | {region} |                  \
                    {samples_file = <STRING>} ]                         \
                  [ {correction} {additive}     | {multiplicative}      \
                                 {zeroth_order} | {first_order} ]       \
#                 [ {rebuild} {inactive_all} | {inactive_region} ]      \
                  [use_gradients] )                                     \
                |                                                       \
                ( {multipoint}                                          \
#                 {tana?} [use_gradients?] [correction?]                \
                  {actual_interface_pointer = <STRING>} )               \
                |                                                       \
                ( {local}                                               \
                  {taylor_series}                                       \
                  {actual_interface_pointer = <STRING>}                 \
                  [actual_interface_responses_pointer = <STRING>] )     \
                |                                                       \
                ( {hierarchical}                                        \
                  {low_fidelity_interface_pointer = <STRING>}           \
                  {high_fidelity_interface_pointer = <STRING>}          \
#                 {high_fidelity_interface_responses_pointer = <STRING>}\
#                 {interface_pointer_hierarchy = <LISTof><STRING>}      \
                  {correction} {additive}     | {multiplicative}        \
                               {zeroth_order} | {first_order} ) )

<KEYWORD = responses>, <FUNCTION = responses_kwhandler>                 \
        [id_responses = <STRING>]                                       \
        [response_descriptors = <LISTof><STRING>]                       \
        ( {num_objective_functions = <INTEGER>}                         \
          [multi_objective_weights = <LISTof><REAL>]                    \
          [num_nonlinear_inequality_constraints = <INTEGER>]            \
          [nonlinear_inequality_lower_bounds = <LISTof><REAL>]          \
          [nonlinear_inequality_upper_bounds = <LISTof><REAL>]          \
          [num_nonlinear_equality_constraints = <INTEGER>]              \
          [nonlinear_equality_targets = <LISTof><REAL>] )               \
        |                                                               \
        ( {num_least_squares_terms = <INTEGER>}                         \
          [num_nonlinear_inequality_constraints = <INTEGER>]            \
          [nonlinear_inequality_lower_bounds = <LISTof><REAL>]          \
          [nonlinear_inequality_upper_bounds = <LISTof><REAL>]          \
          [num_nonlinear_equality_constraints = <INTEGER>]              \
          [nonlinear_equality_targets = <LISTof><REAL>] )               \
        |                                                               \
        {num_response_functions = <INTEGER>}                            \
        {no_gradients}                                                  \
        |                                                               \
        ( {numerical_gradients}                                         \
                [ {method_source} {dakota} | {vendor} ]                 \
                [ {interval_type} {forward} | {central} ]               \
                [fd_step_size = <REAL>] )                               \
        |                                                               \
        {analytic_gradients}                                            \
        |                                                               \
        ( {mixed_gradients}                                             \
                {id_numerical = <LISTof><INTEGER>}                      \
                  [ {method_source} {dakota} | {vendor} ]               \
                  [ {interval_type} {forward} | {central} ]             \
                  [fd_step_size = <REAL>]                               \
                {id_analytic = <LISTof><INTEGER>} )                     \
        {no_hessians}                                                   \
        |                                                               \
        {analytic_hessians}

<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>]                   \
                [ {trust_region}                                       \
                  [initial_size = <REAL>]                              \
                  [minimum_size = <REAL>]                              \
                  [contract_region_threshold = <REAL>]                 \
                  [expand_region_threshold = <REAL>]                   \
                  [contraction_factor = <REAL>]                        \
                  [expansion_factor = <REAL>] ] )                      \
        |                                                              \
        ( {opt_under_uncertainty}                                      \
                {opt_method_pointer = <STRING>} )                      \
        |                                                              \
        ( {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_type}                                                 \
          [variables_pointer= <STRING>]                                \
          [responses_pointer = <STRING>]                               \
          ( {single}  [interface_pointer = <STRING>] )                 \
        | ( {nested}  {sub_method_pointer = <STRING>}                  \
                    [ {interface_pointer = <STRING>}                   \
                      {interface_responses_pointer = <STRING>} ]       \
                      [primary_mapping_matrix = <LISTof><REAL>]        \
                      [secondary_mapping_matrix = <LISTof><REAL>] )    \
        | ( {layered} {interface_pointer = <STRING>} ) ]               \
        [speculative]                                                  \
        [ {output} {debug} | {verbose} | {quiet} | {silent} ]          \
[max_iterations = <INTEGER>]                                        \
[max_function_evaluations = <INTEGER>]                              \
[constraint_tolerance = <REAL>]                                     \
[convergence_tolerance = <REAL>]                                    \
[linear_inequality_constraint_matrix = <LISTof><REAL>]              \
[linear_inequality_lower_bounds = <LISTof><REAL>]                   \
[linear_inequality_upper_bounds = <LISTof><REAL>]                   \
[linear_equality_constraint_matrix = <LISTof><REAL>]                \
[linear_equality_targets = <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>] )                           \
|                                                                   \
( {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}                                                 \
                {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
                [ {pattern_basis} {coordinate} | {simplex} ]            \
                [total_pattern_size = <INTEGER>]                        \
                [no_expansion] [contraction_factor = <REAL>] )          \
        |                                                               \
        {coliny_direct}                                                 \
        |                                                               \
        ( {sgopt_pga_real}                                              \
                [solution_accuracy = <REAL>] [max_cpu_time = <REAL>]    \
                [seed = <INTEGER>] [population_size = <INTEGER>]        \
                [ {selection_pressure} {rank} | {proportional} ]        \
                [ {replacement_type} {random = <INTEGER>} |             \
                  {chc = <INTEGER>} | {elitist = <INTEGER>}             \
                  [new_solutions_generated = <INTEGER>] ]               \
                [ {crossover_type} {two_point} | {blend} | {uniform}    \
                  [crossover_rate = <REAL>] ]                           \
                [ {mutation_type} {replace_uniform} |                   \
                    ( {offset_normal}     [mutation_scale = <REAL>] ) | \
                    ( {offset_cauchy}     [mutation_scale = <REAL>] ) | \
                    ( {offset_uniform}    [mutation_scale = <REAL>] ) | \
                    ( {offset_triangular} [mutation_scale = <REAL>] )   \
                  [dimension_rate = <REAL>] [population_rate = <REAL>]  \
                  [non_adaptive] ] )                                    \
        |                                                               \
        ( {sgopt_pga_int}                                               \
                [solution_accuracy = <REAL>] [max_cpu_time = <REAL>]    \
                [seed = <INTEGER>] [population_size = <INTEGER>]        \
                [ {selection_pressure} {rank} | {proportional} ]        \
                [ {replacement_type} {random = <INTEGER>} |             \
                  {chc = <INTEGER>} | {elitist = <INTEGER>}             \
                  [new_solutions_generated = <INTEGER>] ]               \
                [ {crossover_type} {two_point} | {uniform}              \
                  [crossover_rate = <REAL>] ]                           \
                [ {mutation_type} {replace_uniform} |                   \
                    ( {offset_uniform} [mutation_range = <INTEGER>] )   \
                  [dimension_rate = <REAL>]                             \
                  [population_rate = <REAL>] ] )                        \
        |                                                               \
        ( {sgopt_epsa}                                                  \
                [solution_accuracy = <REAL>] [max_cpu_time = <REAL>]    \
                [seed = <INTEGER>] [population_size = <INTEGER>]        \
                [ {selection_pressure} {rank} | {proportional} ]        \
                [ {replacement_type} {random = <INTEGER>} |             \
                  {chc = <INTEGER>} | {elitist = <INTEGER>}             \
                  [new_solutions_generated = <INTEGER>] ]               \
                [ {crossover_type} {two_point} | {uniform}              \
                  [crossover_rate = <REAL>] ]                           \
        [ {mutation_type} {unary_coord} | {unary_simplex} |     \
            ( {multi_coord}   [dimension_rate = <REAL>] ) |     \
            ( {multi_simplex} [dimension_rate = <REAL>] )       \
          [mutation_scale = <REAL>] [min_scale = <REAL>]        \
          [population_rate = <REAL>] ]                          \
        [num_partitions = <INTEGER>] )                          \
|                                                               \
( {sgopt_pattern_search}                                        \
        [solution_accuracy = <REAL>] [max_cpu_time = <REAL>]    \
        [ {stochastic} [seed = <INTEGER>] ]                     \
        {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
        [ {pattern_basis} {coordinate} | {simplex} ]            \
        [total_pattern_size = <INTEGER>]                        \
        [no_expansion] [expand_after_success = <INTEGER>]       \
        [contraction_factor = <REAL>]                           \
        [ {exploratory_moves} {multi_step} | {best_all} |       \
          {best_first} | {biased_best_first} |                  \
          {adaptive_pattern} | {test} ] )                       \
|                                                               \
( {sgopt_solis_wets}                                            \
        [solution_accuracy = <REAL>] [max_cpu_time = <REAL>]    \
        [seed = <INTEGER>]                                      \
        {initial_delta = <REAL>} {threshold_delta = <REAL>}     \
        [no_expansion] [expand_after_success = <INTEGER>]       \
        [contract_after_failure = <INTEGER>]                    \
        [contraction_factor = <REAL>] )                         \
|                                                               \
( {sgopt_strat_mc}                                              \
        [solution_accuracy = <REAL>] [max_cpu_time = <REAL>]    \
        [seed = <INTEGER>] [batch_size = <INTEGER>]             \
        [partitions = <LISTof><INTEGER>] )                      \
|                                                               \
( {nond_polynomial_chaos}                                       \
        {expansion_terms = <INTEGER>} |                         \
        {expansion_order = <INTEGER>}                           \
        [seed = <INTEGER>] [samples = <INTEGER>]                \
        [ {sample_type} {random} | {lhs} ]                      \
        [response_thresholds = <LISTof><REAL>] )                \
|                                                               \
( {nond_sampling}                                               \
        [seed = <INTEGER>] [fixed_seed]                         \
        [samples = <INTEGER>]                                   \
        [ {sample_type} {random} | {lhs} ]                      \
        [all_variables]                                         \
        [response_thresholds = <LISTof><REAL>] )                \
|                                                               \
( {nond_analytic_reliability}                                   \
        ( {mv} [response_levels = <LISTof><REAL>] )         |   \
        ( {amv} {response_levels = <LISTof><REAL>} )        |   \
        ( {iterated_amv} {response_levels = <LISTof><REAL>} |   \
          {probability_levels = <LISTof><REAL>} )           |   \
        ( {form} {response_levels = <LISTof><REAL>} )       |   \
        ( {sorm} {response_levels = <LISTof><REAL>} ) )         \
|                                                               \
( {dace}                                                        \
        {grid} | {random} | {oas} | {lhs} | {oa_lhs} |          \
        {box_behnken} | {central_composite}                     \
        [seed = <INTEGER>] [fixed_seed]                         \
        [samples = <INTEGER>] [symbols = <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>} )
```

Chapter 3

Commands Introduction

3.1 Overview

In the DAKOTA system, a *strategy* governs how each *method* maps *variables* into *responses* through the use of an *interface*. Each of these five pieces (strategy, method, variables, responses, and interface) 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 (in general) have its own "model," consisting of its own set of variables, its own interface, and its own set of responses. Thus, there may be multiple specifications of the method, 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 provides a template of 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., 2001].

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 dakota.input.spec 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 variables keyword is that of ten optional group specifications for continuous design, discrete design, normal uncertain, lognormal uncertain, uniform uncertain, loguniform uncertain, weibull uncertain, histogram uncertain, continuous state, and discrete state variables. Each of these specifications can either appear or not appear as a group. Next, the interface keyword requires the selection of either an application OR an approximation interface. The type of application interface must be specified with either a system OR fork OR direct OR grid required group specification, or the type of approximation interface must be specified with either a global OR multipoint OR local OR hierarchical required group specification. 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 analytic). The strategy specification requires either a multi-level OR surrogate-based optimization OR optimization under uncertainty OR branch and bound OR multi-start OR pareto set OR single method strategy specification. Lastly, the method keyword is the most lengthy specification; however, its structure is 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. Refer to Strategy Commands, Method 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:

- Documentation of new capability sometimes lags the use of new capability in executables (especially experimental 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.

- 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
```

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

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,                                              \
        application 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 OPT++'s Gauss-Newton method. A similar file is available in the test directory as `Dakota/test/dakota_rosenbrock.in`.

```
strategy,                                       \
        single_method

method,                                         \
        optpp_g_newton                          \
          max_iterations = 50,                  \
          convergence_tolerance = 1e-4
```

```
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,                                      \
        application system                      \
          analysis_driver = 'rosenbrock_ls'

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_thresholds = 3.6e+11 6.e+04 3.5e+05

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,                                                      \
        application 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. 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,                                                  \
        application 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' 'CPS' 'NLP'

method,                                             \
        id_method = 'GA'                            \
        model_type single                           \
          variables_pointer = 'V1'                  \
          interface_pointer = 'I1'                  \
          responses_pointer = 'R1'                  \
        sgopt_pga_real                              \
          population_size = 10                      \
          output verbose

method,                                             \
        id_method = 'PS'                            \
        model_type single                           \
          variables_pointer = 'V1'                  \
          interface_pointer = 'I1'                  \
          responses_pointer = 'R1'                  \
        sgopt_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_type single                     \
          variables_pointer = 'V1'            \
          interface_pointer = 'I1'            \
          responses_pointer = 'R2'            \
        optpp_newton                          \
          gradient_tolerance = 1.e-12         \
          convergence_tolerance = 1.e-15

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'                   \
        application 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., 2001].

3.5 Tabular descriptions

In the following discussions of keyword specifications, tabular formats (Tables 4.1 through 8.7) 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 an interface keyword, the `parameters_file` specification is an optional specification within the `system` and `fork` required group specifications, which are separated from each other and from other required group specifications (`direct` and `grid`) by logical OR's. The selection between the `system`, `fork`, `direct`, or `grid` required groups is contained within another required group specification (`application`), which is separated from the `approximation` required group specification by a logical OR. 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., `application` in Table 7.2), followed by lower levels of specifications (e.g., `system`, `fork`, `direct`, or `grid` in Tables 7.3 through 7.6) 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. Seven strategies currently exist: `multi_level`, `surrogate_based_opt`, `opt_under_uncertainty`, `branch_and_bound`, `multi_start`, `pareto_set`, and `single_method`. These algorithms are implemented within the **DakotaStrategy** class hierarchy in the **MultilevelOptStrategy**, **SurrBasedOptStrategy**, **NonDOptStrategy**, **BranchBndStrategy**, **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.

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 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 model, i.e., a surrogate model, that undergoes periodic re-calibration using data from a ”truth” model. The surrogate model can be either a surface fit model or a low-fidelity simulation model, whereas the truth model typically is 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. This surrogate model can be a global data fit (e.g., a smoothing polynomial or an interpolation function built from a design of computer experiments database), a multipoint approximation, a local Taylor Series expansion, or a hierarchical approximation (e.g., a low-fidelity simulation model calibrated to match the data generated by a high fidelity model). The trust region strategy performs a sequence of optimization runs using the surrogate model. At the end of each optimization run, the candidate optimum point found by the optimizer is evaluated using both the surrogate model and 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 amount of truth function decrease. If sufficient decrease has not been attained, the trust region center point does not move 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 surface fit surrogate models, to smooth noisy data with an easily navigated analytic function.

In optimization under uncertainty (`opt_under_uncertainty`), a nondeterministic iterator is used to evaluate the effect of uncertain variables, modeled using probabilistic distributions, on responses of interest. Statistics on these responses are then included in the objective and constraint functions of the optimization problem (for example, to minimize probability of failure). The nondeterministic iterator may be nested directly within the optimization function evaluations, which can be prohibitively expensive, or the direct nesting can be broken through a variety of surrogate-based optimization under uncertainty formulations. The sub-model recursion features of **NestedModel**, **SurrLayeredModel**, and **HierLayeredModel** enable these formulations.

In the branch and bound strategy (`branch_and_bound`), mixed integer nonlinear programs (nonlinear applications with a mixture of continuous and discrete variables) can be solved through the combination of the PICO parallel branching algorithm with the nonlinear programming algorithms available in DAKOTA. Since PICO supports *parallel* branch and bound techniques, multiple bounding operations can be performed concurrently for different branches, which provides for concurrency in nonlinear optimizations for DAKOTA. This is an additional level of parallelism, beyond those for concurrent evaluations within an iterator, concurrent analyses within an evaluation, and multiprocessor analyses. Branch and bound is applicable when the discrete variables can assume continuous values during the solution process (i.e., the integrality conditions are relaxable). It proceeds by performing a series of continuous-valued optimizations for different variable bounds which, in the end, drive the discrete variables to integer values.

In the multi-start iteration strategy (`multi_start`), a series of iterator runs are performed for different values of some parameters in the model. A common use is for multi-start optimization (i.e., different 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, similar to the branch and bound strategy discussed above.

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. An important feature is that these iterator runs can be performed concurrently, similar to the branch and bound and multi-start strategies 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 the variables, interface, and responses specifications (using `variables_pointer`, `interface_pointer`, and `responses_pointer` from Method Independent Controls) that are used to build the model used by the iterator. If a method specification does not provide a particular pointer, then that component of the model will be built 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 = ‘GA1’, ‘CPS1’, ‘NLP1’
```

A `surrogate_based_opt` example specification is:

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

An `opt_under_uncertainty` example specification is:

```
strategy,                                    \
        opt_under_uncertainty                \
          opt_method_pointer = ‘NLP1’
```

A `branch_and_bound` example specification is:

```
strategy,                                    \
        iterator_servers = 4                 \
        branch_and_bound                     \
          opt_method_pointer = ‘NLP1’
```

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`, `opt_under_uncertainty`, `branch_and_bound`, `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 `opt_under_uncertainty` OR `branch_and_bound` 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., `branch_and_bound`, `multi_start`, and `pareto_set` (`opt_under_uncertainty` will support this in the future once full **NestedModel** parallelism support is in place). Table 4.1 summarizes the strategy independent controls.

Table 4.1 Specification detail for 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

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.

Table 4.2 Specification detail for uncoupled multi-level strategies

Description	Keyword	Associated Data	Status	Default
Multi-level hybrid strategy	`multi_level`	none	Required group (1 of 7 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

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 `lo-cal_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.

Table 4.3 Specification detail for coupled multi-level strategies

Description	Keyword	Associated Data	Status	Default
Multi-level hybrid strategy	`multi_level`	none	Required group (1 of 7 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

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 responsible for selecting a `layered` model for use as the surrogate (see Method Independent Controls). Algorithm controls include `max_iterations` (the maximum number of SBO cycles allowed), `convergence_tolerance` (the relative tolerance used in internal SBO convergence assessments), and `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). In addition, the `trust_region` optional group specification can be used to specify the initial size of the trust region (using `initial_size)`, 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_region_-threshold)` and the trust region size expansion threshold (using `expand_region_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_region_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_region_threshold` determines the trust region value above which the trust region will expand for the next SBO iteration. Table 4.4 summarizes the surrogate based optimization strategy inputs.

Table 4.4 Specification detail for surrogate based optimization strategies

Description	Keyword	Associated Data	Status	Default
Surrogate-based optimization strategy	`surrogate_-based_opt`	none	Required group (1 of 7 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	`conver-gence_-tolerance`	real	Optional	1.e-4
Soft convergence limit for SBO iterations	`soft_-convergence_-limit`	integer	Optional	5
Trust region group specification	`trust_region`	none	Optional group	N/A
Trust region initial size	`initial_size`	real	Optional	0.05
Trust region minimum size	`minimum_size`	real	Optional	1.e-6
Shrink trust region if trust region ratio is below this value	`contract_-region_-threshold`	real	Optional	0.25
Expand trust region if trust region ratio is above this value	`expand_-region_-threshold`	real	Optional	0.75
Trust region contraction factor	`contrac-tion_factor`	real	Optional	0.25
Trust region expansion factor	`expansion_-factor`	real	Optional	2.0

4.6 Optimization Under Uncertainty Commands

The `opt_under_uncertainty` strategy must specify an optimization iterator using `opt_method_-pointer`. In the case of a direct nesting of an uncertainty quantification iterator within the top level model, the method specification identified by `opt_method_pointer` would select a `nested` model (see Method Independent Controls). In the case of surrogate-based optimization under uncertainty, the method specification identified by `opt_method_pointer` might select either a `nested` model or a `layered` model, since the recursive properties of **NestedModel**, **SurrLayeredModel**, and **HierLayered-Model** could be utilized to configure any of the following:

- ”layered containing nested” (i.e., optimization of a data fit surrogate built using statistical data from nondeterministic analyses)
- ”nested containing layered” (i.e., optimization using nondeterministic analysis data evaluated from a data fit or hierarchical surrogate)
- ”layered containing nested containing layered” (i.e., combination of the two above: optimization of

a data fit surrogate built using statistical data from nondeterministic analyses, where the nondeterministic analyses are performed on a data fit or hierarchical surrogate)

Since most of the sophistication is encapsulated within the nested and layered model classes (see nested/layered specifications in Method Independent Controls), the optimization under uncertainty strategy inputs are minimal. Table 4.5 summarizes these inputs.

Table 4.5 Specification detail for optimization under uncertainty strategies

Description	Keyword	Associated Data	Status	Default
Optimization under uncertainty strategy	`opt_under_uncertainty`	none	Required group (1 of 7 selections)	N/A
Optimization method pointer	`opt_method_pointer`	string	Required	N/A

4.7 Branch and Bound Commands

The `branch_and_bound` strategy must specify an optimization method using `opt_method_pointer`. This optimization method is responsible for computing optimal solutions to nonlinear programs which arise from different *branches* of the mixed variable problem. These branches correspond to different bounds on the discrete variables where the integrality constraints on these variables have been relaxed. Solutions which are completely feasible with respect to the integrality constraints provide an upper *bound* on the final solution and can be used to prune branches which are not yet integer-feasible and which have higher objective functions. The optional `num_samples_at_root` and `num_samples_at_node` specifications specify the number of additional function evaluations to perform at the root of the branching structure and at each node of the branching structure, respectively. These samples are selected randomly within the current variable bounds of the branch. This feature is a simple way to globalize the optimization of the branches, since nonlinear problems may be multimodal. Table 4.6 summarizes the branch and bound strategy inputs.

Table 4.6 Specification detail for branch and bound strategies

Description	Keyword	Associated Data	Status	Default
Branch and bound strategy	`branch_and_bound`	none	Required group (1 of 7 selections)	N/A
Optimization method pointer	`opt_method_pointer`	string	Required	N/A
Number of samples at the branching root	`num_samples_at_root`	integer	Optional	0
Number of samples at each branching node	`num_samples_at_node`	integer	Optional	0

4.8 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.

Table 4.7 Specification detail for multi-start strategies

Description	Keyword	Associated Data	Status	Default
Multi-start iteration strategy	`multi_start`	none	Required group (1 of 7 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

4.9 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.

Table 4.8 Specification detail for pareto set strategies

Description	Keyword	Associated Data	Status	Default
Pareto set optimization strategy	`pareto_set`	none	Required group (1 of 7 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

4.10 Single Method Commands

The single method strategy is the default if no strategy specification is included in a user input file. It may also be specified using the `single_method` keyword within a strategy specification. An optional `method_pointer` specification may be used to point to a particular method specification. If `method_pointer` is not used, then the last method specification parsed will be used as the iterator. Table 4.9 summarizes the single method strategy inputs.

Table 4.9 Specification detail for single method strategies

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Single method strategy	`single_method`	string	Required group (1 of 7 selections)	N/A
Method pointer	`method_pointer`	string	Optional	use of last method parsed

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 **DakotaIterator** 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 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_type single               \
          variables_pointer = 'V1'      \
          interface_pointer = 'I1'      \
          responses_pointer = 'R1'      \
        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_thresholds = 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`, `reduced_sqp`, `optpp_cg`, `optpp_q_newton`, `optpp_fd_newton`, `optpp_g_newton`, `optpp_newton`, `optpp_pds`, `coliny_apps`, `coliny_direct`, `sgopt_pga_real`, `sgopt_pga_int`, `sgopt_epsa`, `sgopt_pattern_search`, `sgopt_solis_wets`, `sgopt_strat_mc`, `nond_polynomial_chaos`, `nond_sampling`, `nond_analytic_reliability`, `dace`, `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, 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 type of model to be used by the method is supplied with `model_type` and can be `single`, `nested`, or `layered` (refer to **DakotaModel** for the class hierarchy involved). In the `single` model case, the optional `variables_pointer`, `interface_pointer`, and `responses_pointer` specifications provide strings for cross-referencing with `id_variables`, `id_interface`, and `id_responses` string inputs from particular variables, interface, and responses keyword specifications. These pointers identify which specifications will be used in building the single model, which is to be iterated by the method to map the variables into responses through the interface. In the `layered` model case, the specification is similar, except that the `interface_pointer` specification is required in order to identify a global, multipoint, local, or hierarchical approximation interface (see Approximation Interface) to use in the layered model. In the `nested` model case, a `sub_method_pointer` must be provided in order to specify the nested iterator, and `interface_pointer` and `interface_responses_pointer` provide an optional group specification for the optional interface portion of nested models (where `interface_pointer` points to the interface specification and `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_mapping_matrix` and `secondary_mapping_matrix` inputs (refer to **NestedModel::response_mapping**() for additional information). In all cases, if a pointer string is specified and no corresponding id is available, DAKOTA will exit with an error message. If no pointer string is specified, the last specification parsed will be used. It is appropriate to omit this cross-referencing whenever the relationships are unambiguous due to the presence of only one specification. Since the method specification is responsible for cross-referencing with the interface, variables, and responses specifications, identification of methods at the strategy layer is often sufficient to completely specify all of the object interrelationships.

Table 5.1 provides the specification detail for the method independent controls involving identifiers, pointers, and model type controls.

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

Description	Keyword	Associated Data	Status	Default
Method set identifier	`id_method`	String	Optional	strategy use of last method parsed
Model type	`model_type`	`single` | `nested` | `layered`	Optional group	`single`
Variables set pointer	`variables_-pointer`	String	Optional	method use of last variables parsed
Interface set pointer	`interface_-pointer`	String	`single`: Optional, `nested`: Optional group, `layered`: Required	`single`: method use of last interface parsed, `nested`: no optional interface, `layered`: N/A
Responses set pointer	`responses_-pointer`	String	Optional	method use of last responses parsed
Sub-method pointer for nested models	`sub_method_-pointer`	String	Required	N/A
Responses pointer for nested model optional interfaces	`interface_-responses_-pointer`	String	Required	N/A
Primary mapping matrix for nested models	`primary_-mapping_-matrix`	list of reals	Optional	no sub-iterator contribution to primary functions
Secondary mapping matrix for nested models	`secondary_-mapping_-matrix`	list of reals	Optional	no sub-iterator contribution to secondary functions

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 **DakotaIterator** (algorithm verbosity), **DakotaModel** (synchronize/fd_gradients verbosity), **DakotaInterface** (map/synch verbosity), **DakotaApproximation** (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 SGOPT) 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 SGOPT 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).

Table 5.2 provides the specification detail for the method independent controls involving tolerances, limits, output verbosity, and speculative gradients.

Table 5.2 Specification detail for the method independent controls: tolerances, limits, output verbosity, and speculative gradients

Description	Keyword	Associated Data	Status	Default
Speculative gradients and Hessians	`speculative`	none	Optional	no speculation
Output verbosity	`output`	`silent` | `quiet` | `verbose` | `debug`	Optional	`normal`
Maximum iterations	`max_-iterations`	integer	Optional	`100`
Maximum function evaluations	`max_-function_-evaluations`	integer	Optional	`1000`
Constraint tolerance	`constraint_-tolerance`	real	Optional	Library default
Convergence tolerance	`conver-gence_-tolerance`	real	Optional	`1.e-4`

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 +bigBoundSize (1.e+30, as defined in **DakotaOptimizer**) are treated as +infinity and any lower bound values less than -bigBoundSize 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 < -bigBoundSize). 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 `0.0` for each constraint

$$Ax = 0.0$$

Currently, DOT, CONMIN, NPSOL, NLSSOL, and OPT++ all support specialized handling of linear constraints. SGOPT 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 bigBoundSize. 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.

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

Table 5.3 Specification detail for the method independent controls: linear inequality and equality 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 = `0.0`
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 = `0.0`

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 5.3.

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 5.4 provides the specification detail for the DOT methods and their method dependent controls.

Table 5.4 Specification detail for the DOT methods

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

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 5.3 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 5.5 provides the specification detail for the NPSOL SQP method and its method dependent controls.

Table 5.5 Specification detail for the NPSOL SQP method

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)

5.6 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.

5.6.1 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.

5.6.2 CONMIN method dependent controls

CONMIN does not currently support any method dependent controls.

5.7 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.

5.7.1 OPT++ method independent controls

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

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

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

5.7.2 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 5.6 covers this specification.

Table 5.6 Specification detail for the OPT++ conjugate gradient method

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`

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 5.7 provides the details for the Newton-based methods.

Table 5.7 Specification detail for OPT++ Newton-based optimization methods

Description	**Keyword**	**Associated Data**	**Status**	**Default**
OPT++ Newton-based methods	`optpp_q_newton` | `optpp_fd_newton` | `optpp_newton`	none	Required group	N/A
Search method	`value_based_line_search` | `gradient_based_line_search` | `trust_region` | `tr_pds`	none	Optional group	`trust_region` (unconstrained), `value_based_line_search` (bound/general constraints)
Maximum step size	`max_step`	real	Optional	`1000`.
Gradient tolerance	`gradient_tolerance`	real	Optional	`1.e-4`
Merit function	`merit_function`	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"`)

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 5.8 provides the detail for the parallel direct search method.

Table 5.8 Specification detail for the OPT++ PDS method

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

5.8 Asynchronous Parallel Pattern Search Method

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 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. It utilizes the nonblocking schedulers in DAKOTA (see **DakotaModel::synchronize_nowait**()). APPS is currently interfaced to DAKOTA through use of the COLINY library (method `coliny_apps)`, where COLINY is a collection of optimizers that support the Common Optimization Library INterface (COLIN). Other COLINY optimizers (e.g., `coliny_direct)` will be added in future releases.

5.8.1 APPS method independent controls

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

- DAKOTA "debug"/"verbose": APPS debug level = 10, profile level = 1
- DAKOTA "normal": APPS debug level = 2, profile_level = 1
- DAKOTA "quiet"/"silent": APPS debug level = 0, profile level = 0

5.8.2 APPS method dependent controls

The APPS method is invoked using a `coliny_apps` group specification. Components within this specification group include `initial_delta`, `threshold_delta`, `pattern_basis`, `total_pattern_-size`, `no_expansion`, and `contraction_factor`. The `initial_delta` and `threshold_-delta` specifications are required in order to provide the initial offset size and the threshold size at which to terminate the algorithm, respectively. These sizes are dimensional and are not relative to the bounded region (as they are with `sgopt_pattern_search)`. The `pattern_basis` specification is used to select between a `coordinate` basis or a `simplex` basis. The former uses a plus and minus offset in each coordinate direction, for a total of *2n* function evaluations in the pattern, whereas the latter uses a minimal positive basis simplex for the parameter space, for a total of *n+1* function evaluations in the pattern. 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 `no_expansion` flag instructs the algorithm to omit pattern expansion, which is normally performed after a sequence of improving offsets is found. Finally, the `contraction_factor` specification selects the scaling factor used in computing a reduced offset for a new pattern search cycle after the previous cycle has been unsuccessful in finding an improved point. Table 5.9 summarizes the APPS specification.

Table 5.9 Specification detail for the APPS method

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 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
Pattern contraction factor	`contrac-tion_factor`	real	Optional	`0.5`

5.9 SGOPT Methods

The SGOPT (Stochastic Global OPTimization) library [Hart, W.E., 2001a; Hart, W.E., 2001b] contains a variety of nongradient-based optimization algorithms, with an emphasis on stochastic global methods. SGOPT currently includes the following global optimization methods: evolutionary algorithms (`sgopt_-pga_real`, `sgopt_pga_int`, and `sgopt_epsa`) and stratified Monte Carlo (`sgopt_strat_mc`). Additionally, SGOPT includes nongradient-based local search algorithms such as Solis-Wets (`sgopt_-solis_wets`) and pattern search (`sgopt_pattern_search`). With the exception of the unconstrained `sgopt_solis_wets` method, each of the SGOPT methods support bound constraints. DAKOTA provides access to the SGOPT library through the **SGOPTOptimizer** class.

5.9.1 SGOPT 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 SGOPT 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 by SGOPT: the `silent`, `quiet`, and `normal` settings correspond to minimal reporting from SGOPT, whereas the `verbose` setting corresponds to a higher level of information, and `debug` outputs method initialization and a variety of internal SGOPT diagnostics. The majority of SGOPT's methods have independent function evaluations that can directly take advantage of DAKOTA's parallel capabilities. Only `sgopt_solis_wets` and certain `exploratory_-moves` options in `sgopt_pattern_search` (`multi_step`, `best_first`, `biased_best_first`, and `adaptive_pattern`; see Pattern search) are inherently serial. The parallel methods automatically utilize parallel logic when the DAKOTA configuration supports parallelism. Lastly, neither `specula-tive` gradients nor specialized handling of linear constraints are currently supported with SGOPT since SGOPT methods are nongradient-based and support, at most, bound constraints. Specification detail for method independent controls is provided in Tables 5.1 through 5.3.

5.9.2 SGOPT method dependent controls

`solution_accuracy` and `max_cpu_time` are method dependent controls which are defined for all SGOPT methods. Solution accuracy defines a convergence criterion in which the optimizer will terminate if it finds an objective function value lower than the specified accuracy. The maximum CPU time setting is another convergence criterion in which the optimizer will terminate if its CPU usage in seconds exceeds the specified limit. Table 5.10 provides the specification detail for these recurring method dependent controls.

Table 5.10 Specification detail for SGOPT method dependent controls

Description	Keyword	Associated Data	Status	Default
Desired solution accuracy	`solution_-accuracy`	real	Optional	`-DBL_MAX`
Maximum amount of CPU time	`max_cpu_time`	real	Optional	unlimited CPU

Each SGOPT method supplements the settings of Table 5.10 with controls which are specific to its particular class of method.

5.9.3 Evolutionary Algorithms

DAKOTA currently provides three types of evolutionary algorithms (EAs): a real-valued genetic algorithm (`sgopt_pga_real`), an integer-valued genetic algorithm (`sgopt_pga_int`), and an evolutionary pattern search technique (`sgopt_epsa`), where ”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 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

Controls for seed, population size, selection, and replacement are identical for the three EA methods, whereas the crossover and mutation controls contain slight differences and the `sgopt_epsa` specification contains an additional `num_partitions` input. Table 5.11 provides the specification detail for the controls which are common between the three EA methods.

Table 5.11 Specification detail for the SGOPT EA methods

Description	Keyword	Associated Data	Status	Default
EA selection	`sgopt_pga_real` | `sgopt_pga_int` | `sgopt_epsa`	none	Required group	N/A
Random seed	`seed`	integer	Optional	randomly generated seed
Number of population members	`population_size`	integer	Optional	`100`
Selection pressure	`selection_pressure`	`rank` | `proportional`	Optional	`proportional`
Replacement type	`replacement_type`	`random` | `chc` | `elitist`	Optional group	`random` = `0`
Random replacement	`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`

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

- the `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 `proportional` 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 will is referred to below and in Table 5.11 as the `replacement_size`:

- The `random` setting (the default) 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. CHC is the preferred selection for many engineering problems.

- The `elitist` 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 5.12, Table 5.13, and Table 5.14 show the controls which differ between `sgopt_pga_real`, `sgopt_pga_int`, and `sgopt_epsa`, respectively.

Table 5.12 Specification detail for SGOPT real-valued genetic algorithm crossover and mutation

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` | `offset_-triangular`	Optional group	`offset_-normal`
Mutation scale	`mutation_-scale`	real	Optional	`0.1`
Mutation dimension rate	`dimension_-rate`	real	Optional	$\frac{\sqrt{e/n}}{population\_size}$
Mutation population rate	`population_-rate`	real	Optional	`1.0`
Non-adaptive mutation flag	`non_adaptive`	none	Optional	Adaptive mutation

Table 5.13 Specification detail for SGOPT integer-valued genetic algorithm crossover and mutation

Description	Keyword	Associated Data	Status	Default
Crossover type	`crossover_-type`	`two_point` | `uniform`	Optional group	`two_point`
Crossover rate	`crossover_-rate`	real	Optional	`0.8`
Mutation type	`mutation_-type`	`replace_-uniform` | `offset_-uniform`	Optional group	`replace_-uniform`
Mutation range	`mutation_-range`	integer	Optional	`1`
Mutation dimension rate	`dimension_-rate`	real	Optional	$\frac{\sqrt{e/n}}{population\_size}$
Mutation population rate	`population_-rate`	real	Optional	`1.0`

Table 5.14 Specification detail for SGOPT evolutionary pattern search crossover, mutation, and number of partitions

Description	Keyword	Associated Data	Status	Default
Crossover type	crossover_type	two_point | uniform	Optional group	two_point
Crossover rate	crossover_rate	real	Optional	0.8
Mutation type	mutation_type	unary_coord | unary_simplex | multi_coord | multi_simplex	Optional group	unary_coord
Mutation dimension rate	dimension_rate	real	Optional	$\frac{\sqrt{e/n}}{population\_size}$
Mutation scale	mutation_scale	real	Optional	0.1
Minimum mutation scale	min_scale	real	Optional	0.001
Mutation population rate	population_rate	real	Optional	1.0
Number of partitions	num_partitions	integer	Optional	100

The crossover_type controls what approach is employed for combining parent genetic information to create offspring, and the crossover_rate specifies the probability of a crossover operation being performed to generate a new offspring. SGOPT supports two generic forms of crossover, two_point and uniform, which generate a new individual through coordinate-wise 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 SGOPT 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. The sgopt_pga_real optimizer supports a third option, the blend crossover method, which generates a new individual randomly along the multidimensional vector connecting the two parents.

The mutation_type controls what approach is employed in randomly modifying 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). The population_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 (if crossover does not occur; see algorithm description in Evolutionary Algorithms). The dimension_rate specifies the probabilities that a given dimension is changed given that the individual is having mutation applied to it. The default dimension_rate uses the special formula shown in the preceding tables, where *n* is the number of design variables and *e* is the natural logarithm constant. The mutation_scale specifies a scale factor which scales mutation offsets for sgopt_pga_real and sgopt_epsa; 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 provides an analogous control for sgopt_pga_int, but is not a relative value in that it specifies the total integer range of the mutation. The offset_normal, offset_cauchy, offset_uniform, and offset_triangular mutation types are ”offset” mutations in that they add a 0-mean random variable with a normal, cauchy, uniform, or triangular 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 real-valued genetic algorithm supports each of these 5 mutation types, and integer-valued genetic algorithm supports the `replace_uniform` and `offset_uniform` types. The mutation types for evolutionary pattern search are more specialized:

- `multi_coord`: Mutate each coordinate dimension with probability `dimension_rate` using an "offset" approach with initial scale `mutation_scale` $*$ variable range. Multiple coordinates may or may not be mutated.
- `unary_coord`: Mutate a single randomly selected coordinate dimension using an "offset" approach with initial scale `mutation_scale` $*$ variable range. One and only one coordinate is mutated.
- `multi_simplex`: Apply each of the vector offsets from a regular simplex (n+1 vectors for n dimensions) with probability `dimension_rate` and initial scale `mutation_scale` $*$ variable range. A single vector offset may alter multiple coordinate dimensions. Multiple simplex vectors may or may not be applied.
- `unary_simplex`: Add a single randomly selected vector offset from a regular simplex with an initial scale of `mutation_scale` $*$ variable range. One and only one simplex vector is applied, but this simplex vector may alter multiple coordinate dimensions.

and are described in more detail in [Hart and Hunter, 1999]. Both the real-valued genetic algorithm and the evolutionary pattern search algorithm use adaptive mutation that modifies the mutation scale dynamically. The `non_adaptive` flag can be used to deactivate the self-adaptation in real-valued genetic algorithms, which may facilitate a more global search. The adaptive mutation in evolutionary pattern search is an inherent component that cannot be deactivated. The `min_scale` input specifies the minimum mutation scale for evolutionary pattern search; `sgopt_epsa` terminates if the adapted mutation scale falls below this threshold.

The `num_partitions` specification is not part of the crossover or mutation group specifications; it specifies the number of possible values for each dimension (fractions of the variable ranges) used in the initial evolutionary pattern search population. It is needed for theoretical reasons.

For additional information on these options, see the user and reference manuals for SGOPT [Hart, 2001a; Hart, 2001b].

5.9.4 Pattern search

SGOPT provides a pattern search technique (`sgopt_pattern_search)` whose operation and controls are similar to that of APPS (see Asynchronous Parallel Pattern Search Method). Table 5.15 provides the specification detail for the SGOPT PS method and its method dependent controls.

Table 5.15 Specification detail for the SGOPT pattern search method

Description	Keyword	Associated Data	Status	Default
SGOPT pattern search method	sgopt_-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
Pattern basis selection	pattern_-basis	coordinate | simplex	Optional	simplex
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	contrac-tion_factor	real	Optional	0.5
Exploratory moves selection	ex-ploratory_-moves	multi_step | best_all | best_first | biased_-best_first | adaptive_-pattern | test	Optional group	best_first for serial, best_all for parallel

The `initial_delta`, `threshold_delta`, `pattern_basis`, `total_pattern_size`, `no_-expansion`, and `contraction_factor` controls are identical in meaning to the corresponding APPS controls (see Asynchronous Parallel Pattern Search Method). Differing controls include the `stochas-tic`, `seed`, `expand_after_success`, and `exploratory_moves` specifications. The SGOPT pattern search provides the capability for `stochastic` shuffling of offset evaluation order, for which the random `seed` can be used to make the optimizations repeatable. The `expand_after_success` control specifies how many successful objective function improvements must occur with a specific delta prior to expansion of the delta.

The `exploratory_moves` setting controls how the offset evaluations are ordered as well as the logic for acceptance of an improved point. The following exploratory moves selections are supported by SGOPT:

- 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 `best_all` case waits for completion of all offset evaluations in the pattern before selecting a new iterate. This method is most appropriate for parallel execution of the pattern search.

- The `best_first` case immediately selects the first improving point found as the new iterate, without waiting for completion of all offset evaluations in the cycle. This option does not support any parallelism and will result in a serial pattern search.

- The `biased_best_first` case immediately selects the first improved point as the new iterate, but also introduces a bias toward directions in which improving points have been found previously by reordering the offset evaluations. 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 `best_first` case. This option does not support any parallelism and will result in a serial pattern search.

- The `test` case is used for development purposes. This currently utilizes a nonblocking scheduler (i.e., **DakotaModel::synchronize_nowait**()) for performing the function evaluations.

5.9.5 Solis-Wets

DAKOTA's implementation of SGOPT 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.

Table 5.16 Specification detail for the SGOPT Solis-Wets method

Description	**Keyword**	**Associated Data**	**Status**	**Default**
SGOPT Solis-Wets method	`sgopt_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`

The `seed`, `initial_delta`, `threshold_delta`, `no_expansion`, `expand_after_success`, and `contraction_factor` specifications have identical meaning to the corresponding specifications for `coliny_apps` and `sgopt_pattern_search` (see Asynchronous Parallel Pattern Search Method and Pattern search). 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.

5.9.6 Stratified Monte Carlo

Lastly, DAKOTA's implementation of SGOPT contains a stratified Monte Carlo (sMC) algorithm. One of the distinguishing characteristics of this sampling technique from other sampling methods in Design of Computer Experiments Methods and Nondeterministic sampling method is its stopping criteria. Using `solution_accuracy` (see SGOPT method dependent controls), the sMC algorithm can terminate adaptively when a design point with a desired performance has been located. Table 5.17 provides the specification detail for this method and its method dependent controls.

Table 5.17 Specification detail for the SGOPT sMC method

Description	Keyword	Associated Data	Status	Default
SGOPT stratified Monte Carlo method	`sgopt_-strat_mc`	none	Required group	N/A
Random seed for stochastic pattern search	`seed`	integer	Optional	randomly generated seed
Number of samples per stratification	`batch_size`	integer	Optional	`1`
Partitions per variable	`partitions`	list of integers	Optional	No partitioning

As for other SGOPT methods, the random `seed` is used to make stochastic optimizations repeatable. The `batch_size` input specifies the number samples to be evaluated in each multidimensional partition. And the `partitions` list is used to specify the number of partitions for each design variable. For example, `partitions = 2, 4, 3` specifies 2 partitions in the first design variable, 4 partitions in the second design variable, and 3 partitions in the third design variable. This creates a total of 24 multidimensional partitions, and a `batch_size` of 2 would select 2 random samples in each partition, for a total of 48 samples on each iteration of the sMC algorithm. Iterations containing 48 samples will continue until the maximum number of iterations or function evaluations is exceeded, or the desired solution accuracy is obtained.

5.10 Least Squares Methods

DAKOTA's least squares branch currently contains two methods for solving nonlinear least squares problems: NLSSOL, a sequential quadratic programming (SQP) approach that is from the same algorithm family as NPSOL, and Gauss-Newton, which leverages 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.

5.10.1 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.

5.10.2 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 can exhibit quadratic convergence near the solution.

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 5.7 are also applicable for `optpp_g_newton`.

5.11 Nondeterministic Methods

DAKOTA's nondeterministic branch does not currently make use of any method independent controls. As such, the nondeterministic branch documentation which follows is limited to the method dependent controls for the sampling, analytic reliability, and polynomial chaos expansion methods.

5.11.1 Nondeterministic sampling method

The nondeterministic sampling iterator is selected using the `nond_sampling` specification. This iterator performs sampling within specified probability distributions in order to assess the distributions for response functions. Probability of event occurrence (e.g., failure) is then assessed by comparing the response results against response thresholds. DAKOTA currently provides access to nondeterministic sampling methods through the combination of the **NonDSampling** base class and the **NonDLHSSampling** derived class.

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 `response_thresholds` specification supplies a list of thresholds for comparison with the response functions being computed. Statistics on responses above and below these thresholds are then generated.

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. Table 5.18 provides the specification detail for the nondeterministic sampling method.

Table 5.18 Specification detail for nondeterministic sampling method

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
Response thresholds	`response_thresholds`	list of reals	Optional	Vector values = `0.0`

5.11.2 Analytic reliability methods

Analytic reliability methods are selected using the `nond_analytic_reliability` specification. This method computes approximate response function distribution statistics based on specified uncertain variable probability distributions. Analytic reliability methods perform an internal nonlinear optimization to compute a most probable point (MPP) and then integrate about this point to compute probabilities. Supported techniques include the Mean Value method (MV), Advanced Mean Value method (AMV), an iterated form of AMV (AMV+), first order reliability method (FORM), and second order reliability method (SORM), which are selected using the `mv`, `amv`, `iterated_amv`, `form`, and `sorm` specifications, respectively. DAKOTA currently provides access to each of these methods within the **NonDAdvMeanValue** class.

Each of the analytic reliability methods involves a required group specification, separated by OR's. All of the techniques support a `response_levels` specification, which provide the target response values for generating probabilities. In combination, these response level probabilities provide a cumulative distribution function, or CDF, for a response function. The AMV+ method additionally supports a `probability_levels` option, which iterates to find the response level which corresponds to a specified probability (the inverse of the `response_levels` problem). Table 5.19 provides the specification detail for these methods.

Table 5.19 Specification detail for analytic reliability methods

Description	Keyword	Associated Data	Status	Default
Analytic reliability method	`nond_analytic_reliability`	none	Required group	N/A
Method selection	`mv` | `amv` | `iterated_amv` | `form` | `sorm`	none	Required group	N/A
Response levels for probability calculations	`response_levels`	list of reals	Optional (`mv`); Required (`amv`, `iterated_amv`, `form`, `sorm`)	no CDF calculation (`mv`); N/A (`amv`, `iterated_amv`, `form`, `sorm`)
Probability levels for response calculations	`probability_levels`	list of reals	Required (`iterated_amv` only)	N/A

5.11.3 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. The seed, samples, sample_type, and response_thresholds specifications are used to specify settings for internal use of inherited **NonDSampling** techniques. Refer to Nondeterministic sampling method for information on these specifications. Table 5.20 provides the specification detail for the polynomial chaos expansion method.

Table 5.20 Specification detail for polynomial chaos expansion method

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
Number of samples	samples	integer	Optional	minimum required
Sampling type	sample_type	random | lhs	Optional group	lhs
Response thresholds	response_-thresholds	list of reals	Optional	Vector values = 0.0

5.12 Design of Computer Experiments Methods

The Distributed Design and Analysis of Computer Experiments (DDACE) library provides design of experiments methods for computing response data sets at a selection of points in the parameter space. Current techniques include 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 approximation interface), 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 **DACEIterator** class.

The design of experiments methods do not currently make use of any of the method independent controls. 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). Design of experiments specification detail is given in Table 5.21.

Table 5.21 Specification detail for design of experiments methods

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

5.13 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.

5.13.1 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 5.22.

Table 5.22 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
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

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 5.23.

Table 5.23 step_vector 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

5.13.2 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 5.24.

Table 5.24 Specification detail for the list parameter study

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

5.13.3 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 5.25.

Table 5.25 Specification detail for the centered parameter study

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

5.13.4 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 5.26.

Table 5.26 Specification detail for the multidimensional parameter study

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

Chapter 6

Variables Commands

6.1 Variables Description

The variables section in a DAKOTA input file specifies the parameter set to be iterated by a particular method. This parameter set is made up of design, uncertain, and state variables. Design variables can be continuous or discrete and consist of those variables which an optimizer adjusts in order to locate an optimal design. Each of the design parameters can have an initial point, 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, weibull, or histogram. Each uncertain variable specification contains descriptive tags and, either explicitly or implicitly, distribution lower and upper bounds. Distribution lower and upper bounds are explicit portions of the normal, lognormal, uniform, loguniform, and weibull specifications, whereas they are implicitly defined for histogram variables from the extreme values within the bin/point pairs specifications. 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, weibull variables include alpha and beta specifications, and histogram variables include bin pairs and point pairs specifications. 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 `0.0` 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_dist_lower_bounds =  199.3,  474.63         \
          uuv_dist_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., 2001] for discussion on how different iterators view these mixed variable sets.

6.2 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>  \
        <weibull uncertain variables specification>     \
        <histogram 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.

6.3 Variables Set Identifier

The optional set identifier specification uses the keyword `id_variables` to input a unique string for use in identifying a particular variables set. A method can then identify the use of this variables set by specifying the same string in its `variables_pointer` specification (see Method Independent Controls). For example, a method 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 method only if that method 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 method specification(s), since there is no potential for ambiguity in this case. Table 6.1 summarizes the set identifier inputs.

Table 6.1 Specification detail for set identifier

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

6.4 Design Variables

Within the optional continuous design variables specification group, the number of continuous design variables is a required specification and the initial guess, lower bounds, upper bounds, 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 6.2 summarizes the details of the continuous design variable specification and Table 6.3 summarizes the details of the discrete design variable specification.

Table 6.2 Specification detail for continuous design variables

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 = `0.0`
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`
Descriptors	`cdv_descriptors`	list of strings	Optional	Vector of `'cdv_i'` where `i = 1,2,3...`

Table 6.3 Specification detail for discrete 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,...`

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_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.

6.5 Uncertain Variables

Uncertain variables involve one of several supported probability distribution specifications, including normal, lognormal, uniform, loguniform, weibull, or histogram distributions. Each of these specifications is an optional group specification. 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. 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. 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. 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. Within the weibull uncertain optional group specification, the number of weibull uncertain variables and the alpha and beta parameters are required specifications, and the distribution lower and upper bounds and variable descriptors are optional specifications. And finally, 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.

The inclusion of lower and upper distribution bounds for all uncertain variable types (either explicitly or implicitly) 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 6.4 through 6.9 summarize the details of the uncertain variable specifications.

Table 6.4 Specification detail for normal uncertain variables

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_dist_-lower_bounds`	list of reals	Optional	Vector values = `-DBL_MAX`
Distribution upper bounds	`nuv_dist_-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 6.5 Specification detail for lognormal uncertain variables

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_dist_-lower_bounds`	list of reals	Optional	Vector values = `0.0`
Distribution upper bounds	`lnuv_dist_-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 6.6 Specification detail for uniform uncertain variables

Description	**Keyword**	**Associated Data**	**Status**	**Default**
uniform uncertain variables	`uniform_-uncertain`	integer	Optional group	no uniform uncertain variables
Distribution lower bounds	`uuv_dist_-lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`uuv_dist_-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 6.7 Specification detail for loguniform uncertain variables

Description	**Keyword**	**Associated Data**	**Status**	**Default**
loguniform uncertain variables	`loguniform_-uncertain`	integer	Optional group	no loguniform uncertain variables
Distribution lower bounds	`luuv_dist_-lower_bounds`	list of reals	Required	N/A
Distribution upper bounds	`luuv_dist_-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 6.8 Specification detail for weibull uncertain variables

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
Distribution lower bounds	`wuv_dist_-lower_bounds`	list of reals	Optional	Vector values = `-DBL_MAX`
Distribution upper bounds	`wuv_dist_-upper_bounds`	list of reals	Optional	Vector values = `+DBL_MAX`
Descriptors	`wuv_-descriptors`	list of strings	Optional	Vector of `'wuv_i'` where `i = 1,2,3,...`

Table 6.9 Specification detail for histogram uncertain variables

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,...`

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).

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 an analytic reliability (i.e., `nond_analytic_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 have n^2 entries listed by rows where n is the total number of uncertain variables (all normal, lognormal, uniform, loguniform, weibull, and histogram specifications, in that order). Table 6.10 summarizes the specification details:

Table 6.10 Specification detail for uncertain correlations

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

6.6 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 6.11 summarizes the details of the continuous state variable specification and Table 6.12 summarizes the details of the discrete state variable specification.

Table 6.11 Specification detail for continuous state variables

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 = `0.0`
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 6.12 Specification detail for discrete 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,...`

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 parameterized 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 7

Interface Commands

7.1 Interface Description

The interface section in a DAKOTA input file specifies how function evaluations will be performed. Function evaluations can be performed using either an interface with a simulation code or an interface with an approximation method.

In the former case of a simulation, the application interface is used to invoke the simulation with either system calls, forks, direct function invocations, or computational grid invocations. In the system call and fork cases, communication between DAKOTA and the simulation occurs through parameter and response files. In the direct function case, communication occurs through the function parameter list. The direct case can involve linked simulation codes or analytic test functions which are compiled into the DAKOTA executable. The analytic test functions allow for rapid testing of algorithms without process creation overhead or engineering simulation expense. The grid case is experimental and under development.

In the case of an approximation, an approximation interface can be selected to make use of the global, local, multipoint, and hierarchical surrogate modeling capabilities available within DAKOTA's **ApproximationInterface** class and **DakotaApproximation** class hierarchy.

Several examples follow. The first example shows an application interface specification which specifies the use of system calls, 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 Application Interface for more information on the use of these options.

```
interface,                                              \
        application 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.

```
interface,                                              \
        application direct                              \
          analysis_drivers = 'rosenbrock'
```

The final example shows an approximation interface specification which selects a quadratic polynomial approximation from among the global approximation methods. It uses a pointer to a design of experiments method for generating the data needed for building a 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.

```
interface,                                              \
        approximation global                            \
          quadratic polynomial                          \
          dace_method_pointer = 'DACE'                  \
          reuse_samples region                          \
          correction multiplicative first_order
```

Additional information on interfacing with simulations and approximations is provided in the following sections.

7.2 Interface Specification

The interface specification has the following top-level structure:

```
interface,                                              \
        <set identifier>                                \
        <application specification> OR                  \
        <approximation specification>
```

where the set identifier is an optional specification and either an application or approximation interface must be specified. If an application interface is specified, its type must be system, fork, direct, or grid, i.e.:

```
interface,                                              \
        <set identifier>                                \
        application                                     \
          <system call specification>     OR            \
          <fork specification>            OR            \
          <direct function specification> OR
          <grid specification>
```

If an approximation interface is specified, its type must be global, multipoint, local, or hierarchical, i.e.:

```
interface,                                              \
        <set identifier>                                \
        approximation                                   \
          <global specification>     OR                 \
          <multipoint specification> OR                 \
          <local specification>      OR                 \
          <hierarchical specification>
```

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

7.3 Interface Set Identifier

The optional set identifier specification uses the keyword `id_interface` to input a string for use in identifying a particular interface specification. A method can then identify the use of this interface by specifying the same string in its `interface_pointer` specification (see Method Independent Controls). For example, a method 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 method only if that method 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 method specification(s), since there is no potential for ambiguity in this case. Table 7.1 summarizes the set identifier inputs.

Table 7.1 Specification detail for set identifier

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Interface set identifier	`id_interface`	string	Optional	use of last interface parsed

7.4 Application Interface

The application interface uses a simulator program, 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, 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, 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 will utilize the Condor and/or Globus services and is experimental and incomplete.

The application interface group specification contains several specifications which are valid for all application interfaces as well as additional specifications pertaining specifically to system call, fork, direct, or grid application interfaces. Table 7.2 summarizes the specifications valid for all application interfaces, and Tables 7.3, 7.4, 7.5, and 7.6 summarize the additional specifications for system call, fork, direct, and grid application interfaces, respectively.

Table 7.2 Specification detail for application interfaces

Description	Keyword	Associated Data	Status	Default
Application interface	`application`	none	Required group (1 of 2 selections)	N/A
Analysis drivers	`analysis_-drivers`	list of strings	Required	N/A
Input filter	`input_filter`	string	Optional	no input filter
Output filter	`output_-filter`	string	Optional	no output filter
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
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

In Table 7.2, 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 `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.

The optional `asynchronous` flag specifies use of asynchronous protocols (i.e., background system calls, nonblocking 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.

Failure capturing in application 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 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 the general application interface specifications, the type of application interface involves a selection between `system`, `fork`, `direct`, or `grid` required group specifications. The following sections describe these group specifications in detail.

7.4.1 System call application 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. 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 application interfaces are summarized in Table 7.3.

Table 7.3 Additional specifications for system call application interfaces

Description	Keyword	Associated Data	Status	Default
System call application 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

7.4.2 Fork application interface

For fork application 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 application 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 application interfaces are summarized in Table 7.4.

Table 7.4 Additional specifications for fork application interfaces

Description	Keyword	Associated Data	Status	Default
Fork application 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

7.4.3 Direct function application interface

For direct function application interfaces, `processors_per_analysis` and `modelcenter_file` are additional optional settings within the required group which can be used to specify multiprocessor analysis partitions and the configuration filename for a ModelCenter simulation, respectively. As with the `evaluation_servers`, `analysis_servers`, `evaluation_self_scheduling`, `evaluation_static_scheduling`, `analysis_self_scheduling`, and `analysis_static_scheduling` specifications described above in Application Interface, `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. DAKOTA supports a direct interface to 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 `modelcenter_file` specification provides the means to communicate this configuration file to DAKOTA. The direct application interface specifications are summarized in Table 7.5.

Table 7.5 Additional specifications for direct function application interfaces

Description	Keyword	Associated Data	Status	Default
Direct function application 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
Configuration file for ModelCenter simulation	`modelcenter_file`	string	Optional (required for direct ModelCenter interface)	direct interface to ModelCenter not used

In addition to ModelCenter, a direct interface to Sandia's SALINAS structural dynamics code is available and a direct interface to Sandia's SIERRA multiphysics framework is scheduled to be supported in future releases. In addition to interfaces with simulation codes, a common usage of the direct interface is for invoking internal test problems 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 application interface class and are used for algorithm testing. Refer to **DirectFnApplicInterface** for currently available testers.

7.4.4 Grid application interface

For grid application interfaces, `hostnames` and `processors_per_host` are additional settings within the required group. The `hostnames` specification provides a list of machines for use in distributing evaluations, and the `processors_per_host` specification provides the number of processors to use from each host. This capability is a placeholder for future work with Condor and/or Globus services and is not currently operational. The additional specifications for grid application interfaces are summarized in Table 7.6.

Table 7.6 Additional specifications for grid application interfaces

Description	Keyword	Associated Data	Status	Default
Grid application interface	grid	none	Required group (1 of 4 selections)	N/A
Names of host machines	hostnames	list of strings	Required	N/A
Number of processors per host	processors_-per_host	list of integers	Optional	1 processor from each host

7.5 Approximation Interface

The approximation interface uses an approximate representation of a ”truth” model to perform the parameter to response mappings. This approximation, or surrogate model, 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. Approximation interfaces 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 approximation interfaces 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 approximation interface specification requires the specification of one of the following approximation types: `global`, `multipoint`, `local`, or `hierarchical`. Each of these specifications is a required group with several additional specifications. The following sections present each of these specification groups in further detail.

7.5.1 Global approximation interface

The global approximation interface specification requires the specification of one of the following approximation methods: `neural_network`, `polynomial`, `mars`, `hermite`, 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, 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. For each of the global approximation methods, `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, or matching both truth values and truth gradients in the case of first-order matching. The truth data is matched at a single point, typically the center of the approximation region. Available techniques include `additive zeroth_order` for adding a scalar offset to the approximation to match a truth value at a point, `multiplicative zeroth_order` for multiplying the approximation by a scalar to match a truth value at a point, `additive first_order` for adding a linear function to match the truth value and the truth gradient at a point, and `multiplicative first_order` for multiplying the approximation by a linear function to match the truth value and the truth gradient at a point. The `additive first_order` case is due to [Lewis and Nash, 2000] and the `multiplicative first_order` case is also known as beta correction [Haftka, 1991]. 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 **DakotaApproximation::build**(), but is not yet supported in any global approximation derived class redefinitions of **DakotaApproximation::find_coefficients**(). Table 7.7 summarizes the global approximation interface specifications.

Table 7.7 Specification detail for global approximation interfaces

Description	Keyword	Associated Data	Status	Default
Global approximation interface	`global`	none	Required group (1 of 4 selections)	N/A
Artificial neural network	`neural_-network`	none	Required (1 of 5 selections)	N/A
Polynomial	`polynomial`	`linear` | `quadratic` | `cubic`	Required (1 of 5 selections)	N/A
Multivariate adaptive regression splines	`mars`	none	Required (1 of 5 selections)	N/A
Hermite polynomial	`hermite`	none	Required (1 of 5 selections)	N/A
Kriging interpolation	`kriging`	none	Required group (1 of 5 selections)	N/A
Kriging correlations	`correlations`	list of reals	Optional	internally computed correlations
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 `multiplica-tive`, `zeroth_order` or `first_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

7.5.2 Multipoint approximation interface

Multipoint approximations use data from previous design points to improve the accuracy of local approximations. This specification is a placeholder for future capability as no multipoint approximation algorithms are currently available. Table 7.8 summarizes the multipoint approximation interface specifications.

Table 7.8 Specification detail for multipoint approximation interfaces

Description	Keyword	Associated Data	Status	Default
Multipoint approximation interface	`multipoint`	none	Required group (1 of 4 selections)	N/A
Pointer to the truth interface specification	`actual_-interface_-pointer`	string	Required	N/A

7.5.3 Local approximation interface

Local approximations use value and gradient 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. This is a first order Taylor series expansion, also known as the "linear approximation" in the optimization literature. Other local approximations, such as the "reciprocal" and "conservative/convex" approximations, may become available in the future. The required `actual_interface_pointer` specification and the optional `actual_interface_responses_pointer` specification are the additional inputs for local approximations. The former points to an interface specification which provides the truth model for generating the value and gradient data used in the series expansion. And the latter can be used to employ a different responses specification for the truth model than that used for mappings from the local approximation. For example, the truth model may generate gradient data using finite differences (as specified in the responses specification identified by `actual_interface_responses_pointer`), whereas the local approximation may return (approximate) analytic gradients (as specified in a different responses specification which is identified by the method using the local approximation as its interface). If `actual_interface_responses_pointer` is not specified, then the response set available from truth model evaluations and approximation interface mappings will be the same. Table 7.9 summarizes the local approximation interface specifications.

Table 7.9 Specification detail for local approximation interfaces

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Local approximation interface	`local`	none	Required group (1 of 4 selections)	N/A
Taylor series local approximation	`taylor_series`	none	Required	N/A
Pointer to the truth interface specification	`actual_interface_pointer`	string	Required	N/A
Pointer to the truth responses specification	`actual_interface_responses_pointer`	string	Optional	reuse of responses specification in truth model

7.5.4 Hierarchical approximation interface

Hierarchical approximations use corrected results from a low fidelity interface as an approximation to the results of a high fidelity "truth" model. The required `low_fidelity_interface_pointer` specification points to the low fidelity interface specification. This interface is used to generate low fidelity responses which are then corrected and returned to an iterator. The required `high_fidelity_interface_pointer` specification points to the interface specification for the high fidelity truth model. This model is used only when new correction factors for the low fidelity interface are needed. The `correction` specification specifies which correction technique will be applied to the low fidelity results in order to match the high fidelity results (value or both value and gradient) at a particular point (e.g., the center of the approximation region). In the hierarchical case (as compared to the global case), the `correction` specification is required, since the omission of a correction technique would effectively waste all high fidelity evaluations. If it is desired to use a low fidelity model without corrections, then a hierarchical approximation is not needed and a single application interface should be used. Available correction techniques are `additive zeroth_order`, `multiplicative zeroth_order`, `additive first_order`, and `multiplicative first_order`, as described previously in Global approximation interface. Table 7.10 summarizes the hierarchical approximation interface specifications.

Table 7.10 Specification detail for hierarchical approximation interfaces

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

Chapter 8

Responses Commands

8.1 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 **DakotaResponse** 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` or `analytic_hessians` where the meanings are the same as for the corresponding gradient availability settings. Numerical Hessians are not currently supported, since, in the case of optimization, this would imply a finite difference-Newton technique for which a direct algorithm already exists. Capability for numerical Hessians can be added in the future if the need arises.

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_step_size = .001                        \
        no_hessians
```

The last example shows a specification that could be used with a nondeterministic 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.

8.2 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 two types:

- no Hessians
- analytic Hessians

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

8.3 Responses Set Identifier

The optional set identifier specification uses the keyword `id_responses` to input a string for use in identifying a particular responses specification. A method can then identify the use of this response set by specifying the same string in its `responses_pointer` specification (see Method Independent Controls). For example, a method 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 method only if that method 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 method specification(s), since there is no potential for ambiguity in this case. Table 8.1 summarizes the set identifier input.

Table 8.1 Specification detail for set identifier

Description	Keyword	Associated Data	Status	Default
Responses set identifier	`id_responses`	String	Optional	use of last responses parsed

8.4 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 8.2 summarizes the response descriptors input.

Table 8.2 Specification detail for response labels

Description	Keyword	Associated Data	Status	Default
Response labels	`response_-descriptors`	list of strings	Optional	root strings plus numeric identifiers

8.5 Function Specification

The function specification must be one of three types: 1) a group containing objective and constraint functions, 2) a group containing least squares terms and constraint functions, or 3) a response functions specification. These function sets correspond to optimization, least squares, and uncertainty quantification iterators, respectively. Parameter study and design of experiments iterators may be used with any of the three function specifications.

8.5.1 Objective and constraint functions (optimization data set)

An optimization data set is specified using `num_objective_functions` and optionally `multi_-objective_weights`, `num_nonlinear_inequality_constraints`, `nonlinear_-inequality_lower_bounds`, `nonlinear_inequality_upper_bounds`, `num_nonlinear_-equality_constraints`, and `nonlinear_equality_targets`. 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. 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:

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

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 +bigBoundSize (1.e+30, as defined in **DakotaOptimizer**) are treated as +infinity and any lower bound values less than -bigBoundSize 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 < -bigBoundSize). The same approach is used for the linear inequality bounds as described in Method Independent Controls.

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 `0.0` for each constraint

$$g(x) = 0.0$$

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 8.3 summarizes the optimization data set specification.

Table 8.3 Specification detail for optimization data sets

Description	Keyword	Associated Data	Status	Default
Number of objective functions	`num_objective_functions`	integer	Required group	N/A
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 = `0.0`
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 = `0.0`

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

A least squares data set is specified using `num_least_squares_terms` and optionally `num_nonlinear_inequality_constraints`, `nonlinear_inequality_lower_bounds`, `nonlinear_inequality_upper_bounds`, `num_nonlinear_equality_constraints`, and `nonlinear_equality_targets`. 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. Table 8.4 summarizes the least squares data set specification.

Table 8.4 Specification detail for nonlinear least squares data sets

Description	**Keyword**	**Associated Data**	**Status**	**Default**
Number of least squares terms	`num_least_squares_terms`	integer	Required	N/A
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 = `0.0`
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 = `0.0`

8.5.3 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 8.5 summarizes the generic response function data set specification.

Table 8.5 Specification detail for generic response function data sets

Description	Keyword	Associated Data	Status	Default
Number of response functions	num_-response_-functions	integer	Required	N/A

8.6 Gradient Specification

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

8.6.1 No gradients

The no_gradients specification means that gradient information is not needed in the study. Therefore, it will neither be retrieved from the simulation nor computed with finite differences. The no_gradients keyword is a complete specification for this case.

8.6.2 Numerical gradients

The numerical_gradients specification means that gradient information is needed and will be computed with finite differences using either the native or one of the vendor finite differencing routines.

The method_source setting specifies the source of the finite differencing routine that will be used to compute the numerical gradients: dakota denotes DAKOTA's internal finite differencing algorithm and vendor denotes the finite differencing algorithm supplied by the iterator package in use (DOT, CONMIN, NPSOL, 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 vendor routine supports 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).

Lastly, fd_step_size specifies the relative finite difference step size to be used in the computations. For DAKOTA, DOT, CONMIN, and OPT++, the intervals are computed by multiplying the fd_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_step_size` as their minimum absolute differencing interval. With a `fd_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_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 8.6 summarizes the numerical gradient specification.

Table 8.6 Specification detail for numerical gradients

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_step_size`	real	Optional	`0.001`

8.6.3 Analytic gradients

The `analytic_gradients` specification means that gradient information is available directly from the simulation (finite differencing is not required). The simulation must return the gradient data in the DAKOTA format (enclosed in single brackets; see DAKOTA File Data Formats in the Users Manual) for the case of file transfer of data. The `analytic_gradients` keyword is a complete specification for this case.

8.6.4 Mixed gradients

The `mixed_gradients` specification means that some gradient information is available directly from the simulation (analytic) whereas the rest will have to be finite differenced (numerical). This specification allows the user to make use of as much analytic gradient information as is available and then to 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` list specifies by number the functions which have analytic gradients, and the `id_numerical` 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` and `id_numerical` lists. The `method_source`, `interval_type`, and `fd_step_size` specifications are as described previously in Numerical gradients and pertain to those functions listed by the `id_numerical` list. Table 8.7 summarizes the mixed gradient specification.

Table 8.7 Specification detail for mixed gradients

Description	Keyword	Associated Data	Status	Default
Mixed gradients	`mixed_gradients`	none	Required group	N/A
Analytic derivatives function list	`id_analytic`	list of integers	Required	N/A
Numerical derivatives function list	`id_numerical`	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_step_size`	real	Optional	`0.001`

8.7 Hessian Specification

Hessian availability must be specified with either `no_hessians` or `analytic_hessians`. Numerical Hessians are not currently supported, since, in the case of optimization, this would imply a finite difference-Newton technique for which a direct algorithm already exists. Capability for numerical Hessians can be added in the future if the need arises.

8.7.1 No Hessians

The `no_hessians` specification means that the method does not require Hessian information. Therefore, it will neither be retrieved from the simulation nor computed through other means. The `no_hessians` keyword is a complete specification for this case.

8.7.2 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.

Chapter 9

References

- 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).

- 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.

- 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., Giunta, A. A., van Bloemen Waanders, B. G., Wojtkiewicz, S. F., Jr., Hart, W. E., and Alleva, M. P., 2001. ”DAKOTA: A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 3.0 Users Manual,” Sandia Technical Report SAND2001-3796.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.