Plato 2.12 Input Deck Reference Manual

SAND2023-10961O

Unlimited Release

Plato Development Team
Sandia National Labs [1]

October 16, 2023

[1]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.


plato
OPTIMIZATION-BASED DESIGN

Contents

1 | Introduction

This document describes the Plato input deck. The Plato input deck contains the "recipe" for running a Plato optimization problem. A Plato problem is either a topology optimization problem or a shape/CAD parameter optimization problem. In these problems we are trying to optimize the topology or CAD parameters such that the resulting design meets some user-defined perfromance objective. These types of problems require a geometric definition of the design domain as well as instructions about what the objectives and constraints are, what physics codes will be used to evaluate the objectives and constraints, what optimization algorithm to use, and various other parameters. The geoemtry is provided in the form of a finite element mesh, and all of the other instructions are contained in the Plato input deck. This document will define general concepts used in the setup of a Plato optimization problem and will then define all of the possible options that can be included in the input deck.

2 | General Concepts

Optimization problems can be very complex and require lots of different information. The Plato input deck is designed to be very general in the way it defines an optimization problem, breaking it down into its fundamental pieces that can be combined in different ways to define different kinds of problems. There are five main building blocks upon which all optimization problems are defined in Plato. This section will briefly introduce these building blocks and describe how they are used to define an optimization problem.

2.1 Service

Services are the software executables that will be performing the work during the optimization problem. A typical Plato run always uses at least two different services. There will be a PlatoMain service that contains the optimizer and other utilities like filtering. In some cases this service may also be used to evaluate criterion values that will be used in a constraint (for example, volume). Then there will typically be one or more physics services, which will calculate the physics-based criteria that will be used in objectives and constraints. The services are defined independently in the input deck so that they can be defined once and then referenced in a general way by other items in the input deck that need to use them.

2.2 Criterion

A criterion is a quantity of interest that will be calculated at each iteration as part of evaluating an objective or a constraint. For example, if the objective were to minimize the compliance of a structure (make it stiffer), the criterion would be some measure of the compliance of the structure. If the objective were to match a set of user-supplied mass properties, the criterion would be the mismatch between the current iteration's mass properties and those provided by the user. Criteria are also used in constraints. A volume constraint would be specified in terms of a volume criterion. In the input deck, criteria are defined independently of the objectives and constraints so that they can be defined only once and then referenced repeatedly as needed by different objectives or constraints.

2.3 Scenario

A scenario is a description of the physics problem being solved during the calculation of an objective or constraint. It includes the type of physics as well as the loads and boundary conditions describing the physical problem. Plato supports problems with multiple objectives, multiple physics, multiple load cases, etc. Scenarios provide a general way for defining these types of problems. As with services and criteria, scenarios are defined independently in the input deck so that they can be defined once and then referenced by other items in mulitple ways if needed.

2.4 Objective

An objective defines what is trying to be achieved in the optimization problem. It is made up of the criterion being measured, the service providing the evaluation, and the physical scenario under which the objective value is being considered. The input deck only defines one objective, but the objective can be made up of multiple sub-objectives – each with its own criterion, service, and scenario. The sub-objectives can be weighted based on their importance in the problem.

2.5 Constraint

A constraint defines limitations on quantities of interest in the optimization problem. For example, in a stress-constrained mass minimization problem the user defines a maximum stress that any point in the design can experience. In a compliance minimization problem the user will typically put a constraint on the volume or mass of the final design. Similar to an objective, a constraint is made up of the criterion being measured, the service providing the evaluation, and the physical scenario under which the constraint value is being measured. Multi-constraint problems can be run in Plato, but this feature is new and considered a beta-level capability.

3 | Input Deck Options

This section will define all of the possible entries in the input deck and their corresponding syntax.

3.1 Syntax

The input deck is organized into different groupings of commands called blocks. Five of the types of blocks (criterion, scenario, service, objective, constraint) were described in the "General Concepts" section. Each block will contain one or more lines containing keyword/value pairs. For example, "loads 1 5 6" is a keyword-value pair, where the keyword is "loads" and the value is made up of the three load ids "1 5 6". Table 3.1 shows the different types of values and a description of each.

Table 3.1: Description of value types.

Value Type	**Description**
Boolean	Data that can only take on the value true or false.
integer	Data that can only take on integer values. Example: 4 10 50.
value	Data that can only take on real or floating point values. Example: 4.33.
string	Data that can only take on string values. Example: stress.

For each input deck parameter described in the following sections we use a variation of the syntax "parameter {integer}" to show the syntax for that parameter. Here are some examples:

number_processors {integer}: Indicates that the user should enter a single integer value for this parameter–"number_processors 16".
loads {integer} {...}: Indicates that the user should specify one or more integer values for this parameter separated by spaces–"loads 1 2 3".

prune_mesh {Boolean}: Indicates that the user should specify "true" or "false" for this parameter–"prune_mesh true".
type {string}: Indicates that the user should specify a string for this parameter– "type volume".
load_case_weights {value} {...}: Indicates that the user should specify one or more real values–"load_case_weights 0.1 0.4 0.5".

Comments within an input deck can be specified as lines beginning with "//". For that line, all symbols after the "//" will be ignored by the input deck parsing.

Figure 3.1 shows a portion of a simple Plato input deck example.

3.2 Service

In this section, we show how to specify service blocks. Each service block begins and ends with the tokens "begin service {integer}" and "end service." The integer following "begin service" specifies the identification index of this service. Other blocks in the input deck will use this value to reference the service. The following is a typical service block definition:

```
begin service 3
   code sierra_sd
   number_processors 16
end service
```

Plato input decks can contain one or more service blocks, and the first serivce of every input deck must be a service with "code platomain" and must have id 1. The first service has the optimizer and is the one that orchestrates the execution of the optimization run. The Plato input deck templates from which new plato problems are defined will always have the first service defined as the platomain service. The following tokens can be specified in any order within the service block.

3.2.1 code

Each service must specify the code (software executable) that will be providing the service in the format: "code {string}." Current options include "platomain," "sierra_sd," "sierra_tf," "plato_analyze," and "plato_esp." "plato_esp" services are only used for shape/CAD parameter optimization problems to provide sensitivities with respect to CAD parameters.

3.2.2 number_processors

Each service must specify the number of processors the service will be run on using the format: "number_processors {integer}". For GPU runs using Plato Analyze you will typically

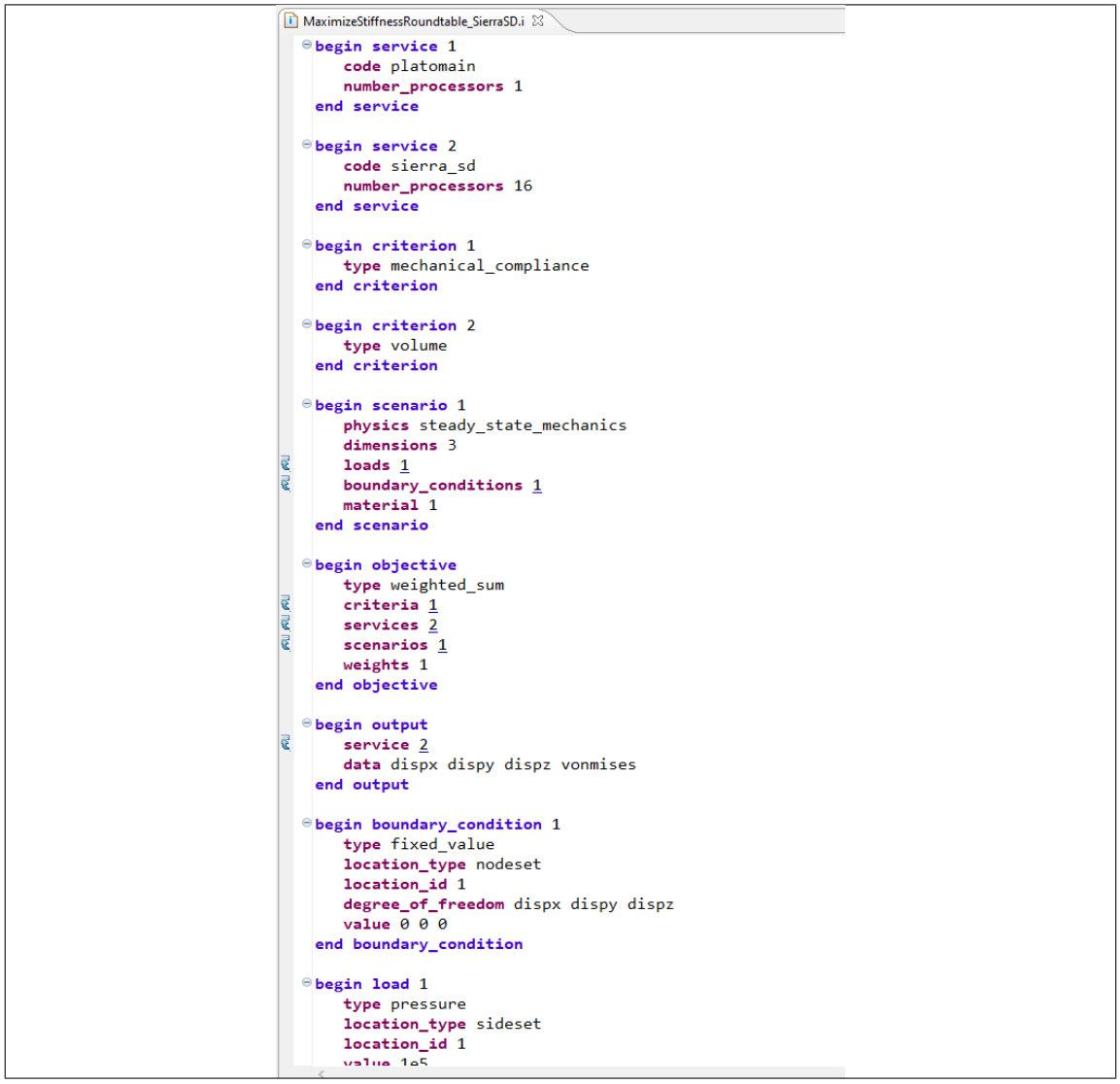

Figure 3.1: Example Plato input deck.

only specify one processor.

3.2.3 device_ids

When running on GPUs with Plato Analyze, you can specify which GPU (device) to use if the machine you are running on has more than one GPU available. The device is specified using the format: "device_ids {integer} {...}."

3.2.4 cache_state

Each service can specify whether it uses the "cache_state" mechanism during the optimization run using the format "cache_state {Boolean}." For efficiency, services can utilize a caching mechanism that reduces the need for recomputing state variables in the physics problem. Currently, the SierraSD code requires the use of the cache_state mechanism, and so should always include "cache_state true" in its service block.

3.2.5 update_problem

Each service can specify whether it uses the "update_problem" mechanism during the optimization run using the format "update_problem {Boolean}." Some optimization problems (such as stress-constrained mass minimization) require the physics code to update local state information at certain frequencies. The "update_problem" flag specifies whether the optimizer will call the update operation for this service.

3.2.6 path

Each service can specify the path to its corresponding executable if needed using the syntax "path {string}." Normally, this is not required, as Plato will know where to find the available services.

3.3 Criterion

In this section, we show how to specify criterion blocks. Each criterion block begins and ends with the tokens "begin criterion {integer}" and "end criterion". The integer following "begin criterion" specifies the identification index of this criterion. Other blocks in the input deck will use this value to reference the criterion. The following is a typical criterion block definition:

```
begin criterion 1
   type mechanical_compliance
end criterion
```

Plato input decks can contain one or more criterion blocks. The following tokens can be specified in any order within the criterion block.

3.3.1 type

Each criterion must have a type specified in the format: "type {string}". Table 3.2 lists a description of the allowable types and what physics code they can be used with. In the table "SD" is sierra_sd, "TF" is sierra_tf, and "PA" is plato_analyze.

3.3.2 criterion_ids

When defining a composite criterion, this parameter defines the criteria that make up the composite criterion. The syntax for this parameter is: "criterion_ids {integer} {...}". The integer values are the ids of criteria making up the composite criterion.

3.3.3 criterion_weights

When defining a composite criterion, this parameter defines the weights of the criteria that make up the composite criterion. The syntax for this parameter is: "criterion_weights {value} {...}".

3.3.4 location_name

When defining a criterion evaluated on a surface or body, this parameter defines the names of the exodus entity sets (e.g. sideset, nodeset, element blocks) where the criterion will be evaluated. The syntax for this parameter is: "location_name {string} {...}". The string values are the names of the exodus entity sets that make up the surfaces or element blocks of interests. For most criteria there will be only one string following the location_name keyword. However, for some types of criteria this option may contain multiple values specified by spaces. An example of this would be criteria related to CFD applications where multiple sidesets are specified for either inlets or outlets.

3.3.5 location_type

When defining a criterion evaluated on an entity set, i.e. "location_name" is set, one can specify the application location type using the format: "location_type {string} {...}". Current options include "sideset", "nodeset", and "element_block". For most criteria there will be only one string following the location_type keyword. However, for some types of criteria this option may contain multiple values specified by spaces, e.g., criteria related to CFD applications where multiple sidesets are specified for either inlets or outlets.

3.3.6 Parameters for Matching Mass Properties

The following mass properties can be specified by the user, and Plato will attempt to generate a design that has matching mass properties, either by using this criterion, "type mass_properties," in a constraint or as part of the objective. When using it in the objective, the user may need to modify the "weight" parameter in the objective to strengthen or weaken the enforcement of the mass property matching. If used as a constraint in a problem that already has a constraint, you should use the ROL optimization algorithms, which handle multiple constraints in a single problem. The following is a typical criterion block specifying mass properties to be matched:

Table 3.2: Description of supported criterion types and applicable physics codes.

Criterion Type	**Description**	**Valid Code**
composite	A criterion that is a combination of multiple sub-criteria. This type is currently only supported in the Plato Analyze service and is only used when you need a single Plato Analyze performer to evaluate multiple sub-criteria and combine them as a weighted sum before returning the value to the optimizer. All the sub-criteria must use the same Scenario definition. A composite criterion includes the ids and weights of the sub-criteria that make it up.	PA
displacement	A measure of the displacement in a given direction. This criterion can be used to constrain the displacement to be a given value.	PA
mass_properties	A measure of the mismatch between user-specified mass properties and those of the current design. Minimizing this mismatch will force the optimized design to have the same mass properties as those specified by the user.	PA
mechanical_compliance	A measure of the stiffness of the structure. Minimizing this measure will make the structure stiffer.	SD, PA
thermal_compliance	A measure of the resistance to heat conduction. Minimizing this measure will maximize heat conduction.	PA
stress_and_mass	Used for doing stress-constrained mass minimization problems.	SD, PA
volume_average_von_mises	Computes the volume-averaged von Mises stress.	SD
stress_p-norm	Superscript p used to compute the norm of the stress.	PA
volume	The volume of the current design.	PA
flux_p-norm	Superscript p used to compute the norm of the heat flux.	PA
modal_projection_error	A measure of the mismatch between the modes of the current design iteration and a user-specified set of modes.	SD
temperature_matching	A measure of the mismatch between the temperature values at specified locations of the current design iteration and a user-specified set of temperature values.	TF
mean_temperature	A measure of the fluid mean temperature.	PA
mean_surface_temperature	A measure of the mean temperature on a surface. This criterion is only available for computational fluid dynamics applications.	PA
mean_surface_pressure	A measure of the mean pressure on a surface. This criterion is only available for computational fluid dynamics applications.	PA

```
begin criterion 3
   type mass_properties
   cgx 0.5 weight 1.0
   cgy 0.75 weight 1.5
end criterion
```

Note: If only a subset of the mass properties listed below are requested, the user must ensure that the coordinate system used to generate target values is the same as the one used during optimization.

3.3.6.1 mass

This is the mass of the object. Syntax: "mass {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.2 cgx

This is the X component of the center of gravity. Syntax: "cgx {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.3 cgy

This is the Y component of the center of gravity. Syntax: "cgy {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.4 cgz

This is the Z component of the center of gravity. Syntax: "cgz {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.5 ixx

This is the mass moment of inertia about the x axis. Syntax: "ixx {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.6 iyy

This is the mass moment of inertia about the y axis. Syntax: "iyy {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.7 izz

This is the mass moment of inertia about the z axis. Syntax: "izz {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.8 ixy

This is the XY product of inertia. Syntax: "ixy {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.9 iyz

This is the YZ product of inertia. Syntax: "iyz {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.6.10 ixz

This is the XZ product of inertia. Syntax: "ixz {value} weight {value}". The value is the value you want the final design to have. The weight can be used to weight different mass properties differently when multiple are specified.

3.3.7 Parameters for Displacement Constraints

The displacement at a specific location on the design can be constrained using the criterion with "type displacement". The location is currently defined using a sideset in the mesh and is specified with respect to a direction. The aspect of displacement being measured in the criterion can be specified using the various options below. The measure of the displacement can be used in a constraint to constrain what the displacement should be at that location by using "absolute_target" in the constraint rather than "relative_target". Here is an example displacement criterion. The various options are described below it.

```
begin criterion 4
   type displacement
   displacement_direction 0 1 0
```

```
   measure_magnitude true
   location_type sideset
   location_name ss4
end criterion
```

3.3.7.1 displacement_direction

The direction in which the displacement will be measured. Syntax: "displacement_direction {value} {value} {value}". The 3 values are the three coordinates of the direction. Whether you normalize this vector depends on the type of displacement measured but in general it is good practice to just always normalize this vector.

3.3.7.2 measure_magnitude

Whether to return a signed displacement magnitude or not. Syntax: "measure_magnitude {Boolean}. If "true" is specified the returned value will be the absolute value of the displacement. If "false" is specified the signed displacement will be returned.

3.3.7.3 target_magnitude

This option allows the user to specify a non-zero target for the displacement magnitude. The value the criterion returns will be the difference between the actual displacement magnitude in the direction specified and the target magnitude specified by the user in that same direction. When using this option the "measure_magnitude" option is ignored. Syntax: "target_magnitude {value}.

3.3.7.4 target_solution_vector

This option allows the user to specify a desired displacement vector. The criterion will return the magnitude of the difference vector between the actual displacement vector and the target displacement vector. When this option is used "measure_magnitude" and "displacement_direction" are ignored. Syntax: "target_solution_vector {value} {value} {value}" where the 3 values are the components of the target displacement vector.

3.3.7.5 location_type

The type of location (sideset or nodeset). Syntax: "location_type {string}. Currently, only sideset can be specified for this parameter.

3.3.7.6 location_name

The name of the location sideset or nodeset (only sidesets are allowed at this time). Syntax: "location_name {string}.

3.3.8 Parameters for Stress-Constrained Mass Minimization

The stress-constrained mass minimization problem formulation ("type stress_and_mass") includes an augmented Lagrangian (AL) enforcement of local stress constraints as part of the objective evaluation. As such, there are various AL parameters that can be set as part of the "stress_and_mass" criterion. This section describes these parameters.

3.3.8.1 stress_limit

The value of the VonMises stress under which the optimizer should try to constrain all points in the design. Syntax: "stress_limit {value}".

3.3.8.2 scmm_initial_penalty

The initial value of the penalization scalar used to enforce the local stress constraint. Syntax: "scmm_initial_penalty {value}".

3.3.8.3 scmm_penalty_expansion_multiplier

The amount to "grow" the stress constraint penalty by each time it is updated. Syntax: "scmm_penalty_expansion_multiplier {value}".

3.3.8.4 scmm_constraint_exponent

The power that the stress constraint term in the formulation will be raised to. Syntax: "scmm_constraint_exponent {value}". A typical value is 2 or 3.

3.3.8.5 scmm_penalty_upper_bound

The maximum value the stress constraint penalty can grow to. Syntax: "scmm_penalty_upper_bound {value}". **Note: this parameter is only applicable when using Plato Analyze.**

3.3.9 Parameters for Mode Matching

The mode matching problems ("type modal_projection_error") require a number of extra parameters to define the modes to match and locations at which to match them.

3.3.9.1 num_modes_compute

The number of modes to compute in the modal analysis calculation. For best results this value should be significantly larger than the number of modes being matched. For example, if you wanted to match the first three non-rigid-body modes, you should set this value to 20 for good convergence. Syntax: "num_modes_compute {integer}".

3.3.9.2 modes_to_exclude

The modes out of the total number computed that we do not want to be included in the mode-matching objective. Syntax: "modes_to_exclude" {integer} {...}". For example, if the user wants to match the first three non-rigid-body modes (modes 7, 8, and 9), and he specified 20 modes to be computed, he would use specify "modes_to_exclude 1 2 3 4 5 6 10 11 12 13 14 15 16 17 18 19 20".

3.3.9.3 match_nodesets

The nodesets on the mesh at which we want to match the modes. Syntax: "match_nodesets {integer} {...}". These will be defined on the "reference" mesh which is the mesh that has the modes we are trying to match (see "ref_data_file"). Each nodeset should contain only a single node.

3.3.9.4 ref_data_file

The mesh file that contains the modal information we are trying to match in the new design. Syntax: "ref_data_file {string}". It contains the nodesets at which we are trying to match modes (see "match_nodesets") and the modal information obtained by running a modal analysis on some geometry that has the modes we want to match.

3.3.9.5 eigen_solver_shift

An eigen solver parameter that should usually be set to a large negative value (-5e7) to help solver convergence. Syntax: "eigen_solver_shift {value}".

3.3.9.6 camp_solver_tol

An eigen solver parameter that can just be set to 1e-4 as a default. Syntax: "camp_solver_tol {value}."

3.3.9.7 camp_max_iter

An eigen solver parameter that can just be set to 5000 as a default. Syntax: "camp_max_iter {integer}."

3.3.10 Parameters for Matching Temperature Matching

The temperature matching problems ("type temperature_matching") require a number of extra parameters to define the temperatures to match and locations at which to match them.

3.3.10.1 match_nodesets

The nodesets on the mesh at which we want to match the temperatures. Syntax: "match_nodesets {integer} {...}". These will be defined on the "reference" mesh which is the mesh that has the temperatures we are trying to match (see "ref_data_file"). Each nodeset should contain only a single node.

3.3.10.2 search_nodesets

The nodesets on the mesh generated with the ESP ".csm" file at each iteration that Plato will search to find node locations that match the "match_nodesets" node locations from the reference data mesh. Finding these matching nodes is required because in shape optimization problems the "current" mesh being evaluated in the physics code changes at each iteration and the code that calculates the temperature mismatch at each nodeset in the reference data mesh will need to know which node to compare against in the current iteration's mesh. Syntax: "search_nodesets {integer} {...}". Typically, these nodesets will be identified by attributing surfaces in the ".csm" file.

3.3.10.3 ref_data_file

The mesh file that contains the temperature information we are trying to match in the new design. Syntax: "ref_data_file {string}". It contains the nodesets at which we are trying to match temperatures (see "match_nodesets") and the temperature information obtained by running a thermal analysis.

3.3.10.4 temperature_field_name

The name of the temperature field in the "ref_data_mesh". Syntax: "temperature_field_name {string}."

3.3.11 Parameters for Volume-Based Criteria

Volume-based criteria can have an optional parameter specified. Optional parameters are currently supported only for the "volume_average_von_mises criterion."

3.3.11.1 block

The block in which to compute the volume-based criterion. Syntax: "block {block_id}".

3.3.12 Parameters for Stress P-Norm Criterion

Parameters related to the stress measure used for computing the stress p-norm can be specified. The default is to use the stress tensor norm.

3.3.12.1 stress_p_norm_measure

The stress measure used in the p-norm. If this parameter is not specified, the tensor norm will be used. Supported measures are: vonmises. Syntax: “stress_p_norm_measure {measure}”.

3.3.12.2 stress_p_norm_volume_scaling

A boolean (true or false) indicating whether the stress value in each element will be scaled by the element volume when computing the p-norm. This is only applied if a stress measure is specified. Syntax: “stress_p_norm_volume_scaling {bool}”.

3.4 Scenario

In this section, we show how to specify scenario blocks. Each scenario block begins and ends with the tokens “begin scenario {integer}" and “end scenario". The integer following “begin scenario" specifies the identification index of this scenario. Other blocks in the input deck will use this value to reference the scenario. The following is a typical scenario block definition:

```
begin scenario 1
   physics steady_state_mechanics
   dimensions 3
   loads 1 2
   boundary_conditions 5
   material 1
   minimum_ersatz_material_value 1e-3
end scenario
```

Plato input decks can contain one or more scenario blocks. The following tokens can be specified in any order within the scenario block.

3.4.1 General Parameters

3.4.1.1 physics

Each scenario must specify the physics in the format: “physics {string}". Table 3.3 lists the supported physics, a brief description, and which physics code can be used. In the table “SD” is sierra_sd, “TF” is sierra_tf, and “PA” is plato_analyze.

3.4.1.2 dimensions

Within each scenario the user must specify the dimensions of the problem in the format: “dimensions {integer}". Possible values are 2 and 3.

Table 3.3: Description of available physics.

Physics	**Description**	**Valid Code**
steady_state_mechanics	Static solution with simple linear elasticity	SD, PA
steady_state_thermal	Steady state heat conduction	PA
steady_state_thermomechanics	Static mechanical solution with heat conduction	PA
steady_state_incompressible_fluids	Incompressible fluid flow steady state solution	PA
modal_response	Modal analysis	SD
transient_thermal	Transient thermal analysis	TF

3.4.1.3 loads

Within each scenario the user must specify its relevant loads in the format: "loads {integer} {...}". The integer values are the ids of load blocks defined elsewhere in the input deck.

3.4.1.4 boundary_conditions

Within each scenario the user must specify its relevant boundary conditions in the format: "boundary_conditions {integer} {...}". The integer values are the ids of boundary_condition blocks defined in the input deck.

3.4.1.5 assemblies

Within each scenario the user may specify assemblies in the format: "assemblies {integer} {...}". The integer values are the ids of assembly blocks defined elsewhere in the input deck.

3.4.1.6 existing_input_deck

Specifies the name of an existing input deck to be used by the physics code associated with this scenario. Syntax: "existing_input_deck {string}". Currently, this is only used with Sierra/TF when doing temperature matching problems.

3.4.2 Density-Based Topology Optimization Parameters

3.4.2.1 minimum_ersatz_material_value

Within each scenario the user may specify the minimum density value for the residual equation using the format: "minimum_ersatz_material_value {value}". The default value

is $1e^{-6}$.

3.4.2.2 material_penalty_model

Within each scenario the user MAY specify the material penalty model to be used in density-based topology optimization problems using the format: "material_penalty_model {string}". Valid models are "simp" and "ramp". The default value is "simp". **Note: the RAMP model will be deprecated in the near future. Users are encouraged to use the default SIMP model.**

3.4.2.3 material_penalty_exponent

Within each scenario the user MAY specify the material penalty exponent to be used in density-based topology optimization problems using the format: "material_penalty_exponent {value}". The default value is 3.0.

3.4.3 Linear Solver Parameters

The following parameters are used to tune the linear solver used to solve the linear system of equations, e.g. the finite element equations.

3.4.3.1 linear_solver_max_iterations

This parameter specifies the maximum number of linear solver iterations to take. The default value is set to 1000. The syntax is: "linear_solver_max_iterations {integer}".

3.4.3.2 linear_solver_type

This parameter specifies the type of linear solver used to solve the linear system of equations. The supported options in Plato Analyze (PA) are tacho, amgx, epetra and tpetra. Tacho is a direct solver, and thus pays no attention to `linear_solver_max_iterations` or `linear_solver_tolerance`. Sierra-SD only uses "gdsw" for its linear solver. Therefore, Plato uses "gdsw" by default when running optimization-based design problems with Sierra-SD. The default solver for PA is set to "amgx" if running on GPU hardware and "epetra" if running on CPU hardware.

The syntax is: "linear_solver_type {string}".

3.4.3.3 linear_solver_tolerance

This parameter specifies the linear solver tolerance. The default value is set to $1e^{-8}$. The syntax is: "linear_solver_tolerance {value}". If the solver has not converged to the specified tolerance within the maximum allowed number of iterations, a fatal error occurs.

3.4.4 Sierra-SD Parameters

The following parameters are specific to Sierra-SD. Therefore, these parameters are not used and recognized by Plato Analyze.

3.4.4.1 weight_mass_scale_factor

A Sierra/SD-specific parameter for doing internal conversion of lbf and lbm when using English units. Here is the description from the SierraSD user's manual: "This variable multiplies all mass and density on the input, and divides out the results on the output. It is provided primarily for the english system of units where the natural units of mass are actually units of force. For example, the density of steel is 0.283 lbs/in3, but lbs includes the units of g= 386.4 in/s2. Using a value of wtmass of 0.00259 (1/386.4), density can be entered as 0.283, the outputs will be in pounds, but the calculations will be performed using the correct mass units." The format for this paramter is: "weight_mass_scale_factor {value}".

3.4.4.2 convert_to_tet10

Specifies that linear tet4 meshes should be converted to quadratic tet10 meshes when doing shape/CAD parameter optimization. Syntax: "convert_to_tet10 {Boolean}". When doing a CAD parameter shape optimization with the Engineering Sketch Pad (ESP) workflow ESP generates linear tet meshes and this option allows the user to specify that he wants midside nodes to be added to the linear tet meshes so that the physics will be calculated using tet10 elements. This is only applicable to problems using sierra_sd as the physics service.

3.4.5 Incompressible Fluid Flow Parameters

3.4.5.1 heat_transfer

This parameter specifies the heat transfer mechanism. Supported options are "natural" and "forced". Syntax: "heat_transfer {string}"

3.4.5.2 steady_state_iterations

This parameter specifies the allowable number of steady state iterations. Syntax: "steady_state_iterations {integer}". The default value is set to 500.

3.4.5.3 steady_state_tolerance

This parameter specifies the minimum tolerance to reach steady state. Syntax: "steady_state_tolerance {value}". The default value for optimization applications is set to 0.001.

3.4.5.4 time_step_safety_factor

This parameter specifies a safety factor greater than zero but less than one applied to the time step at each steady state iteration to guarantee steady state convergence. Syntax: 'time_step_safety_factor {value}". The default value is set to 0.7.

3.5 Objective

In this section, we show how to specify the objective block. Each input deck contains only one objective block. However, the objective can be made up of one or more sub-objectives. The objective block begins and ends with the tokens "begin objective" and "end objective". The following is a typical objective block definition:

```
begin objective
   type weighted_sum
   services 2 3
   criteria 3 4
   scenarios 1 2
   weights 1.0 0.75
end objective
```

The example objective above is made up of two sub-objectives. The first sub-objective uses service 2, criterion 3, scenario 1, and is weighted with a value of 1.0. The second sub-objective uses service 3, criterion 4, scenario 2, and is weighted with a value of 0.75. The services, criteria, and scenarios are defined elsewhere in the input deck and referenced in the objective by their id. In this example, to evaluate the objective the sub-objectives are evaluated, scaled by their corresponding weights, and then summed. The following tokens can be specified in any order within the objective block.

3.5.1 type

Each objective must specify its type using the format: "type {string}." Valid types are "single_criterion," "weighted_sum," and "multi_objective".

3.5.2 services

Each objective must specify the services used by the sub-objectives using the format: "services {integer} {...}". There must be one service specified for each sub-objective.

3.5.3 criteria

Each objective must specify the criteria used by the sub-objectives using the format: "criteria {integer} {...}". There must be one criterion specified for each sub-objective.

3.5.4 scenarios

Each objective must specify the scenarios used by the sub-objectives using the format: "scenarios {integer} {...}". There must be one scenario specified for each sub-objective.

3.5.5 weights

Each objective must specify the weights used by the sub-objectives if it is a "weighted_sum" type objective. This is done using the format: "weights {value} {...}." There must be one weight specified for each sub-objective. Weights need not sum to 1.0.

3.5.6 shape_services

For shape or CAD parameter optimization additional services are needed to provide the CAD parameter sensitivities for the optimizer. These services are specifed using the format: "shape_services {integer} {...}." There must be one shape_service specified for each sub-objective.

3.5.7 multi_load_case

When using the Sierra/SD physics code, there is an option to have a single sierra_sd service calculate the sub-objectives associated with multiple load cases in sequence rather than requiring multiple sierra_sd services. Running sequentially would typically be done when you are resource-limited and can't afford to run multiple instances of sierra_sd–one for each load case. In this case you would specify "multi_load_case true" in the objective block and set all of your sub-objective service ids to be the single sierra_sd service that will be running the different load cases in sequence. When the optimizer asks for the objective evaluation at each iteration, the sierra_sd service will calculate each of the sub-objectives corresponding to the different load cases sequentially, create a weighted sum of the sub-objectives using the weights specified in the objective block, and then return a single objective value to the optimizer.

3.6 Constraint

In this section, we show how to specify constraint blocks. The constraint block begins and ends with the tokens "begin constraint {integer}" and "end constraint." The integer following "begin constraint" uniquely identifies it in multi-constraint problems. Single-constraint problems are fully supported in Plato. Multi-constraint problems are implemented but not considered a fully supported feature yet. The following is a typical constraint block definition:

```
begin constraint 1
```

```
   service 1
   criterion 3
   scenario 1
   relative_target 0.5
end constraint
```

The following tokens can be specified in any order within the constraint block.

3.6.1 service

Each constraint must specify a service in the format: "service {integer}". This service will calculate the constraint value.

3.6.2 criterion

Each constraint must specify a criterion in the format: "criterion {integer}". This criterion will be evaluated in order to determine how well the constraint is being met. For example, in the example constraint above criterion 3 could be volume, in which case service 1 would calculate the volume, and then this value would be used to determine how well the relative constraint target of 0.5 was being met.

3.6.3 scenario

Each constraint must specify a scenario in the format: "scenario {integer}". The scenario defines the physics conditions under which the constraint is evaluated.

3.6.4 relative_target

Each constraint can specify a target value either as a relative target or an absolute target. For relative targets the syntax is: "relative_target {value}". Currently, the only constraint that makes sense to use a relative value for is volume. In this case the user specifies a target relative to the starting volume of the design domain. In the example constraint above the relative_target of 0.5 would mean we want the final design to use 50% of the original design domain as our volume budget.

3.6.5 absolute_target

Each constraint can specify a target value either as a relative target or an absolute target. For absolute targets the syntax is: "absolute_target {value}". An example of a constraint with an absolute target would be a volume constraint where you are specifying an absolute volume value that your final design must adhere to.

3.7 Load

In this section, we show how to specify load blocks. Each load block begins and ends with the tokens "begin load {integer}" and "end load". The integer following "begin load" specifies the identification index of this load. Other blocks in the input deck will use this value to reference the load. The following is a typical load block definition:

```
begin load 1
   type traction
   location_type sideset
   location_name ss_1
   value 0 -3e3 0
end load
```

The following tokens can be specified in any order within the load block.

3.7.1 type

Each load must specify the type using the format: "type {string}." Current options include "traction," "uniform_surface_flux," "pressure," "acceleration," "uniform_thermal_source,"[1], and "force." Table 3.4 lists a description of the allowable types of loads and the physics code they can be used with. In the table "SD" is sierra_sd and "PA" is plato_analyze.

3.7.2 location_type

Each load must specify the application location type using the format: "location_type {string}". Current options include "sideset" and "nodeset". When using Plato Analyze all loads are applied on sidesets.

3.7.3 location_name

Each load must specify an application location either by name or ID, depending on which physics code is being used. The syntax for specifying by name is:"location_name {string}". Sierra/SD uses ids to identify sidesets and nodesets, and Plato Analyze uses names to identify sidesets. Therefore, sidesets must be named when using Plato Analyze.

[1]The following material parameters: 1) "characteristic_length," "reference_temperature," and fluid "thermal_conductivity" or 2) "characteristic_length," "characteristic_velocity," "reference_temperature," and the fluid "thermal_conductivity," and "thermal_diffusivity" must be defined in the material block for natural or forced convection problems, respectively. The fluid solver solves the dimensionless set of equations. These quantities are needed to compute the parameter used to remove the dimensions from the source term.

Table 3.4: Description of supported load types.

Load Type	**Description**	**Valid Code**
traction	Arbitray direction traction load on a surface. Specify traction component values separated by spaces (e.g. 0.2 2e-3 200 in three dimensions).	SD, PA
pressure	Surface load normal to the surface. Specified by a single scalar value.	SD, PA
force	Point load applied at nodes in a nodeset or sideset. Specify force component values separated by spaces (e.g. 0.2 2e-3 200 in three dimensions)	SD
acceleration	Body load applied to the whole model.	SD
uniform_surface_flux	Thermal surface load. Single value specifying the normal flux to the surface.	PA
uniform_thermal_source	Single value uniform thermal source. This load is applied to an element block. The “location_name" parameter must be used to specify the element block.	PA

3.7.4 location_id

Each load must specify an application location either by name or id depending on which physics code is being used. The syntax for specifying by id is:“location_id {integer}.” Sierra/SD uses ids to identify sidesets and nodesets, and Plato Analyze uses names to identify sidesets. Therefore, sidesets must be named when using Plato Analyze.

3.7.5 value

Each load must specify a value using the syntax:“value {value} {...}.” Depending on the type of load, more than one value may need to be specified. For example, traction loads require x, y, and z components so it would be specified like this: “value 0 -3e3 0”.

3.8 Boundary Condition

In this section, we show how to specify essential/Dirichlet boundary_condition blocks. Each boundary_condition block begins and ends with the tokens “begin boundary_condition {integer}" and “end boundary_condition". The integer following “begin boundary_condition" specifies the identification index of this boundary_condition. Other blocks in the input deck will use this value to reference the boundary_condition. The following is a typical boundary_condition block definition:

```
begin boundary_condition 1
   type fixed_value
   location_type sideset
   location_name ss_1
   degree_of_freedom temp
   value 0.0
end boundary_condition
```

The following tokens can be specified in any order within the boundary_condition block.

3.8.1 type

Each boundary condition must specify the type using the format: “type {string}". Current options include “fixed_value” and “insulated”.

3.8.2 location_type

Each boundary condition must specify the application location type using the format: “location_type {string}". Current options include “sideset” and “nodeset”. When using Plato Analyze, all boundary conditions are applied on sidesets.

3.8.3 location_name

Each boundary condition must specify a application location either by name or ID, depending on which physics code is being used. The syntax for specifying by name is:"location_name {string}". Sierra/SD uses IDs to identify sidesets and nodesets, and Plato Analyze uses names to identify sidesets. Therefore, sidesets must be named when using Plato Analyze.

3.8.4 location_id

Each boundary condition must specify a application location either by name or ID, depending on which physics code is being used. The syntax for specifying by id is:"location_name {integer}". Sierra/SD uses IDs to identify sidesets and nodesets, and Plato Analyze uses names to identify sidesets. Therefore, sidesets must be named when using Plato Analyze.

3.8.5 degree_of_freedom

Depending on the boundary condition type, you may need to specify a degree of freedom. The syntax is:"degree_of_freedom {string} {...}". Possible values for degrees of freedom are:

1. "press" (pressure),
2. "temp" (temperature),
3. "velx" (velocity in the x direction),
4. "vely" (velocity in the y direction),
5. "velz" (velocity in the z direction),
6. "dispx" (displacement in the x direction),
7. "dispy" (displacement in the y direction), and
8. "dispz" (displacement in the z direction).

Multiple degrees of freedom can be specified in a single "fixed_value" boundary condition by listing the degrees of freedom separated by spaces. For example, you can specify a fixed value boundary condition for all 3 displacement directions with the following: "degree_of_freedom dispx dispy dispz." If you specify more than one degree of freedom in this way, you must also have the same number or corresponding values in the "value" line. So, for the example just given your "value" line would need to be: "value 0 0 0" for a fixed value of 0.0 in all 3 directions.

3.8.6 value

Each boundary condition can specify a value using the syntax:“value {value} {...}.”

3.9 Assembly

In this section, we show how to specify assembly blocks. Each assembly block begins and ends with the tokens “begin assembly {integer}" and “end assembly". The integer following “begin assembly" specifies the identification index of this assembly. The scenario block in the input deck will use this value to reference the assembly. The following is a typical assembly block definition:

```
begin assembly 1
   type tied
   child_nodeset ns_1
   parent_block 2
end assembly
```

The following tokens can be specified in any order within the assembly block.

3.9.1 type

Each assembly must specify the type using the format: “type {string}". Currently, the only option is “tied”, which refers to tied contact. [2]

3.9.2 child_nodeset

Each assembly must specify a nodeset that defines the interface of the assembly using the format: “child_nodeset {string}". The string must be a valid nodeset name.

3.9.3 parent_block

Each assembly must specify a parent block that the child nodeset will interface with. The syntax for specifying by name is:“parent_block {integer},” where the integer corresponds to the identification index of a block in the input deck.

3.10 Block

In this section, we show how to specify “block” blocks. Each “block” block begins and ends with the tokens “begin block {integer}" and “end block". The integer following “begin

[2] Tied contact is currently only supported in PlatoAnalyze, so the use of assemblies must be restricted to scenarios that are carried out with PlatoAnalyze as the performer.

block" specifies the identification index of this "block". Other blocks in the input deck will use this value to reference the "block". The following is a typical "block" definition:

```
begin block 1
   material 1
   element_type tet4
end block
```

The following tokens can be specified in any order within the "block" block.

3.10.1 name

Each block must specify the application location by name for Plato Analyze problems. The syntax for specifying the block name is: "name {string}". The following is a typical "block" definition for Plato Analyze:

```
begin block 1
   material 1
   name my_element_block_name
end block
```

Plato Analyze only runs with tet4 elements. Therefore, the element_type token does not need to be specified for Plato Analyze problems.

3.10.2 material

Each block must specify a material using the format: "material {integer}".

3.10.3 element_type

Each block can specify the element type using the format: "element_type {string}". Table 3.5 lists a description of the allowable element types and what physics code they can be used with. In the table "SD" is sierra_sd and "PA" is plato_analyze.

Table 3.5: Description of supported element types.

Element Type	Description	Valid Code
tet4	First-order tet element.	SD, PA
hex8	First-order hex element.	SD
tet10	Second-order tet element.	SD
rbar	See SierraSD User's Manual.	SD
rbe3	See SierraSD User's Manual.	SD

3.10.4 sub_block

A region of a block can be used to define a sub-block. Sub-blocks can be useful for shape optimization workflows that generate a single element block. This capability requires that exactly two blocks are defined in the input deck. The sub-block is created from block 1 and used to define block 2. A bounding box defines the sub-block with coordinates xmin ymin zmin xmax ymax zmax. The following shows an example.

```
begin block 1
   material 1
   sub_block xmin ymin zmin xmax ymax zmax
end block
begin block 2
   material 1
end block
```

3.11 Material

In this section, we show how to specify material blocks. Each material block begins and ends with the tokens "begin material {integer}" and "end material". The integer following "begin material" specifies the identification index of this material. Other blocks in the input deck will use this value to reference the material. The following is a typical material block definition:

```
begin material 1
   material_model isotropic_linear_thermal
   thermal_conductivity 210
   mass_density 2703
   specific_heat 900
end material
```

The following tokens can be specified in any order within the material block.

3.11.1 material_model

Each material must specify a material_model using the format: "material_model {string}". Table 3.6 lists the allowable material models and what physics code they can be used with.

3.11.2 Isotropic Linear Elastic Properties

youngs_modulus. Syntax: "youngs_modulus {value}".

poissons_ratio. Syntax: "poissons_ratio {value}".

Table 3.6: Description of supported material models.

Material Model	**Valid Code**
isotropic_linear_elastic	SD, PA
orthotropic_linear_elastic	PA
isotropic_linear_thermal	PA
isotropic_linear_thermoelastic	PA
laminar_flow	PA
forced_convection	PA
natural_convection	PA

mass_density. Syntax: "mass_density {value}".

3.11.3 Orthotropic Linear Elastic Properties

youngs_modulus_x. Syntax: "youngs_modulus_x {value}".

youngs_modulus_y. Syntax: "youngs_modulus_y {value}".

youngs_modulus_z. Syntax: "youngs_modulus_z {value}".

poissons_ratio_xy. Syntax: "poissons_ratio_xy {value}".

poissons_ratio_xz. Syntax: "poissons_ratio_xz {value}".

poissons_ratio_yz. Syntax: "poissons_ratio_yz {value}".

shear_modulus_xy. Syntax: "shear_modulus_xy {value}".

shear_modulus_xz. Syntax: "shear_modulus_xz {value}".

shear_modulus_yz. Syntax: "shear_modulus_yz {value}".

mass_density. Syntax: "mass_density {value}".

3.11.4 Isotropic Linear Thermal Properties

thermal_conductivity. Syntax: "thermal_conductivity {value}".

mass_density. Syntax: "mass_density {value}".

specific_heat. Syntax: "specific_heat {value}".

3.11.5 Isotropic Linear Thermoelastic Properties

thermal_conductivity. Syntax: "thermal_conductivity {value}".

youngs_modulus. Syntax: "youngs_modulus {value}".

poissons_ratio. Syntax: "poissons_ratio {value}".

thermal_expansivity. Syntax: "thermal_expansivity {value}".

reference_temperature. Syntax: "reference_temperature {value}".

mass_density. Syntax: "mass_density {value}".

3.11.6 Laminar Flow Properties

The "laminar_flow" material model is used for incompressible computational fluid dynamics use cases. The parameters defining this material model are:

darcy_number. Syntax: "darcy_number {value}".

reynolds_number. Syntax: "reynolds_number {value}".

impermeability_number. Syntax: "impermeability_number {value}".

The "darcy_number" or "impermeability_number" are **ONLY** needed for density-based topology optimization problems. These parameters define the porosity of the solid material. If the "darcy_number" is defined, the impermeability κ is computed as $\kappa(Re, Da) = 1/(ReDa)$, where Re and Da are the Reynolds and Darcy numbers, respectively. If the "impermeability_number" is defined, Plato will use the value provided for the "impermeability_number" instead of the relationship $\kappa(Re, Da)$ to calculate the

impermeability.

3.11.7 Forced Convection Properties

The “forced_convection" material model is used for incompressible computational fluid dynamics use cases where heat transfer is driven by forced convection. The parameters defining this material model are:

darcy_number. Syntax: “darcy_number {value}”.

reynolds_number. Syntax: “reynolds_number {value}”.

prandtl_number. Syntax: “prandtl_number {value}”.

thermal_diffusivity. Syntax: “thermal_diffusivity {value}”.

kinematic_viscocity. Syntax: “kinematic_viscocity {value}”.

thermal_conductivity. Syntax: “thermal_conductivity {value}”.

characteristic_length. Syntax: “characteristic_length {value}”.

temperature_difference. Syntax: “temperature_difference {value}”.

The “darcy_number" or “impermeability_number" are **ONLY** needed for density-based topology optimization problems. These parameters define the porosity of the solid material. If the “darcy_number" is defined, the impermeability κ is computed as $\kappa(Re, Da) = 1/(ReDa)$, where Re and Da are the Reynolds and Darcy numbers, respectively. If the “impermeability_number" is defined, Plato will use the value provided for the “impermeability_number" instead of the relationship $\kappa(Re, Da)$ to calculate the impermeability.

Users only need to provide values for the “characteristic_length", “thermal_conductivity", and “temperature_difference" parameters if a thermal source is applied. Likewise, the “kinematic_viscocity" and “thermal_diffusivity" are only defined if the user prefers to use a material-aware critical time step.

3.11.8 Natural Convection Properties

The "natural_convection" material model is used for incompressible computational fluid dynamics use cases where heat transfer is driven by natrual convection. The parameters defining this material model are:

darcy_number. Syntax: "darcy_number {value}".

prandtl_number. Syntax: "prandtl_number {value}".

grashof_number. Syntax: "grashof_number {value} {...}".

rayleigh_number. Syntax: "grashof_number {value} {...}".

richardson_number. Syntax: "grashof_number {value} {...}".

thermal_diffusivity. Syntax: "thermal_diffusivity {value}".

kinematic_viscocity. Syntax: "kinematic_viscocity {value}".

thermal_conductivity. Syntax: "thermal_conductivity {value}".

characteristic_length. Syntax: "characteristic_length {value}".

temperature_difference. Syntax: "temperature_difference {value}".

impermeability_number. Syntax: "impermeability_number {value}".

The "darcy_number" or "impermeability_number" are **ONLY** needed for density-based topology optimization problems. These parameters define the porosity of the solid material. If the "darcy_number" is defined, the impermeability κ is computed as $\kappa(Re, Da) = 1/(ReDa)$, where Re and Da are the Reynolds and Darcy numbers, respectively. If the "impermeability_number" is defined, Plato will use the value provided for the "impermeability_number" instead of the relationship $\kappa(Re, Da)$ to calculate the impermeability.

The user only needs to define one dimensionless parameter out of this list:

- "**rayleigh_number**"
- "**richardson_number**"
- "**grashof_number**

in natural convection problems". The "characteristic_length", "thermal_conductivity" and "temperature_difference" are only defined if a thermal source is applied. Likewise, the "kinematic_viscocity" and "thermal_diffusivity" are only defined if the user prefers to use a material-aware critical time step.

3.12 Optimization Parameters

In this section, we show how to specify the optimization parameters block. The optimization parameters block begins and ends with the tokens "begin optimization_parameters" and "end "optimization_parameters". The following is a typical optimization parameters block definition:

```
begin optimization_parameters
   optimization_algorithm oc
   discretization density
   initial_density_value 0.5
   filter_radius_scale 1.75
   max_iterations 10
   output_frequency 5
end optimization_parameters
```

The following tokens can be specified in any order within the optimization parameters block.

3.12.1 General Optimization Parameters

The following set of parameters are used to define general concepts for the optimization problem.

3.12.1.1 optimization_type

This parameter specifies the type of optimization you are doing. The syntax is: "optimization_type {string}" and valid options are "topology", "shape", and "dakota".

3.12.1.2 optimization_algorithm

This parameter specifies which optimization algorithm will be used. The syntax is: "optimization_algorithm {string}". Currently supported options are "rol_linear_constraint", "rol_bound_constrained", and "rol_augmented_lagrangian". "rol_linear_constraint", and "rol_augmented_lagrangian" require a constraint in the optimization problem, but "rol_bound_constrained" does not. "rol_linear_constraint" can be used on any problem with a single, linear constraint.

3.12.1.3 check_gradient

This parameter specifies whether the gradient check should be performed using ROL's gradient checking capability The syntax is: "check_gradient {boolean}." Creates an output file named "ROL_gradient_check_output.txt."

3.12.1.4 rol_gradient_check_perturbation_scale

This parameter specifies the initial perturbation scale that ROL will use during its gradient check. The syntax is: "rol_gradient_check_perturbation_scale {double}".

3.12.1.5 rol_gradient_check_steps

This parameter specifies the number of steps that ROL will take in checking the gradient. Each step denotes a factor of ten reduction in step size in the FD approximation. The syntax is: "rol_gradient_check_steps {integer}".

3.12.1.6 fixed_block_ids

This parameter allows the user to specify which mesh blocks in a topology optimization problem should remain fixed and not "designed" by the optimizer. The syntax is: "fixed_block_ids {integer} {...}". The integer values are the ids of the blocks that should remain fixed.

3.12.1.7 enforce_bounds

This parameter allows the user to specify whether the bounds on the topology values will be enforced explicitly upon output of the design. Explicit enforcement is most relevant when using fixed blocks and the default when fixed blocks exist is for the enforce_bounds option to be set to true. However, the user can override this default. Enforcing bounds will guarantee that material in fixed blocks is not "eaten away" by the optimizer. The syntax is: "enforce_bounds {Boolean}".

3.12.1.8 fixed_sideset_ids

This parameter allows the user to specify which mesh sidesets in a topology optimization problem should remain fixed and not "designed" by the optimizer. The syntax is: "fixed_sideset_ids {integer} {...}". The integer values are the ids of the sidesets that should remain fixed.

3.12.1.9 fixed_nodeset_ids

This parameter allows the user to specify which mesh nodesets in a topology optimization problem should remain fixed and not "designed" by the optimizer. The syntax is: "fixed_nodeset_ids {integer} {...}". The integer values are the ids of the nodesets that should remain fixed.

3.12.1.10 max_iterations

This parameter specifies how many iterations the optimizer will take if it doesn't meet some other stopping criteria first. The syntax is: "max_iterations {integer}."

3.12.1.11 output_frequency

This parameter specifies how often Plato will output a design result. The syntax is: "output_frequency {integer}" where the specified integer is how many iterations between design output. Design iterations are in the form of an exodus mesh and are written to the run directory. They are usually called something like "Iteration005.exo," where the number in the filename indicates which iteration it came from. If running from the Plato GUI, the design result will automatically be loaded as a Cubit^{TM} model and displayed in the graphics window.

3.12.1.12 output_method

This parameter specifies how results in parallel runs will be concatenated to a single file. The syntax is: "output_method {string}." The two valid options are "epu" and "parallel_write". The "epu" option will write result files in parallel to disk and then run the epu utility to concatenate the results. The "parallel_write" option will use a parallel writing capability to write the results to a single concatenated result file without the need to run epu afterwards. The reason you might choose one over the other is if the performance proves to be better with one option over the other.

problem_update_frequency. This parameter specifies how many iterations will happen between calls to the "UpdateProblem" operation during the optimization run. The update problem operation is important for situations like applying continuation methods on penalty and projection function parameters in order to slowly increase the nonlinearity of the optimization problem, often providing better, more consistent results. The syntax is: "problem_update_frequency {integer}."

3.12.1.13 initial_density_value

This parameter allows the user to specify the initial density value for a topology optimization run. The syntax is: "initial_density_value {value}."

3.12.1.14 write_restart_file

This parameter specifies whether to write restart files or not. The syntax is: "write_restart_file {Boolean}." The reason you might set this to false is if writing restart files is taking too long and you want to improve performance. However, be aware that restart files are needed if you will be restarting your run for any reason.

3.12.1.15 normalize_in_aggregator

This parameter allows the user to specify whether or not the objective values being returned from services will be normalized by an aggregation operation before they are passed along to the optimizer. The syntax is: "normalize_in_aggregator {Boolean}." The default is for them to be normalized. There are a couple of advantages of normalizing. First, optimizer performance is typically enhanced when the objective value is approximately on the order of 1. Second, when running a problem that has more than one sub-objective, the sub-objective values will be of the same order of magnitude, allowing the weights to be specified more intuitively.

3.12.1.16 verbose

This parameter allows the user to specify whether verbose information will be output to the console during the optimization run. The syntax is: "verbose {Boolean}". If this is set to "true," information about which stages and operations are being executed will be output to the console.

3.12.2 Filter Parameters

In density-based topology optimization problems, it is often necessary to perform some sort of filtering (i.e. spatial weighted averaging) of the design variables themselves in order to avoid numerical instabilities and alleviate some of the mesh dependency of the optimized result. Additionally, the filter helps to ensure an approximate minimum length scale of features in the optimized design. While the filter does not completely eliminate the issue of mesh-dependency in many cases, it does greatly alleviate much of the issue and, therefore, should almost always be used. We recommend setting the filter radius to at least twice the size of a typical finite element in the design domain. Additionally we provide the option to utilize subsequent projection operations, which often help to result in designs that are closer to black-and-white (i.e., 0 or 1) when the filter operation results in a large transition region with intermediate density values.

3.12.2.1 filter_radius_scale

This parameter allows the user to specify the filter radius as a scaling of the average length of the edges in the mesh. The syntax is: "filter_radius_scale {value}," where the value is the scale factor that will be used to come up with the filter radius. For example, if the average mesh edge length is 1.5 and the scale factor is specified as 2.0, then the resulting filter radius will be 3.0.

3.12.2.2 filter_radius_absolute

This parameter allows the user to specify the filter radius as an absolute value. The syntax is: "filter_radius_absolute {value}," where the value is what will be used as the filter radius.

3.12.2.3 filter_type

This parameter specifies what type of filter to use on the design variables in topology optimization runs. The syntax is: "filter_type {string}." Valid filter types are "identity," "kernel," and "helmholtz." When using type "helmholtz," you must used "filter_radius_absolute" rather than "filter_radius_scale."

3.12.2.4 projection_type

This parameter specifies what type of projection method will be used to try to force density values to zero and one. The syntax is: "projection_type {string}." Valid projection types are "tanh", and "heaviside".

3.12.2.5 boundary_sticking_penalty

This parameter is used to prevent aggressive aggregation of material along the design domain boundaries. Valid values for this parameter are between zero and one. If the "boundary_sticking_penalty" token is set to zero, the Helmholtz filter is free to place material along the design domain boundaries. If the "boundary_sticking_penalty" token is set to one, the Helmholtz filter will avoid placing material on the design domain boundaries. This parameter cannot be used with the Kernel filter. The syntax is: "boundary_sticking_penalty {value}."

3.12.2.6 symmetry_plane_location_names

This parameter is used to specify the location name of the sidesets where symmetry boundary conditions are enforced. If the "boundary_sticking_penalty" token is set to a nonzero value, the Helmholtz filter will exclude from the calculations the sidesets were symmetry

boundary conditions are enforced. This parameter cannot be used with the Kernel filter. The syntax is: "symmetry_plane_location_names {string$_1$,...,string$_N$}."

3.12.2.7 filter_heaviside_min

Determines the initial value of the heaviside steepness parameter for heaviside projection. A value near zero results in a near-linear projection, and as the value increases to infinity, the projection becomes closer to a true heaviside function. The syntax is: "filter_heaviside_min {value}."

3.12.2.8 filter_heaviside_max

Determines the maximum value of the heaviside steepness parameter for heaviside projection. A value near zero results in a near-linear projection, and as the value increases to infinity, the projection becomes closer to a true heaviside function. The syntax is: "filter_heaviside_max {value}."

3.12.2.9 filter_heaviside_update

The value by which the heaviside steepness parameter increases as it increases from filter_heaviside_min to filter_heaviside_max when using heaviside projection. If filter_use_additive_continuation is set to false, the heaviside steepness parameter is multiplied by this value each time the projection is updated – otherwise, this value is added to the heaviside steepness parameter. The syntax is: "filter_heaviside_update {value}."

3.12.2.10 filter_projection_start_iteration

The first optimization iteration that the heaviside steepness parameter will be updated when using heaviside projection. The syntax is: "filter_projection_start_iteration {integer}."

3.12.2.11 filter_projection_update_interval

The frequency of optimization iterations with which to update the heaviside steepness parameter when using heaviside projection. The syntax is: "filter_projection_update_interval {integer}."

3.12.2.12 filter_use_additive_continuation

If set to true, the heaviside steepness parameter will be increased additively by filter_heaviside_update on each update iteration when using heaviside projection. If set to false, the heaviside steepness parameter will be multiplied by filter_heaviside_update on each update iteration. The syntax is: "filter_use_additive_continuation {Boolean}."

3.12.2.13 filter_in_engine

This parameter allows the user to specify whether design variable filtering will take place in the PlatoMain service during topology optimization problems. Filtering in PlatoMain is the default behavior and should be used except when running problems using Plato Analyze that enforce symmetry. In this case filtering is done within Plato Analyze as part of the symmetry enforcement and so shouldn't be done again in PlatoMain. In that case you would set this parameter to "false." The syntax is: "filter_in_engine {Boolean}."

3.12.3 Restart Parameters

3.12.3.1 initial_guess_file_name

This parameter specifies the name of the file used as the initial guess in a restart run. The syntax is: "initial_guess_file_name {string}." This file will typically be a result from a previous run in the form of a restart file or the main platomain.exo output file that contains all of the iteration information from a previous run.

3.12.3.2 initial_guess_field_name

This parameter specifies the name of the nodal field within the initial guess file (see "initial_guess_file_name") that contains the design variable to be used as the initial guess for the restart run. The syntax is: "initial_guess_field_name {string}."

3.12.3.3 restart_iteration

This parameter specifies the iteration in the initial_guess_file_name that will be used to extract design variables for the initial guess for the restart run. The syntax is: "restart_iteration {integer}."

3.12.4 Prune and Refine Parameters

3.12.4.1 prune_mesh

This parameter specifies whether the mesh will be pruned during a prune and refine operation. The syntax is: "prune_mesh {Boolean}."

3.12.4.2 prune_threshold

This parameter specifies the threshold value for whether the mesh will be kept or pruned. Values larger than the threshold will be kept. The syntax is: "prune_threshold {value}."

3.12.4.3 number_refines

This parameter specifies the number of uniform mesh refinements that will take place as part of the prune and refine operation. The syntax is: "number_refines {integer}."

3.12.4.4 number_buffer_layers

This parameter specifies the number of buffer element layers that will be left around the pruned mesh during a prune and refine operation. The syntax is: "number_buffer_layers {integer}." Buffer layers around the pruned mesh allow the design to evolve with additional freedom while avoiding running into the boundary of the pruned mesh. The more buffer layers you specify, the more room the design will have to evolve in. However, the more buffer layers you specify, the larger the mesh and the more computationally expensive the problem becomes.

3.12.4.5 number_prune_and_refine_processors

This parameter specifies the number of processors to use in the prune and refine operation. The syntax is: "number_prune_and_refine_processors {integer}."

3.12.5 Symmetry Parameters

3.12.5.1 symmetry_plane_normal

The normal vector orienting the symmetry plane. The syntax is: "symmetry_plane_normal { x-coordinate y-coordinate z-coordinate }."

3.12.5.2 symmetry_plane_origin

The location of the symmetry plane. The syntax is: "symmetry_plane_origin { x-coordinate y-coordinate z-coordinate }."

3.12.5.3 mesh_map_filter_radius

The radius for filtering symmetrized geometry. The syntax is: "mesh_map_filter_radius { double }."

3.12.5.4 filter_before_symmetry_enforcement

A flag to determine whether the filter is applied before or after symmetry enforcement. The syntax is: "filter_before_symmetry_enforcement { bool }."

3.12.5.5 mesh_map_search_tolerance

The tolerance for finding elements that contain locations of nodes mapped across the symmetry plane. This tolerance is a fraction of the element size. The syntax is: "mesh_map_search_tolerance { double }."

3.12.6 Shape Optimization Parameters

3.12.6.1 csm_file

This parameter specifies the name of the csm file that will be used in the shape optimization. The syntax is: "csm_file {string}." The csm file must be generated in Engineering Sketch Pad (ESP) before running the shape optimization.

3.12.6.2 num_shape_design_variables

This parameter specifies the number of design variables in the shape optimization problem. When setting up a shape optimization problem using Engineering Sketch Pad, you will decide which CAD parameters will be optimized. Then, when you set up the Plato input deck, you will need to specify the number of CAD parameters that are being optimized so that Plato can initialize the problem correctly.

For more help on setting up shape optimization problems contact the Plato team at plato3D-help@sandia.gov. The syntax for this parameter is: "num_shape_design_variables {integer}."

3.12.7 Optimizer Parameters

The following set of parameters are used to tune the optimization algorithms in Plato.

3.12.7.1 Surrogate-Based Global Optimizer

3.12.7.1.1 optimization_type: Must be set to "dakota" for Surrogate-Based Global Optimizer workflow.

3.12.7.1.2 dakota_workflow: Specifies which supported dakota workflow to use. Valid options are "sbgo" and "mdps". The user must define this parameter. For Surrogate-Based Global Optimizer workflow, use "sbgo". The syntax is: "dakota_workflow {string}."

3.12.7.1.3 concurrent_evaluations: Number of concurrent simulations/evaluations orchestrated by Plato. The user must define this parameter. The syntax is: "concurrent_evaluations {integer}."

3.12.7.1.4 num_sampling_method_samples: Number of samples for the sampling algorithm. This samples are used to generate new high-fidelity data in the data pool. The updated data pool is later used to recreate the surrogates and improve their predictive ability. The default is set to 15.

The syntax is: "num_sampling_method_samples {integer}."

3.12.7.1.5 sbgo_max_iterations: Maximum number of iterations for the Surrogate-Based Global Optimizer (SBGO). The default is set to 10. The syntax is: "sbgo_max_iterations {integer}."

3.12.7.1.6 sbgo_surrogate_output_name: File name for writing surrogate model files. If this parameter is not assigned, surrogate model files will not be written. The default is empty. The syntax is: "sbgo_surrogate_output_name {string}."

3.12.7.1.7 moga_population_size: Population size for Multi-Objective Genetic Algorithm (MOGA). The default is set to 300. The syntax is: "moga_population_size {integer}."

3.12.7.1.8 moga_max_function_evaluations: Maximum number of function evaluations for the MOGA. The default is set to 20,000. The syntax is: "moga_max_function_evaluations {integer}."

3.12.7.1.9 moga_niching_distance: This parameter specifies the niching Euclidean distance between design candidates. The minimum allowable distance between any two designs in the performance space is the Euclidian (simple square-root-sum-of-squares calculation) distance defined by these percentages. The purpose of niching is to encourage differentiation along the Pareto frontier and thus a more even and uniform sampling. The default is set to 0.1.

The syntax is: "moga_niching_distance {value}."

3.12.7.2 Multi-Dimensional Parameter Study

3.12.7.2.1 optimization_type: Must be set to "dakota" for Multi-Dimensional Parameter Study workflow.

3.12.7.2.2 dakota_workflow: Specifies which supported dakota workflow to use. Valid options are "sbgo" and "mdps." The user must define this parameter. For Multi-Dimensional Parameter Study workflow, use "mdps." The syntax is: "dakota_workflow {string}."

3.12.7.2.3 concurrent_evaluations: Number of concurrent simulations/evaluations orchestrated by Plato. The user must define this parameter. The syntax is: "concurrent_evaluations {integer}."

3.12.7.2.4 mdps_partitions: This parameter specifies how design variables are sampled on a full factorial grid of study points. The user must define this parameter to do a Multi-Dimensional Parameter Study (MDPS). The syntax is: "mdps_partitions {value}, ..., {value}."

3.12.7.2.5 mdps_response_functions: This parameter specifies the number of generic responses to be evaluated. Each of these functions is simply a response quantity of interest with no special interpretation taken by the method in use. Whereas objective, constraint, and residual functions have special meanings for gradient-based optimization 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. The user must define this parameter.

The syntax is: "mdps_response_functions {integer}."

3.12.7.3 Newton Optimizers

These parameters are associated with the second-order optimizers (e.g., trust-region methods) in Plato.

3.12.7.3.1 hessian_type: This parameter specifies the type of Hessian approximation that will be used. Supported options are "lbfgs" and "analytical." However, the "analytical" options will only work if the physics app code provides the analytical Hessian information. The default Hessian is set to an identity matrix.

The syntax is: "hessian_type {string}"

3.12.7.3.2 limited_memory_storage: This parameter specifies how much gradient history storage will be used for the lbfgs hessian approximation. The default is 8. The syntax is: "limited_memory_storage {integer}."

3.12.7.4 Symmetry Enforcement Parameters

3.12.7.4.1 symmetry_plane_origin: This parameter specifies the origin of the plane about which symmetry will be enforced during topology optimization problems. The syntax is: "symmetry_plane_origin {value} {value} {value}." The three values are the three components of the plane origin.

3.12.7.4.2 symmetry_plane_normal: This parameter specifies the normal of the plane about which symmetry will be enforced during topology optimization problems. The syntax is: "symmetry_plane_normal {value} {value} {value}." The three values are the three components of the plane normal.

3.12.7.4.3 mesh_map_filter_radius: This parameter specifies the radius of the filter when filtering is performed in the Plato Analyze service and not in the PlatoMain service (see "filter_in_engine"). The syntax is: "mesh_map_filter_radius {value}."

3.12.7.4.4 filter_before_symmetry_enforcement: This parameter specifies whether or not to filter before symmetry enforcement in Plato Analyze. The syntax is: "filter_before_symmetry_enforcement {Boolean}."

3.13 Output

In this section, we show how to specify output blocks. Each output block begins and ends with the tokens "begin output" and "end output". Typically you will have one output block for each service that is calculating quantities of interest. The following is a typical output block definition:

```
begin output
   service 2
   data temperature
end output
```

The following tokens can be specified in any order within the output block.

3.13.1 service

Each output block must specify the service providing the output using the syntax: "service {integer}".

3.13.2 data

Each output block must specify the data to output using the syntax: "data {string} {...}". Table 3.7 lists the allowable outputs and what physics code they can be used with. In the table "SD" is sierra_sd and "PA" is plato_analyze.

3.13.3 native_service_output

This option specifies whether to have the service output quantities of interest in its native format. Syntax: "native_service_output {Boolean}." Currently, this option only applies to the Plato Analyze physics service.

Table 3.7: Description of supported output.

Output	**Description**	**Valid Code**
dispx	X displacement	SD, PA
dispy	Y displacement	SD, PA
dispz	Z displacement	SD, PA
vonmises	Vonmises stress.	SD, PA
temperature	Temperature	PA

3.14 Mesh

In this section, we show how to specify the mesh block. The mesh block begins and ends with the tokens “begin mesh" and “end mesh". The following is a typical mesh block definition:

```
begin mesh
   name component.exo
end mesh
```

The following tokens can be specified in any order within the mesh block.

3.14.1 name

The mesh block must specify the name of the mesh file being used for the optimization run. The syntax is: “name {string}".