# Contents

1 Introduction 5

2 General Concepts 6
   2.1 Service 6
   2.2 Criterion 6
   2.3 Scenario 6
   2.4 Objective 7
   2.5 Constraint 7

3 Input Deck Options 8
   3.1 Syntax 8
   3.2 Service 10
      3.2.1 code 10
      3.2.2 number_processors 10
      3.2.3 device_ids 10
      3.2.4 cache_state 10
      3.2.5 update_problem 11
      3.2.6 path 11
   3.3 Criterion 11
      3.3.1 type 11
      3.3.2 criterion_ids 12
      3.3.3 criterion_weights 12
      3.3.4 Criterion Parameters Related to Stress-constrained Mass Minimization Problems (“type stress_and_mass”) 13
      3.3.5 Criterion Parameters Related to Mode Matching Problems (“type modal_projection_error”) 13
   3.4 Scenario 14
      3.4.1 physics 15
      3.4.2 dimensions 15
      3.4.3 loads 15
      3.4.4 boundary_conditions 15
      3.4.5 minimum_ersatz_material_value 16
      3.4.6 tolerance 16
      3.4.7 material_penalty_model 16
      3.4.8 material_penalty_exponent 16
      3.4.9 weight_mass_scale_factor 16
      3.4.10 convert_to_tet10 16
   3.5 Objective 17
      3.5.1 type 17
3.12 Output ......................................................... 33
  3.12.1 service ................................................. 33
  3.12.2 data .................................................... 33
  3.12.3 native_service_output ................................. 33
3.13 Mesh .......................................................... 34
  3.13.1 name ..................................................... 34
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 performance 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 geometry 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 5 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 my objective was to minimize the compliance of a structure (make it stiffer) the criterion would be some measure of the compliance of the structure. If my objective was to match a set of user-supplied mass properties, my criterion would be the mismatch between the current iteration’s mass properties and those provided by the user. Criteria are also used in constraints. If I have a volume constraint as part of my problem the criterion for that constraint would be the volume. 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 inde-
pendently in the input deck so that they can be defined once and then referenced by other items in multiple 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 3 load ids “1 5 6”. Table 1 shows the different types of values and a description of each.

Table 1: Description of value types.

<table>
<thead>
<tr>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Data that can only take on the value true or false.</td>
</tr>
<tr>
<td>integer</td>
<td>Data that can only take on integer values. Example: 4 10 50.</td>
</tr>
<tr>
<td>value</td>
<td>Data that can only take on real or floating point values. Example: 4.33.</td>
</tr>
<tr>
<td>string</td>
<td>Data that can only take on string values. Example: stress.</td>
</tr>
</tbody>
</table>

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 1: Example Plato input deck.

Figure 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 service 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”, “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 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. This is done 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. This is done 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. This is done 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. This is done 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 2 lists a description of the allowable types and what physics code they can be used with. In the table “SD” is sierra_sd 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} {...}”.

Table 2: Description of supported criterion types and applicable physics codes.

<table>
<thead>
<tr>
<th>Criterion Type</th>
<th>Description</th>
<th>Valid Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>composite</td>
<td>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. For this to be valid all of 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.</td>
<td>PA</td>
</tr>
<tr>
<td>mechanical_compliance</td>
<td>A measure of the stiffness of the structure. Minimizing this measure will make the structure stiffer.</td>
<td>SD, PA</td>
</tr>
<tr>
<td>thermal_compliance</td>
<td>A measure of the resistance to heat conduction. Minimizing this will maximize heat conduction.</td>
<td>PA</td>
</tr>
<tr>
<td>stress_and_mass</td>
<td>Used for doing stress-constrained mass minimization problems.</td>
<td>SD, PA</td>
</tr>
<tr>
<td>stress_p-norm</td>
<td>Superscript p used to compute the norm of the stress.</td>
<td>PA</td>
</tr>
<tr>
<td>volume</td>
<td>The volume of the current design.</td>
<td>PA</td>
</tr>
<tr>
<td>flux_p-norm</td>
<td>Superscript p used to compute the norm of the heat flux.</td>
<td>PA</td>
</tr>
<tr>
<td>modal_projection_error</td>
<td>A measure of the mismatch between the modes of the current design iteration and a user-specified set of modes.</td>
<td>SD</td>
</tr>
</tbody>
</table>
3.3.4 Criterion Parameters Related to Stress-constrained Mass Minimization Problems (“type stress_and_mass”)

The stress-constrained mass minimization problem formulation 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.

**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}”.

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

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

**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.

**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.**

3.3.5 Criterion Parameters Related to Mode Matching Problems (“type modal_projection_error”)

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

**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 3 non-rigid body modes you should set this value to 20 for good convergence. Syntax: “num_modes_compute {integer}”.

**modes_to_exclude.** These are 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 3 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”.

**match_nodesets.** These are 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.

**shape_sideset.** This specifies the sideset in the mesh where shape sensitivities will be calculated for shape/CAD optimization problems using the “modal_projection_error” criterion type. Syntax: “shape_sideset {integer}”. Currently all surfaces that should be considered in the shape sensitivity calculation must be included in a single sideset. Surfaces that should be included in this sideset are surfaces that would be affected by changes to the CAD parameters that are being optimized.

**ref_data_file.** This specifies 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.

**eigen_solver_shift.** This is an eigen solver parameter that should usually be set to a large negative value (-5e7) to help solver convergence. Syntax: “eigen_solver_shift {value}”.

**camp_solver_tol.** This is an eigen solver parameter that can just be set to 1e-4 as a default. Syntax: “camp_solver_tol {value}”.

**camp_max_iter.** This is an eigen solver parameter that can just be set to 5000 as a default. Syntax: “camp_max_iter {integer}”.

### 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
```
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 physics

Each scenario must specify the physics in the format: “physics {string}”. Table 3 lists the supported physics, a brief description, and which physics code can be used. In the table “SD” is sierra_sd and “PA” is plato_analyze.

<table>
<thead>
<tr>
<th>Physics</th>
<th>Description</th>
<th>Valid Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>steady_state_mechanics</td>
<td>Static solution with simple linear elasticity</td>
<td>SD, PA</td>
</tr>
<tr>
<td>steady_state_thermal</td>
<td>Steady state heat conduction</td>
<td>PA</td>
</tr>
<tr>
<td>steady_state_thermomechanics</td>
<td>Static mechanical solution with heat conduction</td>
<td>PA</td>
</tr>
<tr>
<td>modal_response</td>
<td>Modal analysis</td>
<td>SD</td>
</tr>
</tbody>
</table>

3.4.2 dimensions

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

3.4.3 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.

3.4.4 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.

3.4.5 minimum_ersatz_material_value

Within each scenario the user may specify the minimum density value that can exist in the design using the format: “minimum_ersatz_material_value {value}”.

3.4.6 tolerance

Within each scenario the user may specify the physics linear solver tolerance using the format: “tolerance {value}”.

3.4.7 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”.

3.4.8 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.

3.4.9 weight_mass_scale_factor

This is a SierraSD-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 parameter is: “weight_mass_scale_factor {value}”.

3.4.10 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.

3.5 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:

```plaintext
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” and “weighted_sum”.

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 specified 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 SierraSD 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. This 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 and multi-constraint problems are implemented but not considered a fully supported feature yet. However, you can run multi-constraint problems if you use the “mma” or “ksal” optimizers. 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 is the criterion that 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”, and “force”. Table 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}”. SierraSD uses ids to identify sidesets and nodesets and Plato Analyze uses names to identify sidesets. Therefore, sidesets must be named when using Plato Analyze.

3.7.4 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}”.
Table 4: Description of supported load types.

<table>
<thead>
<tr>
<th>Load Type</th>
<th>Description</th>
<th>Valid Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>traction</td>
<td>Arbitray direction traction load on a surface. Specify traction component values separated by spaces (e.g. 0.2 2e-3 200 in three dimensions).</td>
<td>SD, PA</td>
</tr>
<tr>
<td>pressure</td>
<td>Surface load normal to the surface. Specified by a single scalar value.</td>
<td>SD, PA</td>
</tr>
<tr>
<td>force</td>
<td>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)</td>
<td>SD</td>
</tr>
<tr>
<td>acceleration</td>
<td>Body load applied to the whole model.</td>
<td>SD</td>
</tr>
<tr>
<td>uniform_surface_flux</td>
<td>Thermal surface load. Single value specifying the normal flux to the surface.</td>
<td>PA</td>
</tr>
</tbody>
</table>

SierraSD 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}”. SierraSD 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}”. SierraSD 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 “temp” (temperature), “dispx” (displacement in the x direction), “dispy” (displacement in the y direction), and “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 disp x disp y disp z”. 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 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 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.

3.9.1 material
Each block must specify a material using the format: “material {integer}”.

3.9.2 element_type
Each block can specify the element type using the format: “element_type {string}”. Table 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.

3.10 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:
Table 5: Description of supported element types.

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Description</th>
<th>Valid Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>tet4</td>
<td>First-order tet element.</td>
<td>SD, PA</td>
</tr>
<tr>
<td>hex8</td>
<td>First-order hex element.</td>
<td>SD</td>
</tr>
<tr>
<td>tet10</td>
<td>Second-order tet element.</td>
<td>SD</td>
</tr>
<tr>
<td>rbar</td>
<td>See SierraSD User’s Manual.</td>
<td>SD</td>
</tr>
<tr>
<td>rbe3</td>
<td>See SierraSD User’s Manual.</td>
<td>SD</td>
</tr>
</tbody>
</table>

```
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.

### 3.10.1 material_model

Each material must specify a `material_model` using the format: “material_model {string}”. Table 6 lists the allowable material models and what physics code they can be used with.

Table 6: Description of supported material models.

<table>
<thead>
<tr>
<th>Material Model</th>
<th>Valid Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>isotropic_linear_elastic</td>
<td>SD, PA</td>
</tr>
<tr>
<td>orthotropic_linear_elastic</td>
<td>PA</td>
</tr>
<tr>
<td>isotropic_linear_thermal</td>
<td>PA</td>
</tr>
<tr>
<td>isotropic_linear_thermoelastic</td>
<td>PA</td>
</tr>
</tbody>
</table>

### 3.10.2 Isotropic Linear Elastic Properties

- **youngs_modulus.** Syntax: “youngs_modulus {value}”.
- **poissons_ratio.** Syntax: “poissons_ratio {value}”.
- **mass_density.** Syntax: “mass_density {value}”.

24
3.10.3 Orthotropic Linear Elastic Properties

\[ \text{youngs\_modulus\_x} \text{. Syntax: “youngs\_modulus\_x \{value\}”}. \]
\[ \text{youngs\_modulus\_y} \text{. Syntax: “youngs\_modulus\_y \{value\}”}. \]
\[ \text{youngs\_modulus\_z} \text{. Syntax: “youngs\_modulus\_z \{value\}”}. \]
\[ \text{poissons\_ratio\_xy} \text{. Syntax: “poissons\_ratio\_xy \{value\}”}. \]
\[ \text{poissons\_ratio\_xz} \text{. Syntax: “poissons\_ratio\_xz \{value\}”}. \]
\[ \text{poissons\_ratio\_yz} \text{. Syntax: “poissons\_ratio\_yz \{value\}”}. \]
\[ \text{shear\_modulus\_xy} \text{. Syntax: “shear\_modulus\_xy \{value\}”}. \]
\[ \text{shear\_modulus\_xz} \text{. Syntax: “shear\_modulus\_xz \{value\}”}. \]
\[ \text{shear\_modulus\_yz} \text{. Syntax: “shear\_modulus\_yz \{value\}”}. \]
\[ \text{mass\_density} \text{. Syntax: “mass\_density \{value\}”}. \]

3.10.4 Isotropic Linear Thermal Properties

\[ \text{thermal\_conductivity} \text{. Syntax: “thermal\_conductivity \{value\}”}. \]
\[ \text{mass\_density} \text{. Syntax: “mass\_density \{value\}”}. \]
\[ \text{specific\_heat} \text{. Syntax: “specific\_heat \{value\}”}. \]

3.10.5 Isotropic Linear Thermoelastic Properties

\[ \text{thermal\_conductivity} \text{. Syntax: “thermal\_conductivity \{value\}”}. \]
\[ \text{youngs\_modulus} \text{. Syntax: “youngs\_modulus \{value\}”}. \]
\[ \text{poissons\_ratio} \text{. Syntax: “poissons\_ratio \{value\}”}. \]
thermal_expansivity. Syntax: “thermal_expansivity {value}”.

reference_temperature. Syntax: “reference_temperature {value}”.

mass_density. Syntax: “mass_density {value}”.

3.11 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.

3.11.1 General Optimization Parameters

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

optimization_algorithm. This parameter specifies which optimization algorithm will be used. The syntax is: “optimization_algorithm {string}”. Currently supported options are “oc” (Optimality Criteria), “mma” (Method of Moving Asymptotes), “ksbc” (Kelley Sachs Bound Constrained), “ksal” (Kelley Sachs Augmented Lagrangian), “rol_linear_constraint”, “rol_bound_constrained”, and “rol_augmented_lagrangian”. “oc”, “ksal”, “rol_linear_constraint”, and “rol_augmented_lagrangian” require a constraint in the optimization problem, however “ksbc”, “mma”, and “rol_bound_constrained” do not (Note: typical implementations of “mma” require a constraint but the one in Plato allows for use on problems without a constraint). “oc” is typically only used for simple linear statics or linear thermal type problems. “rol_linear_constraint” can be used on any
problem with a single, linear constraint. “ksbc” and “rol_bound_constrained” are roughly equivalent and likewise “ksal” and “rol_augmented_lagrangian” are roughly equivalent.

**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.

**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.

**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.

**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}”.

**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™ model and displayed in the graphics window.

**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.

**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}”.

27
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.

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.

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.

3.11.2 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 which 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.

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.
**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.

**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 use “filter_radius_absolute” rather than “filter_radius_scale”. When determining what filter radius to use with the helmholtz filter you can use this relationship between the kernel filter radius and the helmholtz filter radius: kernel_filter_absolute_radius = 2*sqrt(3)*helmholtz_filter_absolute_radius.

**projection_type.** This parameter specifies what type of projection method will be used to try to force density values to 0 and 1. The syntax is: “projection_type {string}”. Valid projection types are “tanh”, and “heaviside”.

**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}”.

**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}”.

**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}”.

**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}”.

**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}”.
**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}”.

**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 in most cases. The exception to this is 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}”.

### 3.11.3 Restart Parameters

**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 will typically be a result file 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.

**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}”.

**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}”.

### 3.11.4 Prune and Refine Parameters

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

**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
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.

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}”.

3.11.5 Shape Optimization Parameters

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.

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}”.

3.11.6 Optimizer Parameters

mma_move_limit. This parameter specifies the move limit value for the Method of Moving Asymptotes optimization algorithm. The syntax is: “mma_move_limit {value}”.

mma_asymptote_expansion. This parameter specifies the asymptote expansion factor used in the Method of Moving Asymptotes optimization algorithm. The syntax is: “mma_asymptote_expansion {value}”.

31
**mma_asymptote_contraction.** This parameter specifies the asymptote contraction factor used in the Method of Moving Asymptotes optimization algorithm. The syntax is: “mma_asymptote_contraction {value}”.

**mma_max_sub_problem_iterations.** This parameter specifies the maximum number of sub-problem iterations that will occur in the Method of Moving Asymptotes optimization algorithm. The syntax is: “mma_max_sub_problem_iterations {integer}”.

**mma_control_stagnation_tolerance.** This parameter specifies the control stagnation tolerance used in the Method of Moving Asymptotes optimization algorithm. The syntax is: “mma_control_stagnation_tolerance {value}”.

**mma_objective_stagnation_tolerance.** This parameter specifies the objective stagnation tolerance used in the Method of Moving Asymptotes optimization algorithm. The syntax is: “mma_objective_stagnation_tolerance {value}”.

**mma_use_ipopt_sub_problem_solver.** This parameter specifies whether to use the interior point sub-problem optimizer in the Method of Moving Asymptotes optimization algorithm rather than KSAL which is the default. The syntax is: “mma_use_ipopt_sub_problem_solver {Boolean}”.

**hessian_type.** This parameter specifies the type of hessian approximation that will be used. The syntax is: “hessian_type {string}”. Valid options are “lbfgs”.

**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}”.

**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}”.

32
3.12 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:

\[
\text{begin output} \\
\hspace{1em} \text{service 2} \\
\hspace{2em} \text{data temperature} \\
\text{end output}
\]

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

3.12.1 service

Each output block must specify the service providing the output using the syntax: “service \{integer\}”.

3.12.2 data

Each output block must specify the data to output using the syntax: “data \{string\} {...\}”. Table 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.

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
<th>Valid Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>displx</td>
<td>X displacement</td>
<td>SD, PA</td>
</tr>
<tr>
<td>dispy</td>
<td>Y displacement</td>
<td>SD, PA</td>
</tr>
<tr>
<td>dispz</td>
<td>Z displacement</td>
<td>SD, PA</td>
</tr>
<tr>
<td>vonmises</td>
<td>Vonmises stress.</td>
<td>SD, PA</td>
</tr>
<tr>
<td>temperature</td>
<td>Temperature</td>
<td>PA</td>
</tr>
</tbody>
</table>

3.12.3 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 only applies to the Plato Analyze physics service.
3.13 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:

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

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

3.13.1 name

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