Plato

Release 0.0

Plato Team

Aug 03, 2021

CONTENTS

Note: These pages are a pre-release version. They are under active development.

These Github pages are meant to serve as the documentation for **Plato Engine** and its associated repositories. These repositories can be found at the Plato Engine GitHub account.

These pages are built using the Plato Docs GitHub repository.

ONE

CONTENTS

1.1 Description

The **PLATO Engine** computer program serves as a collaborative testbed rich in light-weight synthesis tools for optimization-based design. PLATO Engine is a research code designed to facilitate collaboration with academia, labs and industries by providing interfaces for plug-n-play insertion of synthesis technologies in the areas of modeling, analysis and optimization. Currently, PLATO Engine offers a set of light-weight tools for finite element analysis, linear- and nonlinear-programming, and non-gradient based optimization. The PLATO Engine program is designed to run on high-performance computers.

1.1.1 Application

The PLATO Engine testbed is designed to support research in the area of optimization-based design on high-performance computing systems. The PLATO Engine testbed is used to explore interoperability with several analysis, modeling and optimization tools. The testbed is also used to test the viability of these analysis, modeling and optimization tools for the solution of optimization-based design problems.

1.1.2 Approach

PLATO Engine is intended to serve as a collaborative testbed. It is designed to enable intercommunication of modeling, analysis and optimization data using a Multiple Program, Multiple Data (MPMD) parallel programming model. The MPMD model allows multiple, independent programs/executables to share data in-memory. The optimization algorithms orchestrate the execution and communication between multiple analysis codes and aggregates their contributions to create designs that meet multiple performance criteria.

1.1.3 High Performance Computing

PLATO Engine has been designed for MPMD parallel executions. It also targets Single Program, Multiple Data (SPMD) parallel programming model; however, the SPMD model is not heavily used by the targeted PLATO applications. PLATO Engine is also performance-protable, which allow it to optimally perform in current and next-generation computing architectures.

1.1.4 Required Libraries

Trilinos library (provides Epetra, Seacas and STK): https://github.com/trilinos/trilinos

Omega_h library (provides mesh metadata): https://github.com/SNLComputation/omega_h

Netcdf library (provides I/O libraries): https://www.unidata.ucar.edu/software/netcdf/

AMGX library (provides GPU linear solver): https://github.com/NVIDIA/AMGX

Lapack library (provides linear algebra libraries): http://www.netlib.org/lapack/

Boost library (provides C++ source libraries): https://www.boost.org/

1.1.5 Hardware Requirements

Tested compilers are g++ `4.7.2`, g++ `5.4.0`, and g++ `7.2.0` compilers. Tested OS include Linux and Mac. RAM requirements are problem size dependent.

TODO: is the following still true?

Note: Currently, PLATO Analyze only runs on Graphics Processing Units (GPUs).

1.1.6 Contributing

Please open a GitHub issue to ask a question, report a bug, request features, etc. If you'd like to contribute, please fork the repository and use a feature branch. Make sure to follow the team's coding style policies. Pull requests are welcomed.

TODO: create coding style policies page

1.1.7 User Support

Users are welcomed to submit questions via email to plato3D-help@sandia.gov.

1.2 Plato Repositories

These repositories can be found at the Plato Engine GitHub account.

GitHub Repository	Description
AMGX	Distributed multigrid linear solver library on GPU
exo2obj	Convert from an exodus volume mesh to an obj surface mesh
nvcc_wrapper	Wrapper shell script for NVIDIA nvcc to better conform to host compiler command line arguments
Plato3D	
Plato3D_backend	
platoanalyze	Fast physics and gradients for multidisciplinary analysis and optimization
platodocs	Documentation for Plato
platoengine	Plato Engine - Optimization-Based Design Ecosystem
spack	The spack configuration repository for building Plato.
tests	
use_cases	Collection of Plato Engine use cases
verification	verification and validation problems

1.3 Documentation For Plato

This section describes the process for cloning, editing, and commiting to these GitHub pages, i.e., these web pages. These pages are created using reStructuredText in Sphinx.

1.3.1 Installing Sphinx and Other Packages

The sphinx documentation is set up to work with Python3 packages. The user will need to install pip, virtualenv, and Sphinx.

Installing Some Packages

The following code block installs some necessary packages that are probably already installed if the user has installed Plato.

```
$ sudo apt-get update
$ sudo apt-get -y upgrade
$ sudo apt-get -y install build-essential curl git gfortran python python-dev vim tcl␣
↪environment-modules unzip csh python3-distutils
```

Installing pip

Sphinx packages are published on the Python Package Index. The preferred tool for installing packages from *PyPI* is **pip**. Ubuntu users have to manually install **pip** for Python3.

```
$ sudo apt-get -y install python3-venv python3-pip
```

Installing virtualenv

To install the virtual environment to manage Python packages:

```
$ python3 -m pip install --user virtualenv
```

Installing Latex Packages

The user will have to install certain latex packages to enable building a pdf version of Plato Docs.

```
$ sudo apt-get -y install texlive-latex-base texlive-fonts-recommended texlive-fonts-
→extra texlive-latex-extra latexmk
```

Installing Sphinx

Finally, the user will have to install Sphinx from a virtual environment in order to get the most recent version:

```
$ python3 -m venv ~/.venv
$ source ~/.venv/bin/activate
```

Check that the version of Python used in the virtual environment is 3.0.0 or later.

```
(.venv) $ python --version
```

Install Sphinx:

```
(.venv) $ python -m pip install sphinx
```

Check that the Spinx version is 4.0.0 or later.

```
(.venv) $ sphinx-build --version
```

1.3.2 Cloning Plato Docs

Navigate to a directory where /platodocs/ and /platodocs-build/ can be stored.

```
$ git clone https://github.com/platoengine/platodocs.git

$ mkdir platodocs-build

$ cd platodocs-build

$ git clone https://github.com/platoengine/platodocs.git html

$ cd html

$ git checkout -b gh-pages remotes/origin/gh-pages
```

The directory structure should look something like this:

```
topfolder\
|-- platodocs\                 <-- release branch
|   |-- docs\
|   |   |-- source\
|   |   |   |-- images\
|   |   |   |-- _static\
|   |   |   |-- _templates\
|   |   |   |-- conf.py     <-- configuration information to sphinx (extensions, etc.)
|   |   |   |-- index.rst   <-- index source file
|   |   |   `-- other files
|   |   |-- make.bat
|   |   `-- Makefile
|   |-- platodocs.pdf
```

```
|    `-- README.md
|
`-- platodocs-build\
    |-- doctrees\
    `-- html\            <-- gh-pages branch
        |-- _images\
        |-- _source\
        |-- _images\
        |-- index.html  <-- main html file (can open with browser)
        |-- .nojekyll   <-- tells github to not use a jekyll theme
        |-- README.md
        `-- other files
```

The existing webpages can be viewed by opening `/platodocs-build/html/index.html` in a web browser or they can be viewed in the pdf version from `/platodocs/platodocs.pdf`.

1.3.3 Editing Plato Docs

Editing Existing Pages:

Documentation can be edited from the source folder (`/platodocs/docs/source/`). The files in this directory are written using reStructuredText (.rst). These files can be opened in any text editor and edited using syntax found here.

Adding New Pages:

When adding a new page to the html web pages, the user can create another .rst file in the source directory (`/platodocs/docs/source/`). This file is added to the website by adding the file name to the contents in `/platodocs/docs/source/index.rst` or another file if the user does not want the file to show up on the home page.

For example, this page is built based on the `documentation.rst` and was added to the web pages by using "toctree" directive in `index.rst`. An example of this is shown below.

`/platodocs/docs/source/index.rst`

```
.. toctree::
:maxdepth: 2

   description
   githubrepositories
   documentation        <-- adding documentation.rst
```

Updating HTML and PDF in Ubuntu:

The web pages (`.html` files) can be updated to represent the changes in the `.rst` files.

First, the user should navigate to `/platodocs/docs/`. Then they can run the following command in a virtual environment (.venv) from the terminal. For directions to get into a virtual environment, go to *Virtual Environment*.

```
(.venv) $ make html
```

Changes can be viewed by opening `/platodocs-build/html/index.html` in a web browser.

The user can update the pdf version of the documentation by running the following command at from `/platodocs/docs/` from the terminal.

```
(.venv) $ make latexpdf
```

The changes can be viewed by opening `/platodocs/platodocs.pdf`.

1.3.4 Publishing Documentation

In order to publish changes to the Plato Docs GitHub Pages the user will have to Git Push changes to the build repository (`gh-pages` branch). It is also wise to Git Push the changes in the source files (`release` branch). See instruction on *pushing and pulling* to both branches of Plato Docs.

1.3.5 Pushing and Pulling

The process for pushing to and pulling from the `release` branch of Plato Docs (`/platodocs/`) is standard. This is the branch used for the source files of Plato Docs.

In `/platodocs/`:

```
$ git push
```

or

```
$ git pull
```

The user will have to specify the branch when pushing to or pulling from the `gh-pages` branch of Plato Docs (`/platodocs-build/`). This is the branch used for the build files of Plato Docs.

In `/platodocs-build/html/`:

```
$ git push origin gh-pages
```

or

```
$ git pull origin gh-pages
```

1.3.6 Virtual Environment

Sphinx build commands (`$ make html` or `$ make latexpdf`) will have to done from a Python3 virtual environment. To create a virtual environment:

```
$ python3 -m venv ~/.venv
```

Note that the virtual environment only has to be created once.

Once the virtual environment is created it can be sourced:

```
$ source ~/.venv/bin/activate
```

Check that default Python version is 3.0.0 or later and Sphinx is 4.0.0 or later:

```
(.venv) $ python --version      <-- should be 3.0.0 or later
(.venv) $ sphinx-build --version <-- should be 4.0.0 or later
```

1.4 Input Deck Reference

1.4.1 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 toplogy 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 con- straints are, what physics codes will be used to evaluate the objectives and constraints, what optimization algorithm to use, and variuos other parameters. The geoemtry is gen erally 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.

1.4.2 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 optimziation problems are defined in Plato. This section will briefly introduce thesebuilding 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 use 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 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. Currently, Plato only allows a single constraint in a given optimization problem.

1.4.3 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 **TODO:** "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 1.1 shows the different types of values and a description of each.

Table 1.1: 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.

Fig. 1.1 shows 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 1
   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". 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.
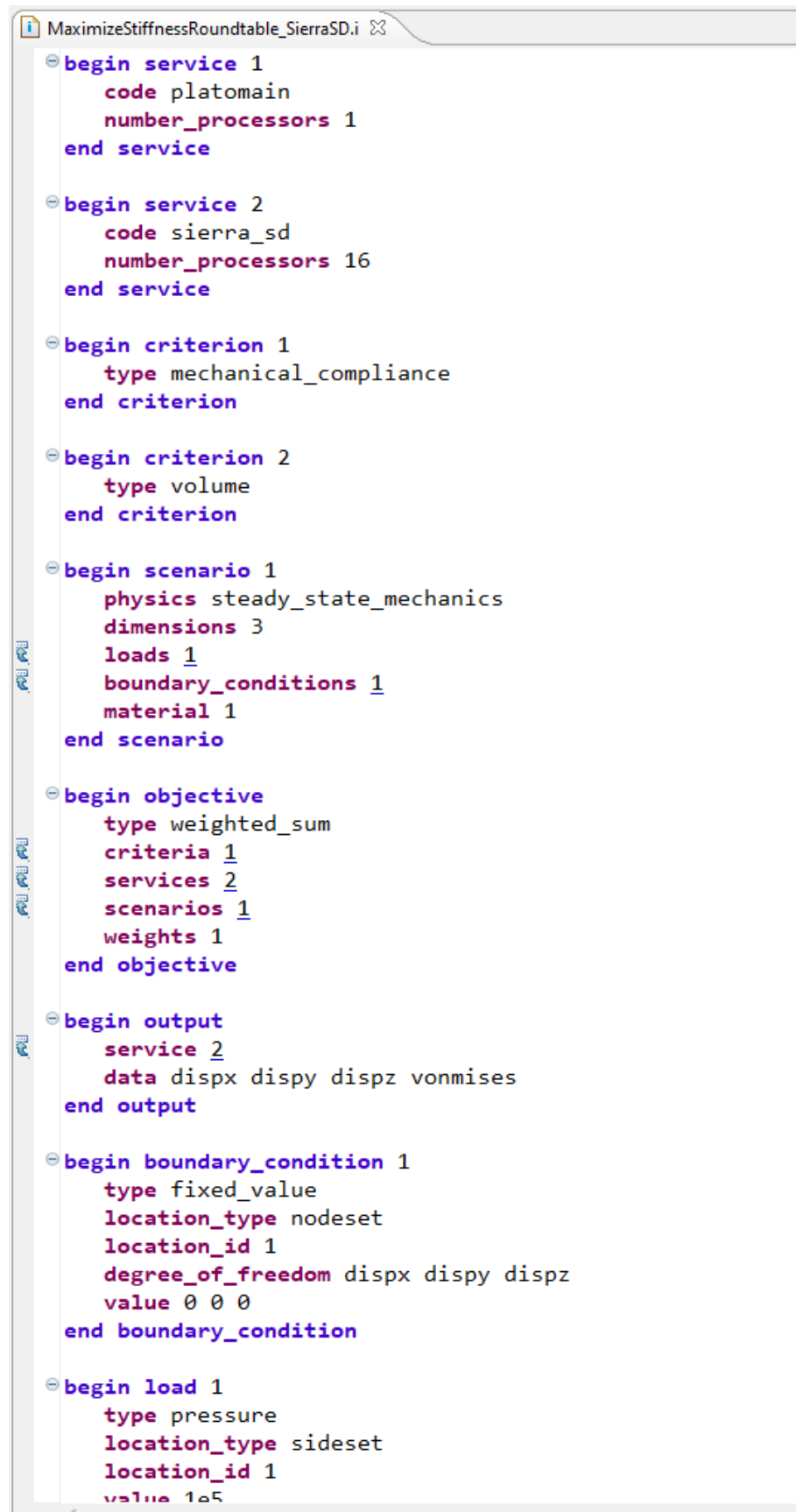
3.2.1 code

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", and "plato analyze".

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}{. . .}". Typically, you will only specify one device id.

```
MaximizeStiffnessRoundtable_SierraSD.i ⊠

begin service 1
    code platomain
    number_processors 1
end service

begin service 2
    code sierra_sd
    number_processors 16
end service

begin criterion 1
    type mechanical_compliance
end criterion

begin criterion 2
    type volume
end criterion

begin scenario 1
    physics steady_state_mechanics
    dimensions 3
    loads 1
    boundary_conditions 1
    material 1
end scenario

begin objective
    type weighted_sum
    criteria 1
    services 2
    scenarios 1
    weights 1
end objective

begin output
    service 2
    data dispx dispy dispz vonmises
end output

begin boundary_condition 1
    type fixed_value
    location_type nodeset
    location_id 1
    degree_of_freedom dispx dispy dispz
    value 0 0 0
end boundary_condition

begin load 1
    type pressure
    location_type sideset
    location_id 1
    value 1e5
```

Fig. 1.1: Example Plato input deck

       Chapter 1. Contents

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.

3.3 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 an arbitrary number of 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 1.2 lists a description of the allowable types and what physics code they can be used with.

Table 1.2: 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. 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.	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 will maximize heat conduction.	PA
stress_and_mass	Used for doing stress-constrained mass minimization problems.	SD, PA
stress_p-norm	Superscript p used to compute the norm of the stress.	PA
volume	The volume of the current design.	PA
mass	The mass of the current design.	PA
CG_x	X component of the center of gravity of the current design.	PA
CG_y	Y component of the center of gravity of the current design.	PA
CG_z	Z component of the center of gravity of the current design.	PA
Ixx	Mass moment of inertia about the x axis.	PA
Iyy	Mass moment of inertia about the y axis.	PA
Izz	Mass moment of inertia about the z axis.	PA
Ixy	XY product of inertia.	PA
Iyz	YZ product of inertia.	PA
Ixz	XZ product of inertia.	PA
flux_p-norm	Superscript p used to compute the norm of the heat flux.	PA

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 paramter 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 paramter is: "criterion weights {value}{...}".

3.3.4 Criterion Parameters Related to Stress-constrained Mass Minimization Problems

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.**

3.4 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 an arbitrary number of scenario blocks. The following tokens can be specified in any order within the scenario block.

3.4.1 pysics

Each scenario must specify the physics in the format: "physics {string}". Table 1.3 lists the supported physics, a brief description, and which physics code can be used.

Table 1.3: 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

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 paramter 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}".

3.5 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" 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 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 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. Currently, the user is only allowed to specify one contraint in the input deck. The constraint block begins and ends with the tokens "begin constraint {integer}" and "end constraint". The integer following "begin constraint" will eventually be used to support multiple constraints in a single Plato run. 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 abosolute 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 1.4 lists a description of the allowable types of loads and the physics code they can be used with.

Table 1.4: 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

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}". 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 boudnary conditions are applied on sidesets.

3.8.3 location_name

Each boundary condition must specify a application location either by name or id de- pending 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 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 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 1.5 lists a description of the allowable element types and what physics code they can be used with.

Table 1.5: 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 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.

3.10.1 material_model

Each material must specify a material model using the format: "material model {string}". Table 1.6 lists the allowable material models and what physics code they can be used with.

Table 1.6: Description of supported element types

Material Model	Valid Code
isotropic_linear_elastic	SD, PA
orthotropic_linear_elastic	PA
isotropic_linear_thermal	PA
isotropic_linear_thermoelastic	PA

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}".

3.10.3 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}".

3.10.4 Isotropic Linear Thermal Properties

thermal conductivity. Syntax: "thermal conductivity {value}".

mass density. Syntax: "mass density {value}".

specific heat. Syntax: "specific heat {value}".

3.10.5 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 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), and "ksal" (Kelley Sachs Augmented Lagrangian). "oc", "mma", and "ksal" require a constraint in the optimization problem, however "ksbc" does not. "oc" is typically only used for simple linear statics or linear thermal type problems. For other physics one of the other optimization algorithms should be used.

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 frequncy {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 CubitTMmodel 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}".

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. How- ever, 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", "kernel then heaviside", and "kernel then tanh".

filter heaviside min. Determines the initial value of the heaviside steepness parameter for kernel then heaviside and kernel then tanh. 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 funciton. The syntax is: "filter heaviside min {value}".

filter heaviside max. Determines the maximum value of the heaviside steepness parameter for kernel then heaviside and kernel then tanh. 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 funciton. 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. 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. 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. 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. 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 paramter 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 {integer}".

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 correclty. 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 Optimization 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}".

(**TODO:** remove everything but this) **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}".

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:

```
begin output
    service 2
    data temperature
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 1.7 lists the allowable outputs and what physics code they can be used with.

Table 1.7: Description of supported output

Output	Description	Valid Code
dispx	X displacement	SD, PA
dispy	Y displacement	SD, PA
dispz	Z displacement	SD, PA
vonmises	Von Mises stress	SD, PA
temperature	Temperature	PA

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:

```
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}".

3.14 Mesh

In this section, we show how to specify the paths block. This block is only necessary if you need to point to non-standard executables when doing Plato runs. An example would be if you were developing in the Plato code and wanted to test something using your own builds of either PlatoMain or the physics codes. In this case you would need to tell Plato where to find the executables you want it to use. The paths block begins and ends with the tokens "begin paths" and "end paths". The following is an example paths block definition:

```
begin paths
    code platomain /directory/to/my/own/PlatoMain/PlatoMain
    code sierra_sd /directory/to/a/different/SierraSD/plato_sd_main
end paths
```

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

3.14.1 code

The alternate location to a code can be specified using the syntax: "code {string}{string}". The first string parameter is the name of the code you are providing an alternative for. Valid options are "platomain", "sierra sd", "plato analyze", and "prune and refine". The second string parameter is the full path to the alternative code to be used.

1.4.4 References

1.5 Things to add

how to make repository

- numfig = True

- extenstions

- figure out weird section numbering from pdf version of documentation

INDICES AND TABLES

- genindex
- modindex
- search