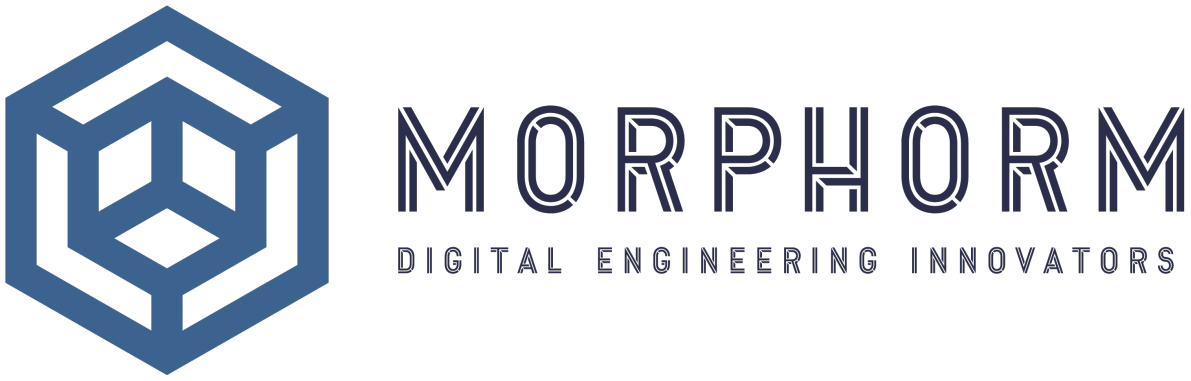
User Manual

Release 1.0.0

Morphorm

Jan 18, 2025

INTRODUCTION



These pages are meant to serve as the `User Manual` for the Morphorm software.

CHAPTER

ONE

INTRODUCTION

The *Introduction* chapter provides a brief description of the Morphorm engineering design application.

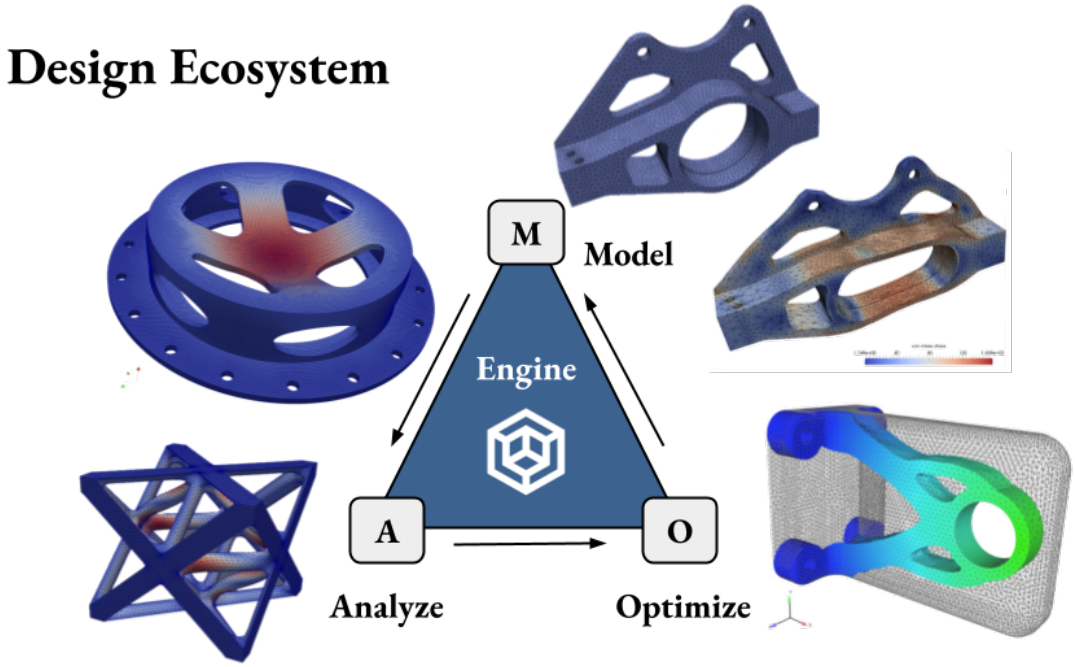


Fig. 1.1: The Morphorm® design ecosystem is comprised by four high level modules: (1) Analyze, (2) Model, (3) Optimize, and (4) Engine. The Analyze module predicts how a physical system interacts with its environment by numerically solving the partial differential equations governing the behavior of the system. The Model module constructs a geometry model of the physical system under investigation. The Optimize module applies optimization methods to optimize the performance, reliability, and efficiency of the physical system. The Engine module facilitates automated exchange of information at runtime between the analysis, modeling, optimization, and other third-party applications. Given multiple environments, i.e., boundary conditions, the Engine applies a Multiple-Program, Multiple-Data parallel model to facilitate concurrent evaluations of the physical systems.

1.1 Description

Morphorm® builds modeling, simulation, and optimization technologies to deliver near real-time design solutions. Our mission is to employ advanced scientific computing to solve complex science and engineering problems. We aspire to be the most innovative digital engineering company, where our clients can discover in near real-time optimized design solutions.

Our products exploit distributed memory parallel programming models and performance portability to perform large-scale calculations on heterogeneous and homogeneous computing systems. The Morphorm platform, Fig. 1.1, includes tools for multiphysics finite element analysis, design optimization, geometry modeling, uncertainty quantification and propagation, and much more.

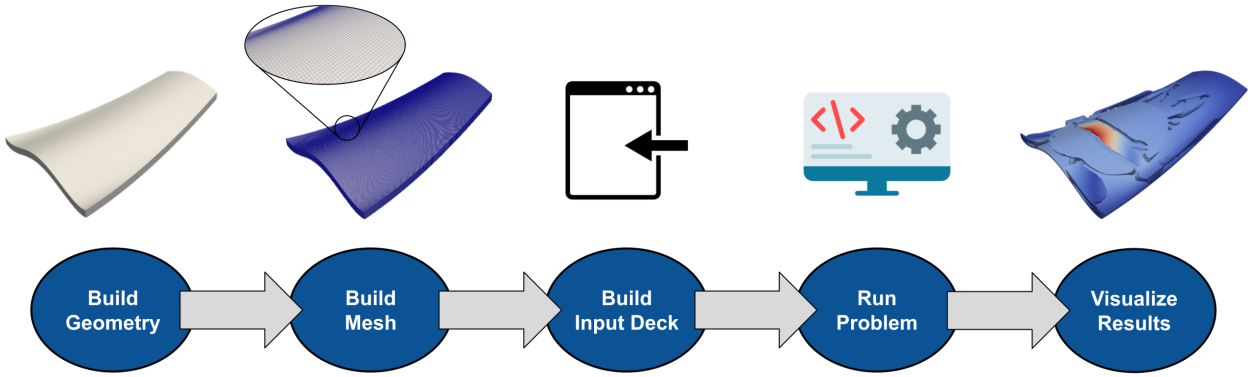


Fig. 1.1.1: Pictorial description of the Morphorm design workflow.

1.2 Workflow

The sequence of steps to setup and run a Morphorm study are described in Fig. 1.1.1. Setting up and running a Morphorm study is done through a sequence of steps, which is refer in this manuscript as a design workflow. The design workflow consists of the following steps:

- Construct the geometry
- Create the exodus mesh [Sjaardema]
- Write the input deck
- Launch the design study
- Postprocess the results

The initial geometry and exodus mesh files needed to run a Morphorm study are currently created outside the Morphorm environment. For example, the Cubit® geometry and mesh toolkit can be used to create the initial geometry and exodus mesh files. If the user has an existing geometry file, the geometry file can be uploaded to Cubit® to create the exodus mesh file. Cubit® also supports other mesh file formats for many commercial-off-the-shelf engineering software tools. For instance, an Abaqus input file can be converted into an exodus mesh file in Cubit®. The user is recommended to consult the Cubit user manual for more information about the mesh and geometry file formats supported in Cubit [Stimpson *et al.*]. The results from a Morphorm design study can be visualized and postprocessed with ParaView or any other commercial-off-the-shelf visualization tool that supports the exodus mesh file format. The `Morphorm User Manual` gives an in-depth presentation of the Morphorm input deck.

1.3 User Support

You can submit questions via email to help@morphorm.com.

CHAPTER

TWO

INPUT DECK

The *Input Deck* chapter describes the main components of the Morphorm input deck.

2.1 Summary

The Morphorm input deck contains the recipe for running a problem. The next sections will describe the general concepts used to define a Morphorm design study, combined with a description of the input options.

2.1.1 General Concepts

Simulation-driven design optimization problems require multiple inputs. The Morphorm input deck is designed to be general in the way it defines a simulation-driven design optimization problem. The input deck breaks the simulation-driven design optimization problem into fundamental pieces that are combined to define a design study. There are several building blocks upon which all simulation-driven design optimization problems are defined within a Morphorm input deck. The next section introduces these main building blocks and describes how they are used to define a design study.

Service

A *service block* describes the software executable responsible for computing quantities of interests needed for the design study. A typical Morphorm run requires two services, one service for the optimizer and another for the multi-physics analysis application. The optimizer updates the design variables based on the most recent *objective* and *constraint* function evaluations. The analysis application simulates the physics and evaluates the performance criteria, i.e., objective and constraint functions. Each *service block* is defined once in the input deckand can be later referenced by other blocks in the input deck.

The following input deck snippet shows a typical *service block* setup:

```
begin service 1
  code analyze
  number_processors 1
  number_ranks 1
end service
```

Scenario

A *scenario block* describes the problem setup for the multi-physics analysis. It includes parameters such as the physics type and boundary conditions for the analysis. As with the *service block*, scenarios are defined once in the input deck so that they can be later referenced by other blocks in the input deck.

The following input deck snippet shows a typical *scenario block* setup:

```
begin scenario 1
  physics steady_state_mechanics
  service 2
  dimensions 3
  loads 1 2
  boundary_conditions 1
end scenario
```

Multiple scenario blocks may appear in the input deck based on the use case.

Load

A *load block* describes the external forces defined for a multi-physics analysis. It includes paramete4rs such as the force type, location type, location name (`exodus` side set) , and the force value. At least one *load block* must be defined in the input deck for a design study. However, multiple load blocks may appear in the input deck based on the use case.

The following input deck snippet shows a typical *load block* setup:

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

Boundary Condition

A *boundary_condition block* describes the boundary conditions, e.g., Dirichlet boundary conditions, defined for a multi-physics analysis. It includes parameters such as the boundary condition type, the location type, the location name, the local degree of freedom where the boundary condition is enforced, and the value. At least one *boundary condition block* must be defined in the input deck for a design study. However, multiple boundary condition blocks may appear in the input deck based on the use case.

The following input deck snippet shows a typical *boundary_condition block* setup:

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

Material

A *material block* describes the material constitutive model used to predict the response of the material under applied forces. It includes parameters such as the material model type and the values assigned to the material properties of the model. At least one material block must be defined in the input deck for a design study. However, multiple material blocks may appear in the input deck based on the use case.

The following input deck snippet shows a typical *material block* setup:

```
begin material 1
  material_model isotropic_linear_thermal
  thermal_conductivity 210
```

(continues on next page)

(continued from previous page)

```
  mass_density 2703
  specific_heat 900
end material
```

Block

A *block block* is used to assign a meterial to a geometric region, i.e., `exodus` element block. The geometric region is defined by a collection of elements grouped into a single entity that shares common traits such as the element type and material properies. The *block block* includes the identifier of the *material block* assigned to the region and the name of the `exodus` element block. At least one *block block* must be defined in the input deck for a design study. However, multiple block blocks may appear in the input deck based on the use case.

The following input deck snippet shows a typical *block block* setup:

```
begin block 1
  material 1
  name design_domain
end block
```

Criterion

A *criterion block* is used to define a design criterion evaluating the performance of the design. For example, in a structural topology optimization problem, the problem formulation seeks to minimize the structural compliance given a mass target. In this example, the structural compliance and the mass functions are considered design criteria. The criteria are defined independently from the objective and constraint functions so that they can be defined only once and later referenced repeatedly as needed in the *objective* and *constraint* blocks.

The following input deck snippet shows a typical *criterion block* setup:

```
begin criterion 1
  type volume
  location_name block_1
end criterion
```

Multiple criterion blocks may appear in the input deck based on the use case.

Objective

An *objective block* is used to define a real-valued function that is minimized or maximized by the optimizer to meet a targeted performance. It is composed of one or more *criterion block* identifiers, the identifier of the *service block* providing the design criteria evaluations, and the identifier of the *scenario block* describing the physics and boundary conditions under which the objective function is being considered. The input deck defines only one objective function. However, the objective can be composed of multiple sub-objectives, each sub-objective defined by a *criterion*. The sub-objectives can be weighted based on their relevance to the design problem.

The following input deck snippet shows a typical *objective block* setup:

```
begin objective
  type single_criterion
  criteria 1
  services 2
  scenarios 1
  weights 0.5
end objective
```

Constraint

A *constraint block* defines limitations on a design performance metric. For example, in a stress-constrained mass minimization problem, a stress limit is set a each material point so that the stress experience at any point in the design is less than the targeted stress limit. Similar to an objective, a constraint is composed of the *criterion block* identifier assigned to the *constraint block*, the identifier of the *service block* providing the evaluation, and the identifier of the *scenario block* describing the physics and boundary conditions under which the constraint is being measured.

The following input deck snippet shows a typical *constraint block* setup:

```
begin constraint 1
  criterion 2
  relative_target 0.3
  type less_than
  service 1
end constraint
```

Multiple constraint blocks may appear in the input deck based on the use case.

Study

A *study block* is used to specify the numerical methods used for a design study and their respective parameter values. At least one study block must be defined in the input deck.

The following input deck snippet is a typical *study block* setup:

```
begin study
  optimization_algorithm mma
  max_iterations 100
  filter_type kernel
  filter_radius 2.0
end study
```

Variables

A *variables block* is used to describe attributes assigned to the optimization variables. At least one `variables block` must be defined when using one of the following studies:

- *multidimensional parameter study*
- *design of experiments*
- *surrogate-based design optimization*
- *shape optimization*

The following input deck snippet is a typical *variables block* setup:

```
begin variables 1
  numvars 2
  type continuous_design
  initial_points 5e9   0.35
  lower_bounds   1e9   0.30
  upper_bounds   1e10  0.40
  descriptors     E     nu
end variables
```

Multiple *variables block* may appear in the input deck.

Prune

A *prune block* is used to specify the parameters for the *prune and refine* tool. The following input deck snippet is a typical *prune block* setup:

```
begin prune
  number_of_refines 1
  number_buffer_layers 1
  prune_threshold 0.6
end prune
```

Output

An *output block* specifies the output quantities of interests. It includes the identifier of the *service block* providing the output quantities of interests and the *identifiers* for the output quantities of interests.

The following input deck snippet is a typical *output block* setup:

```
begin output 1
  service 2
  data temperature
end output
```

Multiple output blocks may appear in the input deck based on the use case.

Writevars

A *writevars block* is used to request analysis application to write one or more *output quantities of interest* to a text file. The following input deck snippet is a typical *writevars block* setup:

```
begin writevars 1
  variable dispx
  entity_ids 8 13 98
end writevars
```

A text file will be created for each of the output quantity of interest requested in the *variable* keyword.

Mesh

A *mesh block* specifies the mesh file describing the geometric model used for a design study. It includes the *name of the mesh file*. The following input deck snippet is a typical *mesh block* setup:

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

The *mesh block* is a mandatory input.

Linear Solver

A *linear_solver block* is used to specify parameters of the linear solver. It includes parameters such as the *type of linear solver*, the *maximum number of iterations*, and the *stopping convergence tolerance*. Default values are provided for these parameters. Therefore, the *linear solver block* does not need to be defined. . .

The following input deck snippet is a typical *linear_solver block* setup:

```
begin linear_solver 1
  type tpetra
  max_iterations 500
  tolerance 1e-12
end linear_solver
```

Multiple linear solver blocks may appear in the input deck based on the use case.

Newton-Raphson

A *newton_raphson block* specifies parameters of the Newton-Raphson solver. Default values are provided for the Newton-Raphson solver parameters. Therefore, the *newton_raphson block* is an optional input.

The following input deck snippet is a typical *newton_raphson block* setup:

```
begin newton_raphson
  residual mechanical
  max_iterations 10
  residual_tolerance 1e-12
  increment_tolerance 1e-12
end newton_raphson
```

Multiple Newton-Raphson solver blocks may appear in the input deck based on the use case.

2.2 Syntax

The input deck is organized into different groupings of commands called *blocks*. The main blocks in the input deck are: *service*, *scenario*, *load*, *boundary_condition*, *material*, *block*, *mesh*, *criterion*, *objective*, *constraint*, and *study*. These blocks were introduced in the *Input Deck* section. The optimization specific blocks, *criterion*, *objective*, and *constraint*, are must be defined for optimization studies. However, the *criterion* blocks may also be defined for an *analysis workflow*. Each block in the input deck contains one or more lines with keyword-value pairs. For example, `loads 1 5 6` is a keyword value pair where the keyword is `loads` and the value is defined by the 3 load block ids `1 5 6`. Table 2.2.1 shows the different value types supported in the input deck.

Table 2.2.1: 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.

Here are some keyword-value pair examples:

- **number_processors {integer}**: Indicates that the user should enter a single integer value for this parameter, e.g., `number_processors 16`.
- **loads {integer}{...}**: Indicates that the user should specify one or more integer values for this parameter separated by spaces, e.g., `loads 1 2 3`.
- **output_data {Boolean}**: Indicates that the user should specify `true` or `false` for this parameter, e.g., `output_data true`.
- **type {string}**: Indicates that the user should specify a string for this parameter, e.g., `type volume`.

- **load_case_weights {value}{...}**: Indicates that the user should specify one or more real values, e.g., `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 parser. In the following code-block:

```
begin service 2
  code analyze
  //number_processors 1
  number_processors 10
  number_ranks 1
end service
```

the `//number_processors 1` line is ignored by the input deck parser.

The next text snippet shows a typical Morphorm input deck for a structural topology optimization study:

```
 begin service 1
   code engine
   number_processors 1
   number_ranks 1
 end service

 begin service 2
   code analyze
   number_processors 1
   number_ranks 1
 end service

 begin criterion 1
   type mechanical_compliance
 end criterion

 begin criterion 2
   type volume
 end criterion

 begin scenario 1
   physics steady_state_mechanics
   service 2
   dimensions 3
   loads 1 2
   boundary_conditions 1
 end scenario

 begin objective
   type single_criterion
   criteria 1
   services 2
   scenarios 1
 end objective

 begin output
   service 2
   output_data true
```

(continues on next page)

(continued from previous page)

```
   data dispx dispy dispz stress vonmises
 end output

 begin boundary_condition 1
  type fixed_value
  location_type nodeset
  location_name ns_1
  degree_of_freedom dispx dispz dispy
  value 0 0 0
end boundary_condition

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

begin load 2
  type traction
  location_type sideset
  location_name ss_2
  value 0 -3e3 0
end load

begin constraint 1
  criterion 2
  relative_target 0.3
  type less_than
  service 1
end constraint

begin block 1
  material aluminum
  name body
end block

begin block 2
  material steel
  name bolt
end block

begin material aluminum
  material_model isotropic_linear_elastic
  poissons_ratio .33
  youngs_modulus 68e9
end material

begin material steel
  material_model isotropic_linear_elastic
  poissons_ratio .27
  youngs_modulus 193e9
```

(continues on next page)

(continued from previous page)

```
end material

begin study
  method topology
  optimization_algorithm oc
  max_iterations 10
end study

begin mesh
  name bb_2block.exo
end mesh
```

The next sections will describe each input deck block and input keywords supported by each block.

2.3 Services

A *service block* begins and ends with the tokens `begin service {string}` and `end service`, respectively. The string following the `begin service` expression specifies an identifier for the *service block*. Other blocks in the input deck will use this identifier to refer to the `service block`. The following is a typical `service block` setup:

```
begin service 1
   code engine
   number_processors 16
end service
```

The input deck can contain one or more service blocks. The first service of every input deck **must** be the `engine` service. The `engine` service provides the numerical methods used to perform design studies. Furthermore, the `engine` service is responsible for coordinating the exchange of information between the services at runtime.

Table 2.3.1 lists the keywords supported within a *service block*.

Table 2.3.1: Description of the keywords supported within a service block

Required/Optional	Keyword	Description
Required	*code*	specifies the code assigned to the service
Required	*number_processors*	specifies the number of processors assigned to the software application; i.e., *code*
Optional	*path*	specifies the path to the code executable
Optional	*device_ids*	specifies a list of device (GPU) identifiers
Optional	*cache_state*	enables the use of the cache state operation
Optional	*update_problem*	enables update problem operation
Optional	*update_problem_frequency*	specifies the rate at which the update_problem operation will be called

2.3.1 code

The `code` keyword is used to specify the code assigned to the service. The syntax for this keyword is:

```
code {string}
```

The supported options are: `engine`, `analyze`, and `prune`. The `engine` code provides the numerical methods used to perform design studies, e.g., gradient-based optimizers. The `analyze` code provides the numerical methods used to simulate physical systems. The `prune` code is used to perform *prune and refine operations*.

2.3.2 number_processors

The `number_processors` keyword is used to specify the number of processors assigned to the code used by a the service. The syntax for this keyword is:

```
number_processors {integer}
```

The `default` number of processors is set to 1.

2.3.3 device_ids

The `device_ids` keyword is used to specify the device identifiers associated with a Graphical Processing Unit (GPU) on the machine. The syntax for this keyword is:

```
device_ids {integer}{...}
```

Typically, only one device id needs to be provided, which by `default` is set to `0`.

2.3.4 cache_state

This keyword is used to specify whether to use the `cache_state` operation in a design study. The `cache_state` operation is used to inform the analysis code when to cache physical quantities of interests during a simulation-driven design study. This mechanism eliminates redundant calls to the linear solver.

This operation is available for when a third-party application is integrated into the Morphorm digital engineering ecosystem. The Morphorm analysis software already knows when to update the physical quantities of interests; thus, avoiding redundant calls to the linear solver.

The `cache_state` operation can be enabled using the following syntax:

```
cache_state {Boolean}
```

The `default` is set to `false`.

2.3.5 path

The `path` keyword is used to specify the full path to the code executable. The syntax for this keyword is:

```
path {string}
```

Usually, the Morphorm software knows where to find the executable. The `path` keyword is mostly available for workflow customization. For example, if a third-party application is integrated into the Morphorm digital engineering ecosystem, the *path* keyword provides a mechanism to inform the Morphorm software where to find the executable for the third-party application.

2.3.6 update_problem

The `update_problem` keyword is used to allow a service to call the update problem operation. The syntax for this keyword is:

```
update_problem {Boolean}
```

For example, the `update_problem` operation is enabled for optimization problems where the *augmented Lagrangian criterion* is used. At every major optimization iteration k, the augmented Lagrangian method solves a sequence of optimization problems. If an update frequency is provided, the `update_problem` operation will proceed to update the Lagrange multipliers and the penalty parameters at the rate set in the *update_problem_frequency* keyword. The `default` value is set to `false`.

2.3.7 update_problem_frequency

The `update_problem_frequency` keyword is used to specify the rate at which the service will call the *update_problem* operation. The syntax for this keyword is:

```
update_problem_frequency {integer}
```

The `default` frequency is set to 1.

Table 2.3.2: Description of the supported keywords within a scenario block.

Required/Optional	Keyword	Description
Required	*physics*	specifies the physics to be simulated
Required	*dimensions*	specifies the spatial dimensions of the problem
Required	*service*	specifies the identifier assigned to the *service block* running the simulation
Required	*boundary_conditions*	specifies the identifers assigned to the *boundary condition blocks* associated with the scenario
Required	*blocks*	specifies the identifiers assigned to the *block blocks* associated with the scenario, which must be set for a *multi-scenario problem*
Optional	*output*	specifies the identifiers assigned to the *output blocks* associated with the scenario, which must be set for a *multi-scenario problem*
Optional	*loads*	specifies the identifiers assigned to the *load blocks* associated with the scenario
Optional	*newton_raphson*	specifies the identifiers assigned to the *newton-raphson blocks* associated with the scenario
Optional	*linear_solver*	specifies the identifiers assigned to the *linear solver blocks* associated with the scenario
Optional	*material_penalty_exponent*	specifies the penalty exponent assigned to the ersatz material penalization function used for density-based topology optimization
Optional	*minimum_ersatz_material_value*	specifies the minimum ersatz material value assigned to the ersatz material penalization function used for density-based topology optimization

2.4 Scenarios

A `scenario` describes the problem setup for the analysis. Each *scenario block* begins and ends with the tokens `begin scenario {string}` and `end scenario`, respectively. The string following the `begin scenario` expression denotes an identifier assigned to the *scenario block*. Other blocks in the input deck will use this identifier to refer to the `scenario block`. The following is a typical scenario block setup:

```
begin scenario 1
  service 2
  physics steady_state_mechanics
  dimensions 3
  loads 1 2
```

(continues on next page)

(continued from previous page)

```
  boundary_conditions 5
end scenario
```

The input deck can contain an arbitrary number of scenario blocks. Table 2.3.2 provides a brief description of the keywords supported within a scenario block. These keywords can be specified in any order within the *scenario block*.

2.4.1 service

The `service` keyword is used to specify the identifier assigned to the *service block* describing the computer hardware needs for the code running the simulation. The syntax for this keyword is:

```
service {string}
```

Table 2.4.1: Description of the supported elliptic physics.

Physics	Description
steady_state_mechanics	steady state linear mechanical physics
steady_state_thermal	steady state linear thermal conduction physics
steady_state_electrical	steady state linear electrical conduction physics
steady_state_electrothermal	steady state linear thermal-electrical conduction physics
steady_state_thermomechanics	steady state linear thermal-mechanical physics
steady_state_nonlinear_mechanics	steady state nonlinear mechanical physics
steady_state_nonlinear_thermomechanics	steady state nonlinear thermal-mechanical physics

2.4.2 physics

The `physics` keyword is used to specify the physics to be simulated by the *service* application. The syntax for this keyword is:

```
physics {string}
```

Table 2.4.1 lists the supported physics. A brief description is provided for each of the physics in Table 2.4.1.

steady_state_mechanics

The `steady_state_mechanics` physics is used to simulate steady state linear mechanical physics. The material constitutive behavior is assumed to follow Hooke's law.

steady_state_thermal

The `steady_state_thermal` physics is used to simulate steady state linear thermal physics. The material constitutive behavior is assumed to follow Fourier's Law of thermal conduction.

steady_state_electrical

The `steady_state_electrical` physics is used to simulate steady state linear electrical physics. The material constitutive behavior is assumed to follow Ohm's Law of electrical conduction.

steady_state_electrothermal

The `steady_state_electrothermal` physics is used to simulate steady state linear thermal-electrical physics. The material constitutive behavior is assumed to follow Fourier's Law and Ohm's Law of thermal and electrical conduction, respectively. Joule heating effects can be simulated with `steady_state_electrothermal` physics. A staggered coupling approach is applied to simulate the thermal-electrical system.

steady_state_thermomechanics

The `steady_state_thermomechanics` physics is used to simulate steady state linear thermal-mechanical physics. the material constitutive behavior is assumed to follow Fourier's Law and Hooke's Law of thermal conduction and mechanical deformations, respectively. A monolithic coupling approach is applied to simulate the thermal-mechanical system.

steady_state_nonlinear_mechanics

The `steady_state_nonlinear_mechanics` physics is used to simulate steady state nonlinear mechanical physics. The physical system is assumed to undergo large displacements and rotations but small strains. The *Materials Section* describes the hyperelastic material constitutive models supported with steady state nonlinear mechanical physics.

steady_state_nonlinear_thermomechanics

The `steady_state_nonlinear_thermomechanics` physics is used to simulate steady state nonlinear thermal-mechanical physics. The physical system is assumed to undergo large displacements and rotations but small strains. The *Materials Section* describes the hyperelastic material constitutive models supported with steady state nonlinear thermal-mechanical physics. The thermal material constitutive models are assumed to behave linearly and follow Fourier's Law of thermal conduction. A staggered coupling approach is employed to simulate the thermal-mechanical system.

2.4.3 dimensions

The `dimensions` keyword specifies the spatial dimensions of the problem. The syntax for this keyword is:

```
dimensions {integer}
```

Possible values are `2` and `3`.

2.4.4 boundary_conditions

The `boundary_conditions` keyword is used to specify the identifiers (ids) assigned to the *boundary condition blocks* assigned to the scenario. The syntax for this keyword is:

```
boundary_conditions {integer}{...}
```

2.4.5 loads

The `loads` keyword is used to specify the identifiers (ids) assigned to the *load blocks* assigned to the scenario. The systax for this keyword is:

```
loads {integer}{...}
```

2.4.6 newton_raphson

The `newton_raphson` keyword is used to specify the Newton-Raphson solver options assigned to the scenario. The systax for this keyword is:

```
newton_raphson {string}{...}
```

The `string` values are the identifiers (ids) assigned to the *newton_raphson blocks* defined in the input deck.

2.4.7 linear_solver

The `linear_solver` keyword is used to specify the linear solver options assigned to the scenario. The systax for this keyword is:

```
linear_solver {string}{...}
```

The `string` values are the identifiers (ids) of the *linear_solver blocks* defined in the input deck.

material_penalty_exponent

The `material_penalty_exponent` keyword is used to specify the penalty exponent assigned to the ersatz material penalization function used by the residual for density-based topology optimization. The syntax for this keyword is:

```
material_penalty_exponent {value}
```

The `default` value is set to `3.0`.

minimum_ersatz_material_value

The `minimum_ersatz_material_value` keyword is used to specify the minimum value the ersatz material penalization function used by the residual for density-based topology optimization can take. The syntax for this keyword is:

```
minimum_ersatz_material_value {value}
```

The `default` value is set to $1.0e^{-9}$.

2.4.8 Multi-Scenario Use Case

The evaluation of the objective function may require contributions from multiple scenarios, e.g., a mechanical scenario and a thermal scenario. If computational resources are limited and each scenario cannot be simulated using the multiple-program, multiple-data parallel workflow, the scenarios must be simulated sequentially to compute their respective contribution to the objective function.

The `weighted_scenarios` objective function evaluator was created to facilitate sequential evaluation of each scenario. Table 2.4.2 describes the keywords that must be defined within the scenario block if a *weighted_scenarios objective* is selected.

Table 2.4.2: Supported keywords within a scenario block for multi-scenario workflows.

Required/Optional	Keyword	Description
Required	*blocks*	use to specify the *block* block ids assigned to the scenario
Required	*output*	use to specify the *output* block ids assigned to the scenario

blocks

The `blocks` keyword is used to specify the *block blocks* identifiers (ids) assigned to the scenario using the format:

```
blocks {string}{...}
```

The *block blocks* is used to define the material properties and element type assigned to a region/domain in the geometry.

output

The `output` keyword is used to specify the *output blocks* identifiers (ids) assigned to the scenario using the format:

```
output {string}
```

The *output block* is used to define the output quantities of interests for the scenario under consideration.

2.5 Loads

Each *load block* begins and ends with the tokens `begin load {string}` and `end load`, respectively. The string following the `begin load` expression specifies an identifier (id) assigned to the *load block*. Other blocks in the input deck will use this id to refer to the `load block`. 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 input deck can contain an arbitrary number of *load blocks*.

2.5.1 Main Keywords

The keywords in Table 2.5.1 must be defined within a `load block`. These keywords can be specified in any order within the load block.

Table 2.5.1: Description of the supported keywords used to define a load block.

Required/Optional	Keyword	Description
Required	*type*	specifies the force type
Required	*location_type*	specifies the type of the `exodus` geometric entity where the force is applied
Required	*location_name*	specifies the name assigned to the `exodus` geometric entity
Required	*value*	specifies the values assigned to the force components
Optional	*material_id*	specifies the identifier of the material block associated with the `exodus` geometric entity

type

The `type` keyword is used to specify the force type. The syntax for this keyword is:

```
type {string}
```

Table 2.5.4 and Table 2.5.5 list the supported surface and volume forces, respectively.

location_type

The `location_type` keyword is used to specify the `exodus` geometric entity type where the force is applied. The syntax for this keyword is:

```
location_type {string}
```

The supported options are `sideset`, `nodeset`, and `elemset`. The `default` value for volume forces is `elemset`, which denotes an `exodus` element set.

location_name

The `location_name` keyword is used to specify the name assigned to the `exodus` geometric entity set, e.g., `side set`, `node set`, and `element set`, where the force is applied. The syntax for this keyword is:

```
location_name {string}
```

value

The `value` keyword is used to specify the values assigned to the force components. The syntax for this keyword is:

```
value {value}{...}
```

The *force type* determines how many force components the user needs to set. For example, if a *traction* force is requested, the x, y, and z components must be defined for three-dimensional problems. In this example, the *traction* force is defined as follows within the load block:

```
value 0.0 -3e3 0.0
```

material_id

The `material_id` keyword is used to specify the identifier assigned to the *material block* that provides access to the material properties needed to evaluate the force. The syntax for this keyword is:

```
material_id {string}
```

The `material_id` keyword **must** be defined when using a *thermal convection force* or a *thermal radiation force*.

2.5.2 Concentrated Forces

A concentrated force is a force that acts at a point in the body. Table 2.5.2 lists the supported concentrated forces.

Table 2.5.2: Description of the supported concentrated forces.

Type	Description
concentrated_load	mechanical force applied at a point, i.e., node, in the body.

concentrated_load

The `concentrated_load` type is used to model a mechanical force applied at a point in the body. The *location_type*, *location_name*, *value*, and *direction* keywords must be defined to define a `concentrated_load`. The following is a typical concentrated load setup:

```
begin load 1
  type concentrated_load
  location_type nodeset
  location_name ns_1
  direction y
  value -3e3
  weight 0.5
end load
```

Keywords for concentrated forces

Table 2.5.3 shows the keywords used to define a concentrated force.

Table 2.5.3: Description of the keywords used to define a concentrated force.

Required/Optional	Keyword	Description
Required	*direction*	specifies the active concentrated force component
Optional	*weight*	specifies a multiplier applied to the concentrated force
Optional	*index*	specifies a list of node indices

direction

The `direction` keyword defines the active spatial components. The syntax for this keyword is:

```
direction {string}{...}
```

The supported options are `x`, `y`, and `z`. A magnitude must be provided for each active spatial component using the *value* keyword. A negative value denotes a compressive force while a positive value denotes a tensile force.

weight

The `weight` keyword defines a scalar multiplier applied to all the force components. The syntax for this keyword is:

```
weight {value}
```

The `default` value is set to `1.0`.

index

The `index` keyword provides a mechanism to define a list of indices associated with the node identifiers (ids) that make up the `exodus` node set assigned to the concentrated force, i.e., the node set assigned to the *location_name* keyword within the load block.

The value of the index, or indices, cannot be greater than the length of the node set minus one. Mathematically, this is described as $N_{ids} - 1$, where N_{ids} is the total number of node ids in the node set. Therefore, the `index` set I is defined as $I = \{0, \ldots, N_{ids} - 1\}$. **Notice that the code uses zero-base numbering to define the index set**.

The input deck syntax for the `index` keyword:

```
index {value}{...}
```

This parameter is useful for `optimization workflows`. For example, instead of optimizing for the spatial coordinates of the node(s) where a concentrated load is applied, the `index` option can be used to define a set of indices

associated with entries in the `exodus` node set, e.g., node set with name `ns_1`. This approach will reduce the number of optimization parameters while attaining the goal of optimizing for the location where the concentrated load will be applied.

The reference `exodus` node set used by the optimizer is the node set assigned to the *location_name* keyword. The following is a typical concentrated load block setup for an optimization workflow:

```
begin load 1
  type concentrated_load
  location_type nodeset
  location_name ns_1
  direction y
  value -3e3
  weight 0.5
  index 10
end load
```

where `index 10` denotes the tenth node id entry in the `exodus` node set with name `ns_1`.

2.5.3 Surface Forces

A surface force is a force that acts on an internal or external surface of a body. Table 2.5.4 lists the supported surface forces.

Table 2.5.4: Description of the supported surface forces.

Type	Description
traction	force applied to the surface of a body.
pressure	force per unit area applied perpendicular to the surface of a body.
follower_pressure	force per unit area that tends to be perpendicular to the deflected application surface.
thermal_flux	rate at which heat is transfered per unit area.
thermal_radiation	a nonlinear process of heat transfer per unit area.
thermal_convection	transfer of heat from one location to another.
electrical_flux	measures the electric field through the surface of a body.

traction

A `traction` is a force applied to generate motion between a body and a tangential surface through the use of dry friction or shear force. The traction components are separated by spaces, e.g., `0.2 2e-3 200` in three dimensions. The *location_name* keyword must be defined to specify the application surface. The following is a typical `traction` block setup:

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

pressure

A `pressure` is a force applied perpendicular to the surface of a body per unit area. The *location_name* keyword must be defined to specify the application surface. The following is a typical `pressure` block setup:

```
begin load 1
  type pressure
  location_type sideset
  location_name top
  value 1e5
end load
```

follower_pressure

A `follower_pressure` is a force per unit area that tends to be perpendicular to the deflected application surface. The *location_name* keyword must be defined to specify the application surface. The following is a typical `follower pressure` block setup:

```
begin load 1
  type follower_pressure
  location_type sideset
  location_name top
  value 1e5
end load
```

thermal_flux

A `thermal_flux` force is the rate at which heat is transfered per unit area. The *location_name* keyword must be used to specify the application surface. The following is a typical `thermal flux` block setup:

```
begin load 1
  type thermal_flux
  location_type sideset
  location_name bottom
  value -1e3
end load
```

thermal_convection

A `thermal_convection` flux is the transfer of heat from one location to another due to natural/forced convection or conduction. The *location_name* keyword must be defined to specify the application surface. Furthermore, the *material_id* keyword must be defined to specify the identifier assigned to the *material block* that defines the reference temperature used by the linear heat transfer model.

The following is a typical `thermal_convection` block setup:

```
begin load 1
  type thermal_convection
  location_type sideset
  location_name sideset_one
  value 750.0
  material_id 1
end load
```

The `value` parameter **must** be set to the material heat transfer coefficient.

thermal_radiation

A `thermal_radiation` flux denotes the process by which thermal energy is emitted by a heated surface. The rate at which a body radiates thermal energy depends upon the nature of the surface. For instance, a blackbody absorbs all the radiant energy that falls on it. Such a perfect absorber is also a perfect emitter. In contrast, a surface made out of silver is a poor emitter and a poor absorber. The *location_name* keyword must be defined to specify the application surface. Furthermore, the *material_id* keyword must be defined to specify the material block that contains the *emissivity* parameter and the *Stefan–Boltzmann constant* used to evaluate the `thermal_radiation` flux.

The following is a typical `thermal_radiation` block setup:

```
begin load 1
  type thermal_radiation
  location_type sideset
  location_name ss_rad
  material_id 1
end load
```

electrical_flux

An `electrical_flux` is a measure of the electric field through the surface of a body. The *location_name* keyword must be specified to set the application surface. The following is a typical `electrical_flux` block setup:

```
begin load 1
  type electrical_flux
  location_type sideset
  location_name left
  value 2e3
end load
```

2.5.4 Volume Forces

A volume force is a force per unit volume that acts throughout the volume of a body. Table 2.5.5 lists the supported volume forces.

Table 2.5.5: Description of the supported volume forces.

Type	Description
body_load	mechanical force per unit volume, e.g., gravity load.
heat_source	thermal force per unit volume.
joule_heating	process where energy from an electric current is converted to heat.
dark_current_source	electric current due to the random generation of electrons and holes.
light_current_source	generation of electric current in an electrical device.

Mechanical Volume Forces

The following options are used to model mechanical volume forces.

body_load

A `body_load` is a force that acts throughout the volume of a body. Body forces are typically used in mechanical applications to define gravity forces. The *location_name* keyword is used to specify the application volume, i.e., `exodus` element block. If the application volume is not provided, the code assumes that the body force acts on the entire body. The following is a typical `body force` block setup:

```
begin load 1
  type body_load
  location_name block_1
  value 0.0 -13243.0 0.0
end load
```

Thermal Volume Forces

The following options are used to model thermal volume forces.

heat_source

A `heat_source` describes heat generation within a volume. Heating and cooling mechanisms are expressed with positive and negative values, respectively. The *location_name* keyword is used to set the application volume, i.e., `exodus` element block. The following is a typical `heat source` block setup:

```
begin load 1
  type heat_source
  location_name block_1
  value 100.0
end load
```

joule_heating

A `joule_heating` force describes the process where the energy of an electric current is converted into heat. The *location_name* keyword is used to specify the application volume, i.e., `exodus` element block. If the application volume is not provided, the code assumes that the joule heating acts throughout the entire body. The following is a typical `joule heating` block setup:

```
begin load 1
  type joule_heating
  location_name block_joule
end load
```

Electrical Volume Forces

The following options are used to model electrical volume forces.

dark_current_source

A `dark_current_source` describes the electric current due to the random generation of electrons and holes within the depletion region of an electrical device. The *location_name* keyword must be used to specify the application volume, i.e., `exodus` element block. The following is a typical `dark current source` block setup:

```
begin load 1
  type dark_current_source
  model quadratic
  location_name bottom_busbar top_busbar fingers
end load
```

The `location_name` keyword accepts a list of `exodus` element set names.

light_current_source

A `light_current_source` describes the generation of current in an electrical device, e.g., a solar cell. The *location_name* keyword must be used to specify the application volume, i.e., `exodus` element block. The following is a typical `light current source` block setup:

```
begin load 1
  type light_current_source
  model constant
  location_name matrix
end load
```

The `location_name` keyword accepts a list of `exodus` element set names.

Keywords for electrical volume forces

The following keyword must be defined for all the supported electrical volume forces.

model

The `model` keyword is used to specify the electrical source model using the syntax:

```
model {string}
```

The only supported option for a *light_current_source* is the `constant` source model and for a *dark_current_source* is the `quadratic` source model.

2.6 Boundary Conditions

The *boundary_condition block* is used to define essential boundary conditions. Each `boundary_condition block` begins and ends with the tokens `begin boundary_condition {string}` and `end boundary_condition`, respectively. The string following the `begin boundary_condition` expression specifies an identifier (id) assigned to the *boundary condition block*. Other blocks in the input deck will use the id to refer to the boundary condition block.

The following is a typical boundary condition block setup:

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

2.6.1 Main Keywords

Table 2.6.1 lists the keywords supported within a `boundary_condition` block.

Table 2.6.1: Description of the keywords supported within a boundary_condition block.

Required/Optional	Keyword	Description
Required	*type*	specifies the essential boundary condition type
Required	*value*	specifies the value assigned to the degree(s) of freedom where the essential boundary condition is enforced
Required	*location_name*	specifies the name assigned to the `exodus` geometric entity set where the essential boundary condition is enforced
Required	*degree_of_freedom*	specifies the degree(s) of freedom where the essential boundary condition is enforced
Optional	*location_type*	specifies the type of the `exodus` geometric entity set

type

The `type` keyword is used to specify the essential boundary condition type. The syntax for this keyword is:

```
type {string}
```

The supported options are `fixed_value` and `time_function`.

location_type

The `location_type` keyword is used to specify the type of the `exodus` geometric entity set. The syntax for this keyword is:

```
location_type {string}
```

The supported options are `elemset`, `sideset`, and `nodeset`. The `default` behavior is to read this information from the exodus mesh file.

location_name

The `location_name` keyword is used to specify the name assigned to the `exodus` geometric entity set where the essential boundary condition is applied. The syntax for this keyword is:

```
location_name {string}
```

Table 2.6.2: Description of supported degrees of freedom

Load Type	Description
temp	temperature
potential	electric potential
dispx	displacement in the x direction
dispy	displacement in the y direction
dispz	displacement in the z direction

degree_of_freedom

The `degree_of_freedom` keyword is used to specify the degree of freedom, or list of degrees of freedom, where the essential boundary condition is enforced. The syntax for this keyword is:

```
degree_of_freedom {string}{...}
```

Table 2.6.2 list the degrees of freedom where essential boundary conditions can be enforced.

Multiple degrees of freedom can be specified in a single `degree_of_freedom` line. For example, in a three-dimensional mechanical simulation, a single `degree_of_freedom` line can be used to constraint the three displacement degrees of freedom using the following format:

```
degree_of_freedom dispx dispy dispz
```

If multiple degrees of freedom are listed after the `degree_of_freedom` keyword, the software will expect the same number of values listed after the *value* keyword using the following format:

```
value 0.0 0.0 0.0
```

This example assigns a fixed value of `0.0` to the three displacement degrees of freedom.

value

The `value` keyword is used to specify the value(s) assigned to the degree(s) of freedom where the essential boundary conditions are enforced. The syntax for this keyword is:

```
value {value}{...}
```

The number of value inputs must match the number of degrees of freedom inputs specified in the *degree_of_freedom* keyword.

2.6.2 Examples

This section presents multiple examples describing how to set up a *boundary_condition block* in the input deck based on the *physics* selected in the *scenario block*.

Temperature

The following is a typical `boundary_condition block` setup for thermal physics:

```
begin boundary_condition 1
  type fixed_value
  location_type sideset
  location_name ss_1
  degree_of_freedom temp
  value 100.0
end boundary_condition
```

Electric Potential

The following is a typical `boundary_condition block` setup for electrical physics:

```
begin boundary_condition 1
  type fixed_value
  location_type sideset
  location_name sideset_1
  degree_of_freedom potential
  value 0.5
end boundary_condition
```

Displacement

The following is a typical `boundary_condition block` setup for mechanical physics:

```
begin boundary_condition 1
  type fixed_value
  location_type sideset
  location_name sideset_fixed
  degree_of_freedom dispx dispy dispz
  value 0.0 0.0 0.0
end boundary_condition
```

2.7 Materials

Each *material block* begins and ends with the tokens `begin material {string}` and `end material`, respectively. The string following `begin material` specifies an identifier for the material block. Other blocks in the input deck will use this material block identifier to refer to the *material block*. The following is a typical material block setup:

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

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

2.7.1 Material Models

The following keyword must be defined within a material block:

material_model

The *material_model* keyword is used to specify the material model used to predict the behavior of the material within an *element block*. At least one material model **must** be defined to run an analysis. The syntax for this keyword is:

```
material_model {string}
```

where the `{string}` placeholder is replaced with one of the material models presented in Table 2.7.1 and Table 2.7.2.

Table 2.7.1: Material constitutive models for single-physics problems.

Material Model	Description
isotropic_linear_elastic	An isotropic elastic material model assumes homogeneous mechanical properties within an *element block*. The material properties associated with this material model are the *Young's Modulus* and *Poisson's Ratio*.
orthotropic_linear_elastic	An orthotropic elastic material assumes homogeneous orthotropic mechanical properties within an *element block*. The material properties associated with this material model are the *Young's Modulus X*, *Young's Modulus Y*, *Young's Modulus Z*, *Poisson's Ratio XY*, *Poisson's Ratio XZ*, *Poisson's Ratio YZ*, *Shear Modulus XY*, *Shear Modulus XZ*, and *Shear Modulus YZ*.
isotropic_linear_thermal	An isotropic thermal material model assumes homogeneous thermal properties within an *element block*. The material properties associated with this material model are the *Thermal Conductivity*, *Thermal Expansivity*, *Reference Temperature*, *Stefan-Boltzmann Constant*, and *Emissivity Constant*.
isotropic_electrical	An isotropic electrical material model assumes homogeneous electrical properties within an *element block*. The material properties associated with this material model are the *Electrical Conductivity* and *Electrical Resistance*.
isotropic_hyperelastic_kirchhoff	A hyperelastic Saint Venant-Kirchhoff material model assumes homogeneous mechanical properties within an *element block*. The material properties associated with this material model are the *Shear Modulus*, *Bulk Modulus*, *C10*, and *D*.
isotropic_hyperelastic_neohookean	An hyperelastic Neo-Hookean material model assumes homogeneous mechanical properties within an *element block*. The material properties associated with this material model are the *Shear Modulus*, *Bulk Modulus*, *C10*, and *D*.

Table 2.7.2: Description of supported material models for multi-physics problems.

Material Model	Description
isotropic_electrothermal	An isotropic electrical-thermal material model assumes homogeneous thermal and electrical properties within an *element block*. The electrical and thermal material properties associated with this material models are defined in sections *Isotropic Electrical Properties* and *Isotropic Thermal Properties*, respectively.
isotropic_linear_thermoelastic	An isotropic thermal-elastic material model assumes homogeneous thermal and mechanical properties within an *element block*. The mechanical and thermal properties associated with this material model are defined in sections *Isotropic Elastic Properties* and the *Isotropic Thermal Properties*, respectively.
isotropic_thermal_hyperelastic_kirchhoff	A thermal-hyperelastic material model assumes homogeneous thermal and mechanical properties within an *element block*. The mechanical and thermal properties associated with this material model are defined in sections *Isotropic Hyperelastic Properties* and *Isotropic Thermal Properties*, respectively.
isotropic_thermal_hyperelastic_neohookean	A thermal-hyperelastic material model assumes homogeneous thermal and mechanical properties within an *element block*. The mechanical and thermal properties associated with this material model are defined in sections *Isotropic Hyperelastic Properties* and *Isotropic Thermal Properties*, respectively.

2.7.2 Isotropic Elastic Properties

Table 2.7.3 describes the keywords used to set the material constants for an isotropic elastic material model.

Table 2.7.3: Material constants for an isotropic elastic material model.

Required/Optional	Property	Description
Required	*youngs_modulus*	measures the tensile or compressive stiffness of a material
Required	*poissons_ratio*	measures the deformation of a material in the direction perpendicular to the load direction

youngs_modulus

The syntax used to specify this material property is:

```
youngs_modulus {value}
```

poissons_ratio

The syntax used to specify this material property is:

```
poissons_ratio {value}
```

2.7.3 Orthotropic Elastic Properties

Table 2.7.4 describes the keywords used to set the material constants for an orthotropic elastic material model.

Table 2.7.4: Material constants for an orthotropic elastic material model.

Required/Optional	Property	Description
Required	*youngs_modulus_x*	measures the tensile or compressive stiffness of a material in the X direction
Required	*youngs_modulus_y*	measures the tensile or compressive stiffness of a material in the Y direction
Required	*youngs_modulus_z*	measures the tensile or compressive stiffness of a material in the Z direction
Required	*poissons_ratio_xy*	measures the contraction in the Y direction when an extension is applied in the X direction
Required	*poissons_ratio_xz*	measures the contraction in the Z direction when an extension is applied in the X direction
Required	*poissons_ratio_yz*	measures the contraction in the Z direction when an extension is applied in the Y direction
Required	*shear_modulus_xy*	measures the shear modulus in the Y direction on the plane whose normal is in the X direction
Required	*shear_modulus_xz*	measures the shear modulus in the Z direction on the plane whose normal is in the X direction
Required	*shear_modulus_yz*	measures the shear modulus in the Z direction on the plane whose normal is in the Y direction

youngs_modulus_x

The syntax used to specify this material property is:

```
youngs_modulus_x {value}
```

youngs_modulus_y

The syntax used to specify this material property is:

```
youngs_modulus_y {value}
```

youngs_modulus_z

The syntax used to specify this material property is:

```
youngs_modulus_z {value}
```

poissons_ratio_xy

The syntax used to specify this material property is:

```
poissons_ratio_xy {value}
```

poissons_ratio_xz

The syntax used to specify this material property is:

```
poissons_ratio_xz {value}
```

poissons_ratio_yz

The syntax used to specify this material property is:

```
poissons_ratio_yz {value}
```

shear_modulus_xy

The syntax used to specify this material property is:

```
shear_modulus_xy {value}
```

shear_modulus_xz

The syntax used to specify this material property is:

```
shear_modulus_xz {value}
```

shear_modulus_yz

The syntax used to specify this material property is:

```
shear_modulus_yz {value}
```

2.7.4 Isotropic Thermal Properties

Table 2.7.5 describes the keywords used to set the material constants for an isotropic thermal material model.

Table 2.7.5: Material constants for an isotropic thermal material model.

Required/Optional	Property	Description
Required	*thermal_conductivity*	measures the ability of a material to conduct heat
Optional	*thermal_expansivity*	measures how the size of an object changes with a change in temperature
Optional	*reference_temperature*	measures the temperature at which the thermal deformation (and hence the stress) of a material is zero
Optional	*emissivity*	measures how effective a surface is in emitting energy as thermal radiation
Optional	*stefan_boltzmann_constant*	specifies the value of the Stefan–Boltzmann constant

thermal_conductivity

The syntax used to specify this material property is:

```
thermal_conductivity {value}
```

thermal_expansivity

The syntax used to specify this material property is:

```
thermal_expansivity {value}
```

The `default` value is set to zero.

reference_temperature

The syntax used to specify this material property is:

```
reference_temperature {value}
```

The `default` value is set to zero.

emissivity

The syntax used to specify this material property is:

```
emissivity {value}
```

Quantitatively, the `emissivity` of a material is the ratio between the thermal radiation from a surface and the radiation from an ideal black surface at the same temperature as given by the Stefan–Boltzmann law. The Stefan–Boltzmann law states that the total energy radiated per unit surface area per unit time is proportional to the fourth power of the black body temperature, T, i.e., σT^4, where σ denotes the *Stefan–Boltzmann constant*.

The value of the `emissivity` parameter varies from 0 to 1. A value of 1 denotes a material capable of emitting thermal radiation with maximum strength, e.g., a black body. A value of 0 denotes a material incapable of emitting thermal radiation to the environment. The `default` value for the `emissivity` parameter is set toi `1.0`.

stefan_boltzmann_constant

The `stefan_boltzmann_constant` parameter is used to specify the value of the Stefan–Boltzmann constant, which has a value of $5.67 \times 10^{-8}\ W \cdot m^{-2} \cdot K^{-4}$. The syntax for specifying this material property is:

```
stefan_boltzmann_constant {value}
```

the `default` value is set to 5.67×10^{-8}.

2.7.5 Isotropic Electrical Properties

Table 2.7.6 describes the keywords used to set the material constants for an isotropic electrical material model.

Table 2.7.6: Material constants for an isotropic electrical material model.

Required/Optional	Property	Description
Required	*electrical_conductivity*	measures the ability of a material to conduct electricity
Optional	*electrical_resistance*	measures the opposition of a material to the flow of electric current

electrical_conductivity

The syntax used to specify this material property is:

```
electrical_conductivity {value}
```

electrical_resistance

The syntax used to specify this material property is:

```
electrical_resistance {value}
```

2.7.6 Isotropic Hyperelastic Properties

The following material constants are associated with isotropic hyperelastic material models. A hyperelastic material model is defined by one of the following material parameter pairs:

1. E-ν,
2. μ-B,
3. C10-D1,

where μ is the shear modulus, B is the bulk modulus, E denotes *Young's modulus*, ν denotes *Poisson's ratio*, *C10* is a shear constant used for hyperelastic materials, and *D1* is an incompressiblity constant used for hyperelastic materials.

Table 2.7.7 describes the keywords used to set the material constants for an isotropic hyperelastic material model.

Table 2.7.7: Supported input parameters for isotropic hyperelastic material models.

Required/Optional	Property	Description
Optional	*youngs_modulus*	measures the tensile or compressive stiffness of a material
Optional	*poissons_ratio*	measures the deformation of a material in the direction perpendicular to the load direction
Optional	*shear_modulus*	measures the elastic shear stiffness of a material
Optional	*bulk_modulus*	measures the resistance of a substance to bulk compression
Optional	*C10*	measures the elastic shear stiffness of a material
Optional	*D1*	measures the resistance of a substance to bulk compression
Optional	*use_continuation*	enables continuation of the ersatz material parameters
Optional	*update_intervals*	sets the frequencies at which the material parameters can be updated when continuation is enabled
Optional	*update_thresholds*	sets thresholds for the material parameters that are updated through continuation
Optional	*update_increments*	sets update increments for the material parameters that are updated through continuation

The material parameter pair assigned to the hyperelastic material model **must** be set from the parameter pairs introduced in the *Isotropic Hyperelastic Properties* section.

shear_modulus

The `shear_modulus` measures the elastic shear stiffness of a material. It is defined as the ratio of shear stress to the shear strain. The syntax used to specify this material property is:

```
shear_modulus {value}
```

bulk_modulus

The `bulk_modulus` measures the resistance of a material to bulk compression. The syntax used to specify this material property is:

```
bulk_modulus {value}
```

C10

The `C10` parameter is defined as $\mathrm{C10} = \frac{\mu}{2}$, where μ is the shear modulus. The syntax used to specify this material property is:

```
c10 {value}
```

D1

The `D1` parameter is defined as $\mathrm{D1} = \frac{2}{B}$, where B is the bulk modulus. The syntax used to specify this material property is:

```
d1 {value}
```

use_continuation

The `use_continuation` keyword is used to update material parameters through a continuation scheme. For instance, in density-based topology optimization problems, continuation is used to modify the `penalty_exponent` assigned to the ersatz material model. The syntax used to specify this parameter is:

```
use_continuation {Boolean}
```

The `default` value is set to `false`. The continuation mechanism is only available for density-based topology optimization problems. Furthermore, only the `penalty_exponent` parameter can be modified through continuation.

Additional Comments

The length of the *update_intervals*, *update_thresholds*, and *update_increments* arrays **must** be the same. The following text snippet shows a valid continuation setup:

```
use_continuation true
update_intervals    2     5
update_thresholds  2.0   3.0
update_increments 0.025 0.050
```

The initial `penalty_exponent` for this example is set to `1.0`. There are two update intervals in this example. In the first update interval, the interval from `1.0` to `2.0`, the `penalty_exponent` will increase by `0.025` every `2` optimization iterations until it reaches the `2.0` threshold. In the second update interval, the interval from `2.0` to `3.0`, the `penalty_exponent` will increase by `0.05` every `5` optimization iterations until it reaches the `3.0` threshold, which is the maximum allowable value for the `penalty_exponent` parameter.

The following text snippet shows an invalid continuation problem setup:

```
use_continuation true
update_intervals    2     5
update_thresholds  2.0   3.0
update_increments 0.025
```

This continuation setup example is made invalid by the fact that two update intervals (and two update thresholds) are defined, but only one update increment is provided.

update_intervals

The `update_intervals` keyword is used to set the update intervals, i.e., rate at which the material parameter is updated through continuation. The syntax used to specify this keyword is:

```
update_intervals {integer}{...}
```

The `default` value is set to `2`, which means that the material parameter is updated every two optimization iterations. Multiple update intervals can be specified for a design study.

update_thresholds

The `update_thresholds` keyword is used to set one or more thresholds on the material parameter being updated through the continuation mechanism. The `update_thresholds` keyword can be used for the following purposes:

1. set a maximum value on the parameter of interest, and
2. set custom *update intervals* and *update increments* for the material parameter of interest.

The syntax used to specify this keyword is:

```
update_thresholds {value}{...}
```

The `default` value is set to `3.0`, which means that the maximum allowable value for the `penalty_exponent` parameter is set to `3.0`. Multiple update thresholds can be specified for a design study.

update_increments

The `update_increments` keyword is used to set the update increments for the material parameter, e.g., `penalty_exponent`, being updated through the continuation mechanism. The syntax used to specify this keyword is:

```
update_increments {value}{...}
```

The `default` value is set to `0.025`. Therefore, the `penalty_exponent` in a density-based topology optimization study will increase by a factor of `0.025` each time the continuation mechanism is called by the optimizer, which is set by the *update_intervals* parameter. Notice that the continuation mechanism applies an additive update scheme, i.e.,

```
new_parameter = old_parameter + 0.025
```

Multiple update increments can be specified for a design study.

2.7.7 Examples

Multiple `material block` examples are presented in this section to show how to set the material model and material constants within a `material block`.

Isotropic Elastic Material

```
begin material 1
  material_model isotropic_linear_elastic
  youngs_modulus 1e9
  poissons_ratio 0.33
end material
```

Orthotropic Elastic Model

```
begin material 1
  material_model orthotropic_linear_elastic
  poissons_ratio_xy 0.28
  poissons_ratio_xz 0.28
  poissons_ratio_yz 0.4
  shear_modulus_xy 6.6e9
  shear_modulus_xz 6.6e9
  shear_modulus_yz 3.9e9
  youngs_modulus_x 126e9
  youngs_modulus_y 11e9
  youngs_modulus_z 11e9
end material
```

Isotropic Thermal Model

```
begin material 1
  material_model isotropic_linear_thermal
  thermal_conductivity 210.0
end material
```

Isotropic Thermo-Mechanical Model

```
begin material 1
  material_model isotropic_linear_thermoelastic
  poissons_ratio 0.33
  youngs_modulus 68e9
  thermal_expansivity 2.0
  thermal_conductivity 16.0
  reference_temperature 100.0
end material
```

Isotropic Electrical Model

```
begin material 1
  material_model isotropic_linear_electrical
  electrical_conductivity 1e4
end material
```

Isotropic Electro-Thermal Model

```
begin material 1
  material_model isotropic_electrothermal
  thermal_conductivity 400.0
  reference_temperature 298.0
  electrical_resistance 0.212e-3
  electrical_conductivity 5.998e7
end material
```

The parameter *electrical_resistance* in the text snippet presented above is optionali. The *electrical_resistance* is only required if the `electric_field` is requested as an *output quantity of interest*.

Isotropic Hyperelastic Model

Option 1:

```
begin material 1
  material_model isotropic_hyperelastic_neohookean
  youngs_modulus 1.0e9
  poissons_ratio 0.30
end material
```

Option 2:

```
begin material 1
  material_model isotropic_hyperelastic_neohookean
  bulk_modulus  1.0e9
```

(continues on next page)

(continued from previous page)

```
  shear_modulus 1.5e6
end material
```

Option 3:

```
begin material 1
  material_model isotropic_hyperelastic_neohookean
  d1  0.1667
  c10 0.25
end material
```

where *D1* denotes an incompressibility constant for hyperelastic materials and *C10* is a material constant related to shear distortions.

Isotropic Thermal-Hyperelastic Model

Option 1:

```
begin material 1
  material_model isotropic_thermal_hyperelastic_neohookean
  youngs_modulus 1.0e9
  poissons_ratio 0.30
  thermal_expansivity 1.0
  thermal_conductivity 32.0
  reference_temperature 100.0
end material
```

Option 2:

```
begin material 1
  material_model isotropic_thermal_hyperelastic_neohookean
  bulk_modulus  1.0e9
  shear_modulus 1.5e6
  thermal_expansivity 1.0
  thermal_conductivity 32.0
  reference_temperature 100.0
end material
```

Option 3:

```
begin material 1
  material_model isotropic_thermal_hyperelastic_neohookean
  d1  0.1667
  c10 0.25
  thermal_expansivity 2.0
  thermal_conductivity 16.0
  reference_temperature 10.0
end material
```

where *D1* denotes an incompressibility constant and *C10* is a material constant of hyperelastic materials related to shear distortions.

2.8 Blocks

Each *block block* begins and ends with the tokens `begin block {string}` and `end block`, respectively. The string following the `begin block` expression specifies an identifier assigned to the *block block*. Other blocks in the input deck will use this identifier to refer to the `block block`. The following is a typical `block block` setup:

```
begin block 1
  material 1
  name design_domain
end block
```

The keywords in Table 2.8.1 can be specified in any order within the `block block`.

Table 2.8.1: Description of the keywords supported within a block block.

Required/Optional	Keyword	Description
Required	*material*	specifies the identifier assigned to the material block in the input deck
Required	*name*	specifies the name of the `exodus` element block assigned to the region
Optional	*initial_dvars_value*	specifies the value to be uniformly assigned to the design variables within the region

2.8.1 material

The `material` keyword is used to specify the identifier assigned to the `material block` assigned to this region, i.e. `block`. The syntax for this keyword is:

```
material {integer}
```

2.8.2 name

The `name` keyword is used to specify the name of the `exodus` element block associated with this region, i.e., `block`. The syntax for this keyword is:

```
name {string}
```

2.8.3 initial_dvars_value

The `initial_dvars_value` keyword is used to specify the value to be uniformly assigned to the design variables, e.g., density field in a topology optimization problem, within this region. The syntax for this keyword is:

```
initial_dvars_value {value}
```

The `default` value is set to `0.5`. The `initial_dvars_value` keyword is used when the *initialize_method* is set to `uniform_by_region`.

2.9 Criteria

Each *criterion block* begins and ends with the tokens `begin criterion {string}` and `end criterion`, respectively. The string following the `begin criterion` expression specifies an identifier assigned to the *criterion block*. Other blocks in the input deck will use the identifier to refer to the `criterion block`. The following is a typical `criterion block` setup:

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

The input deck can contain an arbitrary number of criterion blocks. The keywords to set the criterion parameters can be specified in any order within the `criterion block`.

Table 2.9.1: List of supported design criteria.

Type	Description
composite	a criterion defined by a combination of sub-criteria
inverse	a criterion for inverse problems
state_misfit	a measure of the misfit between the true and targeted state response
mass_properties	a measure of the mismatch between the true and targeted mass properties
mechanical_response	seeks to improve the mechanical performance at one or more surfaces
mechanical_compliance	a measure of the structural stiffness
thermal_compliance	a measure of the resistance to heat conduction
thermal_response	seeks to improve the thermal performance at one or more surfaces
thermomechanical_compliance	a measure of the structural stiffness and resistance to heat conduction in the structure
volume	a measure of the volume of the design
mass	a measure of the mass of the design
stress_pnorm	a measure of the p-norm of the stress tensor
thermal_flux_pnorm	a measure of the p-norm of the thermal flux
local_constraint	constraints a local quantity of interest(s) at each material point using the augmented Lagrangian method
elastic_energy_potential	a measure of the internal elastic energy of the structure in nonlinear mechanical and nonlinear thermo-mechanical problems
electrical_response	seeks to improve the electrical performance at one or more surfaces
power_surface_density	a measure of the output power produced by an electrical device

2.9.1 Main criterion block keywords

Table 2.9.2 lists the criterion keywords that can be used to set up any of the criteria in Table 2.9.1.

Table 2.9.2: Main criterion keywords.

Required/Optional	Keyword	Description
Required	*type*	specifies the criterion type
Optional	*location_name*	specifies the `exodus` geometric entities, e.g., `exodus` element blocks or side sets, where the criterion will be evaluated
Optional	*material_penalty_exponent*	specifies the penalty exponent assigned to the ersatz material penalization function used for density-based topology optimization
Optional	*minimum_ersatz_material_value*	specifies the minimum ersatz material value assigned to the ersatz material penalization function used for density-based topology optimization

type

The `type` keyword is used to specify the criterion type. Table 2.9.1 lists the supported criterion types. The syntax for this keyword is:

```
type {string}
```

Additional keywords may be required based on the type of criterion selected for evaluation. These keywords will be introduced in the *Examples* section.

location_name

The `location_name` keyword is used to specify the `exodus` geometric entities, e.g., `exodus` element block or side set, where the criterion will be evaluated. The syntax for this keyword is:

```
location_name {string}
```

The `default` behavior is to consider all the regions, i.e., `exodus element blocks`, in the evaluation of the criterion.

material_penalty_exponent

The `material_penalty_exponent` keyword is used to specify the penalty exponent assigned to the ersatz material penalization function used by the criterion for density-based topology optimization. The syntax for this keyword is:

```
material_penalty_exponent {value}
```

The `default` value is set to `3.0`.

minimum_ersatz_material_value

The `minimum_ersatz_material_value` keyword is used to specify the minimum value the ersatz material penalization function used by the criterion for density-based topology optimization can take. The syntax for this keyword is:

```
minimum_ersatz_material_value {value}
```

The `default` value is set to 1.0e^{-9}.

Table 2.9.3: List of the criteria that can be used with each scenario.

Criterion Type	Description
steady_state_mechanics	*composite*, *inverse*, *state_misfit*, *mass_properties*, *mechanical_response*, *mechanical_compliance*, *volume*, *mass*, *stress_pnorm*, *local_constraint*
steady_state_thermal	*composite*, *inverse*, *state_misfit*, *thermal_compliance*, *volume*, *thermal_flux_pnorm*, *local_constraint*, *thermal_response*
steady_state_electrical	*composite*, *inverse*, *state_misfit*, *volume*, *electrical_response*, *power_surface_density*
steady_state_thermomechanics	*composite*, *inverse*, *state_misfit*, *thermomechanical_compliance*, *volume*, *stress_pnorm*, *mechanical_response*
steady_state_electrothermal	*composite*, *inverse*, *state_misfit*, *thermal_compliance*, *volume*, *thermal_flux_pnorm*, *local_constraint*, *thermal_response*
steady_state_nonlinear_mechanics	*composite*, *inverse*, *state_misfit*, *local_constraint*, *volume*, *elastic_energy_potential*
steady_state_nonlinear_thermomechanics	*composite*, *inverse*, *state_misfit*, *local_constraint*, *volume*, *elastic_energy_potential*

2.9.2 Criteria alternatives

Table 2.9.3 lists the criteria alternatives based on the physics to be simulated. The *location_name* keyword can be used with all the criteria in Table 2.9.1 to specify the `exodus` geometric entity sets, e.g., `exodus` element blocks, where the criterion will be evaluated.

The next subsections will describe which quantity of interest the criteria seek to improve through optimization.

composite

The `composite` criterion is defined by a combination of multiple sub-criteria. This criterion type is used to combine multiple sub-criteria into a weighted sum. For the composite criterion 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 up the composite criterion.

inverse

The `inverse` criterion is used to define a criterion used to solve an inverse problem. The `inverse` criterion is defined by one or more sub-criteria. An `inverse` criterion includes the *ids* and *weights* of the sub-criteria that make up the `inverse` criterion. Furthermore, an `inverse` criterion must set the *target data* to be targeted as the data to be matched by the optimizer.

mass_properties

The `mass_properties` criterion is a measure of the mismatch between the true and the targeted mass properties. Minimizing this mismatch will force the optimized design to have the same mass properties as the targeted properties.

state_misfit

The `state_misfit` criterion is a measure of the mismatch between the true state response and the targeted state response. Minimizing this mismatch will force the optimized design to have the same state response as the targeted response. Mathematically, the state misfit criterion is defined as:

$$f(\mathbf{u}(\mathbf{z})) = \frac{1}{2}\sum_{i=1}^{N_u}\left(\frac{u_i^s(\mathbf{z})}{u_i^t} - 1.0\right)^2, \tag{2.9.1}$$

where $\mathbf{z}$ is the vector of design variables, u_i^t is the i-th component of the targeted state response, u_i^s is the i-th component of the true response, and N_u is the number of states, which is equal to the number of measurement points times the number of state degrees of freedom per measurement point, i.e., node.

mechanical_response

The `mechanical_response` criterion seeks to improve the mechanical response, i.e., displacements, by minimizing or maximizing the mechanical response at one or more surfaces, i.e., `exodus` side sets.

thermal_response

The `thermal_response` criterion seeks to improve the thermal response, i.e., temperature, by minimizing or maximizing the thermal response at one or more surfaces, i.e., `exodus` side sets.

electrical_response

The `electrical_response` criterion seeks to improve the electrical response, i.e., potential, by minimizing or maximizing the electrical response at one or more surfaces, i.e., `exodus` sidesets.

mechanical_compliance

The `mechanical_compliance` criterion is a measure of the structural stiffness. Minimizing this criterion will make the structure stiffer.

thermal_compliance

The `thermal_compliance` criterion is a measure of the resistance to heat conduction. Minimizing this criterion will reduce the temperature in a body.

thermomechanical_compliance

The `thermomechanical_compliance` criterion is a measure of the stiffness of a body and the resistance in a body to heat conduction. Minimizing this criterion will reduce the temperature in the body and make the body stiffer.

volume

The `volume` criterion is a measure of volume.

mass

The `mass` criterion is a measure of mass.

stress_pnorm

The `stress_pnorm` criterion computes the p-norm of the stress tensor, where p denotes the superscript used to compute the norm. This criterion seeks to reduce the average stress in a body.

thermal_flux_pnorm

The `thermal_flux_pnorm` criterion computes the p-norm of the thermal flux, where p denotes the superscript used to compute the norm. This criterion seeks to minimize the average thermal flux in a body.

local_constraint

The `local_constraint` criterion applies the augmented Lagrangian method to enforce one or more constraints at every material point. The following quantities can be enforced as local constraints: the `von Mises stress` and the `thermal flux`. Multiple quantities can be constrained at every material point. For example, in a thermo-mechanical problem, the `von Mises stress` and the `thermal flux` can be constrained simultaneously at every material point.

elastic_energy_potential

The `elastic_energy_potential` criterion is a measure of the internal elastic energy of the structure. An elastic material constitutive model is used to evaluate the stress.

power_surface_density

The `power_surface_density` is a measure of the output power produced by an electrical device. This criterion is not penalized with an ersatz material penalty. Thus, the `power_surface_density` criterion cannot be used with density-based topology optimization methods.

2.9.3 Examples

The following section includes examples showing the common `criterion block` setup for each of the criterion in Table 2.9.1. The *location_name* keyword can be used with all the criteria in Table 2.9.1 to specify the `exodus` geometric entity sets, e.g., `exodus` element blocks, where the criterion will be evaluated.

Design Mass

The following text snippet is a typical `criterion block` setup for the `mass` criterion:

```
begin criterion 1
  type mass
  location_name block_1 block_2
end criterion
```

The *location_name* keyword is used to specify the names of the `exodus` element blocks considered for the mass evaluation. If the *location_name* keyword is not defined, all the `exodus` element blocks will be considered in the mass evaluation.

Design Volume

The following text snippet is a typical `criterion block` setup for the `volume` criterion:

```
begin criterion 1
  type volume
  location_name block_1 block_2
end criterion
```

The *location_name* keyword is used to specify the names of the `exodus` element blocks considered for the volume evaluation. If the *location_name* keyword is not defined, all the `exodus` element blocks will be considered in the volume evaluation.

Mechanical Compliance

The following text snippet is a typical `criterion block` setup for the `mechanical_compliance` criterion:

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

Thermal Compliance

The following text snippet is a typical `criterion block` definition for the `thermal_compliance` criterion:

```
begin criterion 1
  type thermal_compliance
end criterion
```

Thermomechanical Compliance

The following text snippet is a typical `criterion block` setup for the `thermomechanical_compliance` criterion:

```
begin criterion 1
  type thermomechanical_compliance
  thermal_weighting_factor 1.0
  mechanical_weighting_factor 0.5
end criterion
```

The keywords in Table 2.9.4 are optional within the `thermomechanical_compliance` criterion block.

Table 2.9.4: Parameter options for a criterion of type `thermomechanical_compliance`.

Required/Optional	Keyword	Description
Optional	*thermal_weighting_factor*	weights the thermal compliance criterion
Optional	*mechanical_weighting_factor*	weights the mechanical compliance criterion

thermal_weighting_factor

The `thermal_weighting_factor` keyword is used to specify a scalar weigthing factor for the thermal compliance criterion. The syntax for this keyword is:

```
thermal_weighting_factor {value}
```

The `default` is set to `-1.0`.

mechanical_weighting_factor

The `mechanical_weighting_factor` keyword is used to specify a scalar weigthing factor for the mechanical compliance criterion. The syntax for this keyword is:

```
mechanical_weighting_factor {value}
```

The `default` is set to `1.0`.

Elastic Energy Potential

The following text snippet is a typical `criterion block` setup for the `elastic_energy_potential` criterion:

```
begin criterion 1
  type elastic_energy_potential
end criterion
```

Thermal Flux P-Norm

The following text snippet is a typical `criterion block` setup for the `thermal_flux_pnorm` criterion:

```
begin criterion 1
  type thermal_flux_pnorm
  pnorm_exponent 8
end criterion
```

The `pnorm_exponent` keyword is an optional parameter. This keyword is used to specify the exponent used in the p-norm calculation. Mathematically, The p-norm criterion is defined as:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}} \tag{2.9.2}$$

The `default` value for the `pnorm_exponent` is `6.0`.

Stress P-Norm

The following text snippet is a typical `criterion block` isetup for the `stress_pnorm` criterion:

```
begin criterion 1
  type stress_pnorm
  pnorm_exponent 8
end criterion
```

Mathematically, the p-norm criterion is given by Eq.2.9.2. The `default` value for the `pnorm_exponent` is `6.0`.

Power Surface Density

The following text snippet is a typical `criterion block` setup for the `power_surface_density` criterion:

```
begin criterion 1
  type power_surface_density
  load 2
end criterion
```

The `load` keyword is a **required** parameter. This keyword is used to specify the identifier (id) assigned to the *load block* containing the forcing function information, which is required to evaluate the `power_surface_density` criterion.

Local Constraint

The `local_constraint` criterion attempts to create a design that generates values of one or more quantities of interest; e.g., von Mises stress, below a targeted limit at every material point. The `local_constraint` criterion is modeled using the augmented Lagrangian method. The following text snippet is a typical `criterion block` setup for the `local_constraint` criterion:

```
begin criterion 1
  type local_constraint
  limits 5e6
  local_measure vonmises
  location_name block_1
end criterion
```

This example attempts to generate a design that meets the targeted von Mises stress limit at every material point.

The `local_constraint` criterion is typically used in combination with other criteria, e.g. *mass*. Table 2.9.5 list the input keywords that can be used to define a `local_constraint` criterion.

Table 2.9.5: Description of the keywords used to define a `local_constraint` criterion.

Required/Optional	Keyword	Description
Required	*limits*	imposes a limit on a quantity of interest at every material point
Required	*local_measure*	quantity of interests constrained at every material point
Optional	*max_penalty*	maximum value allowed for the penalty in the augmented Lagrangian formulation
Optional	*initial_penalty*	initial value for the penalty in the augmented Lagrangina formulation
Optional	*penalty_increment*	increment multiplier for the penalty in the augmented Lagrangina formulation

Limits

The `limits` keyword is used to specifiy a limit on a quantity of interest. The syntax for this keyword is:

```
limits {value}
```

Application Location

The `location_name` keyword is used to specify the regions, i.e., element blocks, where the *local measure* will be evaluated. Furthermore, the *location_name* keyword enables the enforcement of location-specific limits on the local measure. The syntax for the `location_name` keyword is:

```
location_name {string}
```

The `default` value is set to `all`, which assigns the same local measure limit to all the regions.

Local Measures

The `local_measure` keyword is used to specify the quantity of interest that will be constrained at every material point. The syntax for this keyword is:

```
local_measure {string}
```

Table 2.9.6 lists the quantities of interests that can be used as a local measure when using the *local_constraint* criterion.

Table 2.9.6: Local measures that can be used with the local constraint criterion.

Measure	Description
vonmises	measures weather an isotropic and ductile material will yield when subjected to a force, see Eq.2.9.3.
thermal_flux	measures the flow of thermal energy per unit area per unit time, see Eq.2.9.4.

von Mises

The von Mises local measure is defined as:

$$\sqrt{\frac{3}{2}\sigma_{ij}^{d}\sigma_{ij}^{d}}, \tag{2.9.3}$$

where σ_{ij}^{d} is the deviatoric stress tensor and $i, j = 1, \ldots, d$ with $d \in \{1, 2, 3\}$ denoting the spatial dimension.

Thermal Flux

The thermal flux measure is defined as:

$$\sqrt{\phi_i \phi_i}, \tag{2.9.4}$$

where ϕ_i is the thermal flux and $i = 1, \ldots, d$ with $d \in \{1, 2, 3\}$ denoting the spatial dimension.

Augmented Lagrangian Method

The following keywords are used to set the parameters for the augmented Lagrangian algorithm.

max_penalty

The `max_penalty` keyword is used to specify a maximum value on the penalty used in the augmented Lagrangian formulation. The syntax for this keyword is:

```
max_penalty {value}
```

The `default` value is set to `10000`.

initial_penalty

The `initial_penalty` keyword is used to specify an initial value for the penalty used in the augmented Lagrangian formulation. The syntax for this keyword is:

```
initial_penalty {value}
```

The `default` value is set to `10`.

penalty_increment

The `penalty_increment` keyword is used to specify a scalar multiplier to increase the penalty parameter. The syntax for this keyword is:

```
penalty_increment {value}
```

The `default` value is set to `1.1`.

Mass Properties

The `mass_properties` criterion attempts to generate a design that has desired mass properties. The supported mass property types are listed in Table 2.9.7. The targeted mass property `value` and a `weight` **must** be defined. The following text snippet is a typical `criterion block` setup specifying a subset of the mass properties to be matched by the optimizer:

```
begin criterion 1
  type mass_properties
  cgx  0.50  weight 1.0
  cgy  0.75  weight 1.5
end criterion
```

Note: If only a subset of the mass properties are requested, the target mass properties must be generated in the same coordinate system as the one used for the optimization problem.

The following text snippet is a typical `criterion block` setup directing the optimizer to match all the mass properties:

```
begin criterion 1
  type mass_properties
  mass 0.0664  weight 1.0
  ixx  3.7079  weight 1.0
  iyy  2.7113  weight 1.0
  izz  4.2975  weight 1.0
  cgx  4.1376  weight 1.0
  cgy  5.6817  weight 1.0
  cgz  2.9269  weight 1.0
end criterion
```

The `mass_properties` criterion can be modeled as an objective or a constraint function. If modeled as an objective, the `mass_properties` criterion is formulated as a multi-objective problem of the form:

$$F(x) = \sum_{i=1}^{N_f} \alpha_i f_i(x), \tag{2.9.5}$$

where $F(x)$ is the objective function, $f_i(x)$ is the i-th mass property criterion, $\alpha_i > 0$ is a scalar weight, and N_f is the number of criteria. The `weight` keyword is used to strengthen or weaken the enforcement of the mass property matching. If modeled as a constraint, the Method of Moving Asymptotes (MMA) optimizer is used to solve the multi-constraint optimization formulation. Consult the *Study* section to review the keywords for the MMA optimizer.

The mass properties in Table 2.9.7 are the mass properties that can be matched using the `mass_properties` criterion.

Table 2.9.7: Description of the mass properties that can be matched when using the `mass_properties` criterion.

Mass Property	Description
mass	mass of the object
cgx	x-component of the center of gravity
cgy	y-component of the center of gravity
cgz	z-component of the center of gravity
ixx	mass moment of inertia about the x axis
iyy	mass moment of inertia about the y axis
izz	mass moment of inertia about the z axis
ixy	xy product of inertia
iyz	yz product of inertia
ixz	xz product of inertia

Mass property

The `mass` property specifies the mass of the object. The syntax is:

```
mass {value} weight {value}
```

where the `mass {value}` keyword is the target mass and the `weight {value}` keyword is a scalar weight for the mass criterion, i.e., α_i in Eq.2.9.5.

X-component of the center of gravity

The `cgx` property specifies the x-component of the center of gravity. The syntax is:

```
cgx {value} weight {value}
```

where the `cgx {value}` keyword is the target x-component of the center of gravity and the `weight {value}` keyword is a scalar weight for the x-component center of gravity criterion, i.e., α_i in Eq.2.9.5.

Y-component of the center of gravity

The `cgy` property specifies the y-component of the center of gravity. The syntax is:

```
cgy {value} weight {value}
```

where the `cgy {value}` keyword is the target y-component of the center of gravity and the `weight {value}` keyword is a scalar weight for the y-component center of gravity criterion, i.e., α_i in Eq.2.9.5.

Z-component of the center of gravity

The `cgz` property specifies the z-component of the center of gravity. The syntax is:

```
cgz {value} weight {value}
```

where the `cgz {value}` keyword is the target z-component of the center of gravity and the `weight {value}` keyword is a scalar weight for the z-component center of gravity criterion, i.e., α_i in Eq.2.9.5.

Mass moment of inertia about the x-axis

The `ixx` property specifies the mass moment of inertia about the x axis. The syntax is:

```
ixx {value} weight {value}
```

where the `ixx {value}` keyword is the target moment of inertia and the `weight {value}` keyword is a scalar weight for the moment of inertia criterion, i.e., α_i in Eq.2.9.5.

Mass moment of inertia about the y-axis

The `iyy` property specifies the mass moment of inertia about the y axis. The syntax is:

```
iyy {value} weight {value}
```

where the `iyy {value}` keyword is the target moment of inertia and `weight {value}` keyword is a scalar weight for the moment of inertia criterion, i.e., α_i in Eq.2.9.5.

Mass moment of inertia about the z-axis

The `izz` property specifies the mass moment of inertia about the z axis. The syntax is:

```
izz {value} weight {value}
```

where the `izz {value}` keyword is the target moment of inertia and `weight {value}` keyword is a scalar weight for the moment of inertia criterion, i.e., α_i in Eq.2.9.5.

XY product of inertia

The `ixy` property specifies the xy product of inertia. The syntax is:

```
ixy {value} weight {value}
```

where the `ixy {value}` keyword is the target moment of inertia and the `weight {value}` is a scalar weight for the moment of inertia criterion, i.e., α_i in Eq.2.9.5.

YZ product of inertia

The `iyz` property specifies the yz product of inertia. The syntax is:

```
iyz {value} weight {value}
```

where the `iyz {value}` keyword is the target moment of inertia and `weight {value}` keyword is a scalar weight for the moment of inertia criterion, i.e., α_i in Eq.2.9.5.

XZ product of inertia

The `ixz` property specifies the xz product of inertia. The syntax is:

```
ixz {value} weight {value}
```

where the `ixz {value}` keyword is the target moment of inertia and the `weight {value}` is a scalar weight for the moment of inertia criterion, i.e., α_i in Eq.2.9.5.

State Misfit

The `state_misfit` criteria seeks to match a targeted state response. For instance, in a shape matching design problem, the optimizer will seek to find a design that matches a targeted deformation profile; i.e., displacements. The following text snippet is a typical `state_misfit` criterion block setup:

```
begin criterion 1
  type state_misfit
end criterion
```

Table 2.9.8 lists other keywords used to define a `state_misfit` criterion.

Table 2.9.8: Description of the keywords used to define a `state_misfit` criterion block.

Required/Optional	Keyword	Description
Optional	*transform*	specifies a data transform applied to the state misfit criterion

transform

The `transform` keyword is used to specify a data transform applied to the *state misfit* criterion. The syntax for this keyword is:

```
transform {string}
```

where the `{string}` keyword is replaced with the name of the data transform. Table 2.9.9 shows the supported data transforms.

Table 2.9.9: Supported data tansforms for the state misfit criterion.

Keyword	Description
exponential	specifies an exponential data transform

Exponential

The exponential data transform is defined as:

$$\phi(f(\mathbf{u}(\mathbf{z}))) = \exp^{(f(\mathbf{u}(\mathbf{z})))}, \tag{2.9.6}$$

where $f(\mathbf{u}(\mathbf{z}))$ is the state misfit criterion, $\mathbf{u}$ are the state variables, and $\mathbf{z}$ are the control variables, i.e., optimization variables.

Surface State Response

The surface state response criterion seeks to improve the performance of a design by minimizing or maximizing the state response at one or more surfaces. Table 2.9.10 describes the type of surface state responses that can be targeted by the optimizer for improvement.

Table 2.9.10: Surface state response criteria types.

Criterion	Description
thermal_response	seeks to optimize the thermal response at one or more surfaces
mechanical_response	seeks to optimize the mechanical response at one or more surfaces
electrical_response	seeks to improve the electrical response at one or more surfaces

Thermal response

The following text snippet is a typical `thermal_response` criterion block setup:

```
begin criterion 1
  type thermal_response
  location_name ss_temp
  degree_of_freedom temp
end criterion
```

Mechanical response

The following text snippet is a typical `mechanical_response` criterion block setup:

```
begin criterion 1
  type mechanical_response
  location_name ss_dispx
  degree_of_freedom dispx
end criterion
```

Electrical response

The following text snippet is a typical `electrical_response` criterion block setup:

```
begin criterion 1
  type electrical_response
  location_name ss_potential
  degree_of_freedom potential
end criterion
```

Keywords

Table 2.9.11 lists the keywords used to define a surface state criterion block.

Table 2.9.11: Description of the keywords used to define a state response criterion block.

Required/Optional	Keyword	Description
Required	*location_name*	specifies the names of the `exodus` surface sets, i.e., `exodus` side sets, where the state response will be evaluated
Required	*degree_of_freedom*	specifies the state degrees of freedom that will be targeted by the optimizer for improvement

Measurement locations

The `location_name` keyword is used to specify the names of the `exodus` surface sets, i.e., side sets, where the state response will be evaluated. The syntax for this keyword is:

```
location_name {string}{...}
```

Target degrees of freedom

The `degree_of_freedom` keyword is used to specify the state degrees of freedom that will be targeted by the optimizer for improvement. The syntax for this keyword is:

```
degree_of_freedom {string}{...}
```

Table 2.9.12 lists the degrees of freedom that can be targeted for optimization.

Table 2.9.12: Supported degree of freedom options

Criterion	Description
thermal_response	temp
mechanical_response	dispx, dispy, dispz
electrical_response	potential

Composite Criterion

The `composite` criterion is defined as a combination of multiple sub-criteria. The `composite` criterion enables the multiphysics solver to evaluate multiple criteria and combine them as a weighted sum before returning the value to the optimizer. All the sub-criteria must use the same *scenario*. The `composite` criterion **must** include the sub-criteria identifiers (ids) and weights that define it. The following text snippet is a typical `criterion block` setup for a `composite` criterion:

```
begin criterion 1
  type composite
  criterion_ids 2 3
  criterion_weights 1.0 1.0
end criterion

begin criterion 2
  type neohookean_energy_potential
end criterion

begin criterion 3
  type elastic_energy_potential
end criterion
```

Table 2.9.13 lists additional keywords used to define a `composite` criterion block.

Table 2.9.13: Additional keywords used to define a composite criterion block.

Required/Optional	Keyword	Description
Required	*criterion_ids*	specifies the criterion ids that make up the composite criterion
Optional	*criterion_weights*	specifies the weight of each criterion that make up the composite criterion

criterion_ids

The `criterion_ids` keyword is used to specify the criterion block identifiers (ids) that make up the composite criterion. The syntax for this keyword is:

```
criterion_ids {integer}{...}
```

The integer values are the ids of the criterion blocks that make up the `composite` criterion.

criterion_weights

The `criterion_weights` keyword is used to specify a scalar weight for each of the criteria that make up the composite criterion. The syntax for this keyword is:

```
criterion_weights {value}{...}
```

The `default` weight for each criterion is `1.0`.

Inverse Criterion

The `inverse` criterion **must** include the criterion block identifiers (ids) and weights that defined the `inverse` criterion. In addition, the `inverse` criterion **must** include information about the targeted data; i.e., the data to be matched by the optimizer. The following text snippet is a typical `criterion block` setup for the `inverse` criterion:

```
begin criterion 1
  type inverse
  criterion_ids 2
  criterion_weights 1.0
  target_data target_data_1.csv
  degree_of_freedom dispx dispy dispz
end criterion

begin criterion 2
  type state_misfit
end criterion
```

Table 2.9.14 lists the keywords that may need to be defined to use the `inverse` criterion.

Table 2.9.14: Keywords to define an inverse criterion.

Required/Optional	Keyword	Description
Required	*criterion_ids*	specifies the criterion block ids that make up the composite criterion
Required	*criterion_weights*	specifies a weight for each criterion that make up the composite criterion
Required	*target_data*	specifies the name of the csv file containing the data to be matched
Required	*degree_of_freedom*	specifies the identifiers of the degrees of freedom to be matched

Note: Only one criterion block id can be assigned to the *criterion_ids* keyword. Meaning, only one sub-criterion can be associated with an `inverse` criterion at the moment.

target_data

The `target_data` keyword specifies the name of the csv file containing the data to be matched by the optimizer. The syntax for this keyword is:

```
target_data {string}
```

The expected csv file format is:

```
EXODUS_GLOBAL_NODE_ID_1, TARGET_VALUE_FOR_QOI_1, ... , TARGET_VALUE_FOR_QOI_N
           :                        :                              :
           :                        :                              :
EXODUS_GLOBAL_NODE_ID_M, TARGET_VALUE_FOR_QOI_1, ... , TARGET_VALUE_FOR_QOI_N
```

where `EXODUS_GLOBAL_NODE_ID_1` is the exodus global node identifier (measurement point) and `TARGET_VALUE_FOR_QOI_#` is the target value for the quantity of interests at the measurement point. Table 2.6.2 shows the supported keywords used for the supported quantities of interest.

Matlab File

The `exodus global node ids` can be retrieved using the `exo2mat` application, which is part of the Morphorm engineering environment. The `exo2mat` application writes a Matlab file containing the finite element mesh data. The command to run the `exo2mat` application is:

```
exo2mat mesh.exo
```

where `mesh.exo` must be replaced with the name assigned by the user to the exodus mesh file. The output from this operation is a Matlab file named `mesh.mat`, where the `mesh` prefix will be replaced with the prefix assigned by the user to the exodus file. The Matlab file is written in the directory from where the `exo2mat` application is launched. The user can upload the `mesh.mat` file in Matlab or Python to retrieve the `exodus global node ids` for the location of interests, e.g., nodes.

The following python code snippet shows a simple python script that can be used to retrieve the `exodus global node ids` for node set *id=5*:

```
>>> from mat4py import loadmat
>>> data = loadmat('mesh.mat')
>>> data.get('nsnod05')
  [[15847], [16703], [17993], [17995], [17997], [17999], [18001], [18003], [17151],␣
↪[16723], [14997], [14999], [15001], [15003],
  [15421], [16275], [15419], [17131], [17133], [17561], [17563], [17991], [18005],␣
↪[17577], [17579], [16295], [16293], [15865],
  [15863], [15435], [15433], [15005], [14995], [15423], [16273], [16705], [16707],␣
↪[16709], [16711], [16713], [16715], [16717],
  [16719], [16721], [17135], [17137], [17139], [17141], [17143], [17145], [17147],␣
↪[17149], [17565], [17567], [17569], [17571],
  [17573], [17575], [15425], [15427], [15429], [15431], [15849], [15851], [15853],␣
↪[15855], [15857], [15859], [15861], [16277],
  [16279], [16281], [16283], [16285], [16287], [16289], [16291]]
```

The next python code snippet shows how to print the names of the accessor functions to the terminal. These functions are used to access mesh-based data from the python dictionary:

```
>>> data.
data.clear(       data.fromkeys(    data.items(       data.pop(         data.setdefault(␣
↪ data.values(
data.copy(        data.get(         data.keys(        data.popitem(     data.update(
```

For instance, the `data.keys()` command will print the keys of the python dictionary, .e.g.,

```
>>> data.keys()
dict_keys(['naxes', 'nnodes', 'nelems', 'nblks', 'nnsets', 'nssets', 'nsteps',
'ngvars', 'nnvars', 'nevars', 'nnsvars', 'nssvars', 'Title', 'x0', 'y0', 'z0',
'ssids', 'ssnum01', 'ssnod01', 'ssfac01', 'ssside01', 'sselem01', 'ssnum02',
'ssnod02', 'ssfac02', 'ssside02', 'sselem02', 'ssusernames', 'nsssides',
'nssdfac', 'nsids', 'nsnod01', 'nsfac01',    'nsnod02', 'nsfac02', 'nsnod03',
'nsfac03', 'nsnod04', 'nsfac04', 'nsnod05', 'nsfac05', 'nsnod06', 'nsfac06', '
nsusernames', 'nnsnodes', 'nnsdfac', 'blkids', 'blk01', 'blk01_nattr', 'blkusernames',
'blknames', 'node_num_map', 'elem_num_map'])
```

The `data.get('nsusernames')` command prints the names assigned by the user to the node sets:

```
>>> data.get('nsusernames')
'lower_surf\nsymm_plane\nfront_surf\nactuator1\nactuator2\nle\n'
```

where the relationship between the node set names and the node set identifiers (ids) is:

```
'lower_surf' = 'nsnod01'
'symm_plane' = 'nsnod02'
'front_surf' = 'nsnod03'
'actuator1'  = 'nsnod04'
'actuator2'  = 'nsnod05'
'lower_edge' = 'nsnod06'
```

Similarly, the `data.get('ssusernames')` command prints the names assigned by the user to the side sets:

```
>>> data.get('ssusernames')
'lower_surf_ss\n'
```

where the relationship between the side set names and the side set ids is:

```
'lower_surf' = 'ssnod01'
```

degree_of_freedom

The `degree_of_freedom` keyword is used to specify the degrees of freedom (DOF) identifiers to be matched by the optimizer, see Table 2.6.2. The syntax for this keyword is:

```
degree_of_freedom {string}{...}
```

The ordering of the degree of freedom identifiers **must** match the ordering in the csv file, i.e., column index associated with the `target_data` for this degree of freedom. For example, if the csv file is defined as:

```
1,0.1,0.2,0.3
2,0.4,0.5,0.6
```

where the values in the second, third, and fourth columns are the displacements in the y, z, and x directions. Then, the degree of freedom identifiers enter in the `degree_of_freedom` keyword **must** match the following ordering:

```
degree_of_freedom dispy dispz dispx
```

Table 2.9.15: Physical quantities of interest that can be matched when using the `state_misfit` criterion.

Load Type	Description
temp	temperature
potential	electric potential
dispx	displacement in the x direction
dispy	displacement in the y direction
dispz	displacement in the z direction

2.10 Constraint

The *constraint block* begins and ends with the tokens `begin constraint {string}` and `end constraint`, respectively. The string following the `begin constraint` expression denotes an identifier assigned to the *constraint block*. Other blocks in the input deck will use this identifier to refer to the `constraint block`. The following text snippet is a typical constraint block setup:

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

The user can specify multiple constraint blocks in the input deck. The *Method of Moving Asymptotes* will be selected as the `default` *optimizer* if the number of constraints is greater than one.

2.10.1 Constraint block keywords

The keywords in Table 2.10.1 can be specified in any order within the `constraint block`.

Table 2.10.1: Descriptions of the keywords used to define a constraint block.

Required/Optional	Keyword	Description
Required	*service*	specifies the service evaluating the constraint
Required	*scenario*	specifies the scenario evaluating the constraint
Required	*criterion*	specifies the criterion evaluated by the constraint function
Required	*relative_target* or *absolute_target*	specifies the constraint limit

service

The `service` keyword is used to specify the integer assigned to the *service block* responsible for evaluating the constraint. The syntax for this keyword is:

```
service {string}
```

criterion

The `criterion` keyword is used to specify the identifier assigned to the *criterion block* responsible for evaluating the criterion evaluating the constraint function. The syntax for this keyword is:

```
criterion {integer}
```

scenario

The `scenario` keyword is used to specify the identifier assigned to the *scenario block* evaluating the constraint. The syntax for this keyword is:

```
scenario {integer}
```

A scenario defines the physics, boundary conditions, and forces under which the constraint is evaluated.

relative_target

The `relative_target` keyword is used to specify a relative constraint limit. The syntax for this keyword is:

```
relative_target {value}
```

At the moment, the only constraints that use the `relative_target` keyword are the *mass* and *volume* criteria. In these use cases, a relative target with respect to the starting volume, or mass, of the design domain must be provided. For example, if the `relative_target` keyword is set to `0.5`, the optimized design will target 50% of the original design volume, or mass, as budget.

absolute_target

The `absolute_target` keyword is used to specify an absolute constraint limit. The syntax for this keyword is:

```
absolute_target {value}
```

For example, if an absolute target volume is provided, the optimizer will try to produce a design that meets the absolute target volume.

2.11 Objective

Each input deck contains only one *objective block*. However, the `objective block` can be composed of one or more sub-objectives. The objective block begins and ends with the tokens `begin objective` and `end objective`, respectively. The following is a typical `objective block` setup:

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

The `objective block` example presented above is defined by two sub-objectives. The first sub-objective uses `service 2` to simulate `scenario 1` and evaluate `criterion 3`, which is weighted with a value of `1.0`. The second sub-objective uses `service 3` to simulate `scenario 2` and evaluate `criterion 4`, which 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 block identifiers (ids). In the example above, to compute the final objective value, each sub-objective is evaluated, scaled by a weight, and then summed.

2.11.1 Objective block keywords

The keywords in Table 2.11.1 can be specified in any order within the `objective block`.

Table 2.11.1: Descriptions of the keywords used to define an objective block.

Required/Optional	Keyword	Description
Required	*type*	specifies the objective function evaluator
Required	*services*	specifies the services evaluating the sub-objectives
Required	*scenarios*	specifies the scenarios simulated to evaluate the sub-objectives
Required	*criteria*	specifies the criteria used to evaluate the sub-objectives

continues on next page

Table 2.11.1 – continued from previous page

Required/Optional	Keyword	Description
Optional	*weights*	specifies the weights assigned to the sub-objectives
Optional	*shape_services*	specifies the shape services computing the shape sensitivities

type

The `type` keyword is used to specify the objective function evaluator. The syntax for this keyword is:

```
type {string}
```

The supported objective function evaluators are shown in Table 2.11.2.

Table 2.11.2: Description of the supported objective evaluators

Type	Description
single_criterion	specifies a single objective evaluator, e.g., $f(\mathbf{x})$
weighted_sum	specifies a multi-objective evaluator, e.g, $\beta_j \sum_{j=1}^{N_f} f_j(\mathbf{x})$
weighted_scenarios	specifies a multi-scenario, multi-objective evaluator, e.g, $\alpha_i \sum_{i=1}^{N_s} \left(\beta_j \sum_{j=1}^{N_f} f_j(\mathbf{x}) \right)$

In Table 2.11.2, $f(x)$ is an objective function, $\mathbf{x}$ is the vector of design variables, β_j is a scalar weight multiplying the j-th objective function, N_f is the number of objective functions, $alpha_i$ is a scalar weight multiplying the contribution from the i-th scenario to the objective function, and N_s is the number of scenarios under consideration.

single_criterion

The `single_criterion` evaluator is used when only one criterion contributes to the objective function. A *weighted_sum* evaluator defined by a single criterion is equivalent to the `single_criterion` evaluator.

weighted_sum

The `weighted_sum` evaluator is used when multiple weighted criteria contribute to the final objective function value.

weighted_scenarios

A *scenario* defines the physics, boundary conditions, and forces under which each sub-objective is evaluated. For example, if the design is required to perform across multiple environments, e.g., mechanical and thermal environments, the design study must considered how the optimized design will perform inin both environments. If the computational resources are limited and multiple `analyze` instances cannot be afforded to simulate the mechanical and thermal environments concurrently, the `weighted_scenarios` evaluator allows users to sequenatially evaluate each environment using the same `analyze` instance. Meaning, one `analyze` instance will be used to evaluate each scenario.

The following is a typical `objective block` setup for the `weighted_scenarios` evaluator:

```
begin objective
   type weighted_scenarios
   services 2 2
   criteria 1 2
   scenarios 1 2
   weights 1.0 0.75
end objective
```

Notice that the scenario assigned to to each sub-objective uses the same `analyze` service. Thus, `service 2` will simulate `scenario 1` and `scenario 2` sequentially. When the optimizer requests the evaluation of the objective function, the `analyze` service, i.e., `service 2` will evaluate each scenario and their respective criteria, i.e. sub-objectives, sequentially, compute a weighted sum of the sub-objectives using the weights specified in the `objective block`, and return the weighted objective value to the optimizer.

services

The `services` keyword is used to specify the service block identifiers evaluating the sub-objectives. The syntax for this keyword is:

```
services {integer}{...}
```

There **must** be one `service` block id assigned to each sub-objective.

criteria

The `criteria` keyword is used to specify the criteria block identifiers used to define the sub-objectives. The syntax for this keyword is:

```
criteria {integer}{...}
```

There **must** be one criterion block id assigned to each sub-objective.

scenarios

The `scenarios` keyword is used to specify the scenario block identifiers simulating each sub-objective. The syntax for this keyword is:

```
scenarios {integer}{...}
```

There **must** be one scenario block id assigned to each sub-objective.

shape_services

Additional service block identifiers are required for `shape optimization problems`. The shape services are responsible for computing the shape sensitivities for each CAD parameter. The service block identifiers are specifed using the format:

```
shape_services {integer}{...}
```

There **must** be one shape service block id specified for each sub-objective.

weights

The `weights` keyword is used to specify the weights assigned to each sub-objective. The syntax for this keyword is:

```
weights {value}{...}
```

The weights are not required to sum to 1.0. The `default` weight is set to `1.0` for each sub-objective.

2.12 Study

The *study block* begins and ends with the tokens `begin study` and `end study`, respectively. The following text snippet is a simple *study block* setup for a topology optimization problem:

```
begin study
  method topology
  optimization_algorithm mma
  filter_radius_scale 1.75
  max_iterations 25
end study
```

The study keywords can be specified in any order within the study block.

2.12.1 Method

A `method` is used to specify type of design study to be done. The `default` method is set to `topology`. Thus, the numerical methods used for the design study will correspond to those used to solve topology optimization problems. The *method* keyword can be used to select other type of design studies using the format:

```
method {string}
```

Table 2.12.1 lists the supported `methods` within the Morphorm engineering design ecosystem.

Table 2.12.1: Supported design study types.

Method	Description
prune	prunes void elements.
prune_and_topology	prunes void elements and runs a topology optimization problem with the pruned mesh
shape	runs a shape optimization problem
topology	runs a topology optimization problem
analysis	simulates a physical system (no optimization).
design_of_experiment	runs a design of experiment
multidim_parameter_study	runs a multi-dimensional parameter study
surrogate_based_global_optimization	runs a surrogate-based global optimization problem

2.12.2 Analysis

The `analysis method` is used to simulate a physical system. Design criteria can be evaluated as part of the simulation. However, the physical system will not be optimized.

evaluate_criteria

The `evaluate_criteria` keyword is used to specify whether the design criteria will be evaluated by the `analyze` service. The syntax for this keyword is:

```
evaluate_criteria {Boolean}
```

The `default` is set to `false`.

2.12.3 Gradiant-Based Optimization

The following keywords are used to set the parameters for the gradient-based optimization methods. The *topology* and `shape` *methods* are solved using gradient-based optimization algorithms.

General

The following keywords are used to set parameters of the gradient-based optimization algorithms.

optimization_algorithm

The `optimization_algorithm` keyword is used to specify the gradient-based optimizer. The syntax for this keyword is:

```
optimization_algorithm {string}
```

The `default` optimizer is set to `oc`. Table 2.12.2 lists the supported gradient-based optimizers.

Table 2.12.2: Supported gradient-based optimization algorithms.

Keyword	Description
oc	Optimality Criteria [Rozvany]
mma	*Method of Moving Asymptotes* [Svanberg]
umma	Unconstrained Method of Moving Asymptotes [Giraldo-Londono *et al.*]
ksbc	Kelley-Sachs Bound Constrained [Kelley and Sachs]
ksal	Kelley-Sachs Augmented Lagrangian[1]

The `oc`, `mma`, and `ksal` algorithms are suited for constrained optimization problems. The `ksbc` algorithm is suited for bound constrained optimization problems only. The `oc` algorithm is best suited for topology optimization problems with one linear constraint; e.g., a compliance minimization problem with a volume constraint. If the constraints are nonlinear, the `mma` or the `ksal` optimizer must be used. The `umma` is a specialized implementation of the `mma` optimizer for problems with a large number of nonlinear constraints, e.g., a von Mises stress constraint at every material point.

max_iteration

The `max_iteration` keyword is used to specify the maximum number of optimization iterations. The syntax for this keyword is:

```
max_iterations {integer}
```

The `default` value is `100`.

Stopping Criteria

The following keywords are used to set tolerances on the stopping criteria for the gradient-based optimizers. Table 2.12.3 provides a brief description of the primary stopping criteria.

Table 2.12.3: Primary stopping criteria for the gradient-based optimizers.

Criterion	Description
norm of the gradient	euclidean norm of the objective gradient is less than some tolerance.
norm of the descent direction	euclidean norm of the search direction is less than some tolerance.
objective stagnation	difference between two consecutive objective function evaluations is less than some tolerance.
control stagnation	maximum difference between two consecutive design solutions is less than some tolerance.

[1] The *ksal* algorithm uses the augmented Lagrangian method and the *ksbc* algorithm to find a solution to the augmented Lagrangian formulation.

stationarity_tolerance

The `stationarity_tolerance` keyword is used to set the stopping tolerance for the Euclidean norm of the descent direction, i.e., stationarity. The syntax for this keyword is:

```
stationarity_tolerance {value}
```

The `default` tolerance is set to `1e-8`.

norm_gradient_tolerance

The `norm_gradient_tolerance` keyword is used to set the stopping tolerance for the Euclidean norm of the objective gradient, i.e., optimality. The syntax for this keyword is:

```
norm_gradient_tolerance {value}
```

The `default` tolerance is set to `1e-8`.

control_stagnation_tolerance

The `control_stagnation_tolerance` keyword is used to set a stopping tolerance on the solution. If the maximum difference between two consecutive solutions is below the *control_stagnation_tolerance*, the optimizer will stop. The syntax for this keyword is:

```
control_stagnation_tolerance {value}
```

The `default` tolerance is set to `1e-4`.

objective_stagnation_tolerance

The `objective_stagnation_tolerance` keyword is used to set a stopping tolerance on the objective function. If the difference between two consecutive objective function evaluations is below the *objective_stagnation_tolerance*, the optimizer will stop. The syntax for this keyword is:

```
objective_stagnation_tolerance {value}
```

The `default` tolerance is set to `1e-8`.

Operations

The following keywords are used to set specialized operations that can be used with the gradient-based optimizers.

normalize_in_aggregator

The `normalize_in_aggregator` keyword is used to specify whether or not the objective values returned from the analysis services will be normalized by the aggregation operation in the engine before they are passed to the optimizer. The syntax for this keyword is:

```
normalize_in_aggregator {Boolean}
```

The `default` is set to `false`.

update_problem_frequency

The `update_problem_frequency` keyword is used to specify the rate at which the *update problem operation* is called by the optimizer. The syntax for this keyword is:

```
update_problem_frequency {integer}
```

The `default` value is set to `1`, i.e., the optimizer will call the *update problem operation* at every iteration.

Optimizer

The following keywords are used to control the performance of the gradient-based optimizers.

Method of Moving Asymptotes

The following keywords are used to control the performance of the `Method of Moving Asymptotes` and the `Unconstrained Method of Moving Asymptotes` algorithms.

mma_move_limit

The `mma_move_limit` keyword is used to specify the move limit for the `mma` optimizer. The syntax for this keyword is:

```
mma_move_limit {value}
```

The *move limit* controls the step size taken by the optimizer when updating the design variables. The `default` move limit is set to `0.5`.

mma_asymptote_expansion

The `mma_asymptote_expansion` keyword is used to specify a scalar multiplier used to expand the moving asymptotes. The syntax for this keyword is:

```
mma_asymptote_expansion {value}
```

The `default` value is set to `1.2`.

mma_asymptote_contraction

The `mma_asymptote_contraction` keyword is used to specify a scalar multipler used to contract the moving asymptotes. The syntax for this keyword is:

```
mma_asymptote_contraction {value}
```

The `default` value is set to `0.7`.

mma_max_sub_problem_iterations

The `mma_max_sub_problem_iterations` keyword is used to specify the maximum number of iterations for the subproblem. The syntax for this keyword is:

```
mma_max_sub_problem_iterations {integer}
```

The `default` value is set to `50`.

mma_use_ipopt_sub_problem_solver

The `mma_use_ipopt_sub_problem_solver` keyword is used to specify whether to use the interior point optimizer [Wachter and Biegler] rather than the Morphorm `KSAL` optimizer to solve the sub-problem. The syntax for this parameter is:

```
mma_use_ipopt_sub_problem_solver {Boolean}
```

The `default` value is set to `false`.

2.12.4 Topology Optimization

The following keywords are used to set parameters of the numerical methods used to solve topology optimization problems.

Discretization

The following keywords are used to set parameters associated with the discretization method used to discretized the design variables in a topology optimization problem.

initial_density_value

In density-based topology optimization problems, the `initial_density_value` keyword is used to assign uniform value to the density variable at every location. The syntax for this keyword is:

```
initial_density_value {value}
```

The `default` value is set to `0.5`. The *uniform operation* is used to initialize the density variables.

initialize_method

The `initialize_method` keyword is used to specify the method used to initialize the design variables. The syntax for this keyword is:

```
initial_density_value {string}
```

A description of the supported initialization methods is provided in Table 2.12.4. The `uniform` method sets all the design variables to the scalar value provided in the *initial_density_value* keyword. The `uniform_by_region` method permits assignment of a uniform value to all the design variables within a design region, i.e., `exodus` element block. The `default` initialization method is set to `uniform`.

Table 2.12.4: Design variable initialization methods for topology optimization problems.

Method	Description
uniform	uniformly assigns a value for the design variables
uniform_by_region	uniformly assigns a value for the design variables per region

Discretization Type

The *discretization* keyword is used to specify the preferred discretization method for topology optimization. The syntax for this keyword is:

```
discretization {string}
```

The options are: `density` and `levelset`. The `density` option applies density-based numerical methods to solve the topology optimization problem. The `levelset` option applies level set based numerical methods to solve the topology optimization problem. The `default` value is set to `density`. **The ``levelset`` method is under development.**

Filters

The keywords presented in this section are used to set the parameters for the topology optimization filters. Table 2.12.5 lists the supported filters for topology optimization. **We recommend to use a filter radius of at least twice the size of the average finite element in the design domain**.

Table 2.12.5: Supported topology optimization filters.

Keyword	Description
kernel	specifies the kernel filter
helmholtz	specifies the Helmholtz filter

Kernel filter

The kernel filter is defined as:

$$F_{ij} = \max\left(0, 1 - \frac{d(i,j)}{R}\right) \tag{2.12.1}$$

where R is the filter radius, $d(i,j)$ is the distance between the pseudo-densities ρ_i^m and ρ_j^m for candidate material m. The filtered pseudo-density $\bar{\rho}_i^m$ for candidate material m is computed via:

$$\bar{\rho}_j^m = \sum_{i=1}^{N_{pts}} = w_{ij}\rho_i^m \tag{2.12.2}$$

where N_{pts} denotes the number of pseudo-densities inside the filter radius. The weights w_{ij} are computed via:

$$w_{ij} = \frac{F_{ij}}{\sum_{k\in\mathcal{N}_j} F_{kj}} \tag{2.12.3}$$

$\mathcal{N}_j = \{\rho_i^m : d(i,j) \leq R\}$ is the neighborhood of pseudo-densities inside the filter radius R, including the pseudo-densities at the boundary of the search radius, with respect to pseudo-density ρ_j^m. Eq.2.12.1 can take on nonlinear forms too.

Helmholtz filter

The Helmholtz filter, also know as the Partial Differential Equation (PDE) filter, is defined as:

$$\begin{aligned} \delta\tilde{\Pi}(\hat{\mathbf{z}}) &= -\int_\Omega \ell_0^2 \Delta\bar{\rho}\delta\bar{\rho}d\Omega - \int_\Omega (\rho-\bar{\rho})\delta\bar{\rho}d\Omega + \int_\Gamma (\ell_0^2\nabla\bar{\rho}\cdot\mathbf{n} + \ell_s\bar{\rho})d\Gamma = 0 \\ \text{with:}\quad & \ell_0^2\nabla\bar{\rho}\cdot\mathbf{n} = -\ell_s\bar{\rho} \quad \text{on } \Gamma, \end{aligned} \tag{2.12.4}$$

where ℓ_0 is a length scale parameter, ℓ_s is the surface length scale parameter, and Γ denotes the boundary of the domain Ω. The first integral in Eq.2.12.4 aims to smooth highly oscillatory design solutions. The second integral in Eq.2.12.4 aims to keep the filtered design variables $\bar{\rho}$, close to the unfiltered design variables ρ. Meaning, the filtered design variables should not be significantly different than the unfiltered design variables. The compromise between these two goals is regulated through the lenght scale parameter ℓ_0. The Robin boundary condition in Eq.2.12.4 puts a cost on any material placed along the domain boundaries as $\ell_s \to \infty$. Thus, design solutions that adhere to the domain boundaries are discouraged.

General filter keywords

The keywords presented in Table 2.12.6 are used to specify general filter parameters.

Table 2.12.6: Keywords used to set general filter parameters.

Required/Optional	Keyword	Description
Optional	*filter_type*	specifies the filter type
Optional	*filter_radius_scale*	specifies the filter radius as a scaling of the average length of the mesh element edges
Optional	*filter_radius_absolute*	specifies the filter radius as an absolute value
Optional	*filter_in_engine*	specifies whether the design variable filtering will take place in the engine service

filter_type

The `filter_type` keyword is used to specify the type of filter to be used for the topology optimization problem. The syntax for this keyword is:

```
filter_type {string}
```

The supported options are: *kernel* and *helmholtz*. The `default` filter is set to `kernel`.

filter_radius_scale

The `filter_radius_scale` keyword is used to specify the filter radius as a scaling of the average length of the mesh element edges. The syntax for this keyword is:

```
filter_radius_scale {value}
```

The `default` value is set to `2.0`. If the average element edge length is 1.5 and the *filter_radius_scale* is set to 2.0, the resulting filter radius will be 3.0.

filter_radius_absolute

The `filter_radius_absolute` keyword is used to specify the filter radius as an absolute value. The syntax for this keyword is:

```
filter_radius_absolute {value}
```

where the scalar value will be used as the filter radius. The `filter_radius_absolute` parameter **must** be used to set the average length scale for the *helmholtz filter*. The `default` value is set to `1.0` if the `helmholtz` filter is selected. The `filter_radius_scale` options does not work with the *helmholtz filter*.

filter_in_engine

The `filter_in_engine` keyword is used to specify whether the design variable filtering will take place in the `engine` service. The syntax for this keyword is:

```
filter_in_engine {Boolean}
```

The `default` value is set to `true`, which also sets the `default` *filter type* to *kernel*. If the *filter_type* keyword is set to *helmholtz*, the filtering is done by the `analyze` service. Meaning, the `filter_in_engine` keyword is set to `false`.

Note: This legacy feature is available for use cases when a third-party filtering tool is integrated with the Morphorm engineering design ecosystem. This keyword will be retired in the future.

Helmholtz filter keywords

The following keywords are used to set the Helmholtz filter parameters.

boundary_sticking_penalty

The `boundary_sticking_penalty` keyword is used to prevent aggresive aggregation of material along the boundaries of the design. The input value for the *boundary_sticking_penalty* parameter **must** be between `0.0` and `1.0`. The syntax for this keyword is:

```
boundary_sticking_penalty {value}
```

A value of `0.0` indicates no penalization, which means that the Helmholtz filter is free to place material along the boundaries. A value of `1.0` indicates maximum penalization, which means that the Helmholtz filter is going to avoid placing material along the boundaries.

symmetry_plane_location_names

The `symmetry_plane_location_names` keyword is used to inform the optimizer of the geometric surface entities, i.e., `exodus` side sets, where symmetry boundary conditions are applied. The syntax for this keyword is:

```
symmetry_plane_location_names {string}{...}
```

If the *boundary_sticking_penalty* is nonzero, the optimizer will exclude the `exodus` side sets where symmetry boundary conditions are applied from the surface integral term in Eq.2.12.4.

Projection Methods

The following keywords are used to set the parameters for the projection methods.

Types of projection methods

Table 2.12.7 lists the projection methods that can be used together with the *Kernel or Helmholtz filters* to steer the optimizer to a `0-1` design solution when using density-based topology optimization methods.

Table 2.12.7: Projection methods that can be used together with filter methods to steer the optimizer to a "0-1" design solution.

Keyword	Description
kernel_then_heaviside	applies the kernel filter plus the heaviside projection method
kernel_then_tanh	applies the kernel filter plus the hyperbolic tangent projection method
heaviside	applies the heaviside projection filter
tanh	applies the hyperbolic tangent projection filter

kernel_then_heaviside

The `kernel_then_heaviside` keyword is used to instruct the optimizer to use the *kernel filter* with the heaviside projection. The heaviside projection is defined as:

$$\hat{\rho}_i = 1.0 - \exp(-\beta\bar{\rho}_i) + \bar{\rho}_i \exp(-\beta), \tag{2.12.5}$$

where β is the steepness parameter, $\bar{\rho}_i$ is the i-th filtered density, and $\hat{\rho}_i$ is the i-th projected density.

kernel_then_tanh

The `kernel_then_tanh` keyword is used to instruct the optimizer to use the *kernel filter* with the hyperbolic tangent projection method. The hyperbolic tangent projection is defined as:

$$\hat{\rho}_i = \frac{\tanh\left(\beta\eta\right) + \tanh\left(\beta\left(\bar{\rho}_i - \eta\right)\right)}{\tanh\left(\beta\eta\right) + \tanh\left(\beta\left(1.0 - \eta\right)\right)}, \tag{2.12.6}$$

where β is the steepness parameter, $\eta \in [0; 1]$ determines the projection threshold, $\bar{\rho}_i$ is the i-th filtered density, and $\hat{\rho}_i$ is the i-th projected density. The `default` projection threshold η is set to `0.5`.

heaviside

The `heaviside` keyword is used to instruct the optimizer to use the heaviside projection method, i.e., Eq.2.12.5, to filter the densities. This option is typically used with the *Helmholtz filter* to steer the optimizer to a `0-1` design solution when using density-based topology optimization methods.

tanh

The `tanh` keyword is used to instruct the optimizer to use the hyperbolic tangent projection method, i.e., Eq.2.12.6, to filter the densities. This projection method is typically used with the *Helmholtz filter* to steer the optimizer to a `0-1` design solution when using density-based topology optimization methods.

Keywords for the projection methods

Table 2.12.8 lists the keywords used to define the behavior of the projection methods.

Table 2.12.8: Keywords used to define the behavior of the projection methods.

Required/Optional	Keyword	Description
Required	*projection_type*	specifies the projection method
Optional	*filter_heaviside_min*	specifies the initial value for the steepness parameter
Optional	*filter_heaviside_max*	specifies the maximum value for the steepness parameter
Optional	*filter_heaviside_update*	specifies the value by which the steepness parameter increases
Optional	*filter_projection_start_iteration*	specifies the first optimization iteration at which the steepness parameter will be updated
Optional	*filter_projection_update_interval*	specifies the rate at which the steepness parameter will be updated
Optional	*filter_use_additive_continuation*	specifies if an additivate continuation scheme is used to update the steepness parameter

projection_type

The `projection_type` keyword is used to specify the projection method. The syntax for this keyword is:

```
projection_type {string}
```

Table 2.12.7 lists the supported projection methods.

filter_heaviside_min

The `filter_heaviside_min` keyword is used to specify the initial value for the *heaviside steepness parameter* used by the projection methods. The syntax for this keyword is:

```
filter_heaviside_min {value}
```

The `default` value is set to `1.0`. A value near zero results in a near linear projection. As the value increases to infinity, the projection becomes closer to a true heaviside function.

filter_heaviside_max

The `filter_heaviside_max` keyword is used to specify the maximum value for the *heaviside steepness parameter* used by the projection methods. The syntax for this keyword is:

```
filter_heaviside_max {value}
```

The `default` value is set to `50.0`. 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.

filter_heaviside_update

The `filter_heaviside_update` keyword is used to specify the value by which the *heaviside steepness parameter* increases as it increases from *filter_heaviside_min* to *filter_heaviside_max*. The syntax for this keyword is:

```
filter_heaviside_update {value}
```

The `default` value is set to `1.25`.

If the *filter_use_additive_continuation* parameter is set to `false`, the heaviside steepness parameter is updated by **multiplying the current heaviside steepness parameter value** by the *filter_heaviside_update* parameter each time the *update problem operation* is called. Otherwise, the filter heaviside update parameter value is added to the heaviside steepness parameter each time the *update problem operation* is called. The `default` behavior is to multiply the heaviside steepness parameter by the *filter_heaviside_update* parameter, i.e., the *filter_use_additive_continuation* parameter is set to `false`.

filter_projection_start_iteration

The `filter_projection_start_iteration` keyword is used to specify the first optimization iteration at which the heaviside steepness parameter will be updated. The syntax for this keyword is:

```
filter_projection_start_iteration {integer}
```

The `default` value is set to `50`.

filter_projection_update_interval

The `filter_projection_update_interval` keyword is used to specify the rate at which the heaviside steepness parameter will be updated. The syntax for this keyword is:

```
filter_projection_update_interval {integer}
```

The `default` is set to `10`.

filter_use_additive_continuation

The syntax for the `filter_use_additive_continuation` keyword is:

```
filter_use_additive_continuation {Boolean}
```

The `default` value is set to `false`. Thus, a multiplicative update scheme is used to update the steepness parameter β.

The `filter_use_additive_continuation` keyword is used to specify if the heaviside steepness parameter will be increased additively at each update iteration. The following additive update scheme is used at each update iteration:

$$\beta_{new} = \beta_{old} + \alpha, \tag{2.12.7}$$

where α is the value by which the heaviside steepness parameter, β, is increased. The value of α is set using the *filter_heaviside_update* parameter. If the *filter_use_additive_continuation* parameter is set to `false`, the heaviside steepness parameter $beta$ will be updated using the following multiplicative update scheme at each update iteration:

$$\beta_{new} = \alpha \times \beta_{old}, \tag{2.12.8}$$

2.12.5 Shape Optimization

Table 2.12.9 lists the keywords used to set the parameters for shape optimization problems.

Table 2.12.9: Keywords used to set parameters for shape optimization problems.

Required/Optional	Keyword	Description
Required	*csm_file*	specifies the name of the file with the parameterized geometry definition
Required	*num_shape_design_variables*	specifies the number of shape design variables

csm_file

The `csm_file` keyword is used to specify the name of the `csm` file that holds the parameterized geometry definition and the shape parameters to be optimized. The syntax for this keyword is:

```
csm_file {string}
```

The csm file **must** be created with the Engineering Sketch Pad (ESP) application [Haimes and Dannenhoffer], before running the shape optimization problem.

num_shape_design_variables

The `num_shape_design_variables` keyword is used to specify the number of design parameters. The design parameters (shape parameters to be optimized) are set within the Engineering Sketch Pad [Haimes and Dannenhoffer] application. The syntax for this keyword is:

```
num_shape_design_variables {integer}
```

2.12.6 Multi-Dimensional Parameter Study

Table 2.12.10 lists the keywords used to control the performance of the *multi-dimensional parameter study* method.

Table 2.12.10: Supported keywords for a multi-dimensional parameter study.

Required/Optional	Keyword	Description
Optional	*partitions*	specifies the factorial grid of surveyed points
Optional	*link_files*	links files to the work directory
Optional	*copy_files*	copies files to the work directory
Optional	*concurrent_evaluations*	number of concurrent simulation code executions

partitions

The `partitions` keyword is used to specify the factorial grid of surveyed points. The syntax for this keyword is:

```
partitions {integer}{...}
```

The `default` partition in each parameter dimension is set to `5`. The number of function evaluations is computed by adding one to each partition in the parameter space and performing a full factorial combination. For instance, if the grid is defined by a 6×5 grid, the total number of function evaluations equals 42, i.e., $(6 + 1) \times (5 + 1) = 42$.

link_files

The `link_files` keyword is used to link files to the work directory. The syntax for this keyword is:

```
link_files {string}{...}
```

The work directory contains the results for the study, e.g, the results generated by the optimizer and other applications used to evaluate the design criteria.

copy_files

The `copy_files` keyword is used to copy files to the work directory. The syntax for this keyword is:

```
copy_files {string}{...}
```

The work directory contains the results for the study, e.g, the results generated by the optimizer and other applications used to evaluate the design criteria.

concurrent_evaluations

The `concurrent_evaluations` keyword is used to specify the number of concurrent simulation code executions. The syntax for this keyword is:

```
concurrent_evaluations {integer}
```

The `default` number of concurrent evaluations is set to `1`.

2.12.7 Design of Experiments

Table 2.12.11 lists the keywords used to control the performance of the *Design of Experiments* method.

Table 2.12.11: Keywords used to control the performance of the design of experiments method

Required/Optional	Keyword	Description
Optional	*sample_type*	specifies the sampling method
Optional	*num_samples*	specifies the number of samples
Optional	*distribution*	specifies how probabilities and reliability indices are reported
Optional	*sampling_random_seed*	specifies random seed for the sampling method
Optional	*link_files*	links files to the work directory
Optional	*copy_files*	copies files to the work directory
Optional	*concurrent_evaluations*	number of concurrent simulation code executions

sample_type

The `sample_type` keyword is used to specify the sampling method. The syntax for this keyword is:

```
sample_type {string}
```

The supported options are `lhs` and `random`. The `lhs` option uses Latin Hypercube Sampling (LHS) to sample variables. The `random` option uses purely random Monte Carlo sampling. The `default` sampling method is set to `lhs`.

num_samples

The `num_samples` keyword is used to specify the number of samples. The syntax for this keyword is:

```
num_samples {integer}
```

The `default` number of samples is set to `50`.

distribution

The `distribution` keyword is used to specify how probabilities and reliability indices are reported. The syntax for this keyword is:

```
distribution {string}
```

The supported options are `cumulative` and `complementary`. The `cumulative` option computes statistics according to a cumulative function. The `complementary` option computes statistics according to a complementary cumulative function. The `default` distribution is set to `cumulative`.

sampling_random_seed

The `sampling_random_seed` keyword is used to specify the seed for the random number generator used by the sampling method. The syntax for this keyword is:

```
sampling_random_seed {integer}
```

The `default` seed is set to `787`.

Table 2.12.12: Keywords used to control the performance of the surrogate-based global optimization method.

Required/Optional	Keyword	Description
Optional	*max_iterations*	specifies the maximum number of optimization iterations
Optional	*max_function_evaluations*	specifies the maximum number of function evaluations
Optional	*sample_type*	specifies the sampling method
Optional	*num_samples*	specifies the number of samples
Optional	*distribution*	specifies how probabilities and reliability indices are reported
Optional	*num_parents*	specifies the number of parents in random shuffle crossover
Optional	*num_offspring*	specifies the number of offspring in random shuffle crossover
Optional	*mutation_type*	specifies the mutation type for the genetic algorithm
Optional	*mutation_rate*	specifies the probability for a mutation event
Optional	*crossover_rate*	specifies the probability for a crossover event
Optional	*crossover_type*	specifies the crossover type for the genetic algorihtm
Optional	*population_size*	specifies how probabilities and reliability indices are reported
Optional	*moga_random_seed*	specifies the random seed for the global optimizer
Optional	*moga_output*	specifies how much information is written to the terminal and the output file
Optional	*sampling_random_seed*	specifies the random seed for the sampling method
Optional	*link_files*	links files to the work directory
Optional	*copy_files*	copies files to the work directory
Optional	*concurrent_evaluations*	number of concurrent simulation code executions

2.12.8 Surrogate-Based Global Optimization

Table 2.12.12 lists the keywords used to control the performance of the *surrogate-based global optimization method*. The surrogate-based global optimizer uses a sampling algorithm to construct the initial Gaussian process from a set of initial sampling points. The parameters for the *design of experiment algorithm* are used to control the performance and accuracy of the sampling algorithm used by the surrogate-based global optimizer.

max_function_evaluations

The `max_function_evaluations` keyword is used to specify the maximum number of function evaluations before prompting the genetic algorithm to stop. If the other stopping criteria are not reached first, the optimizer will stop after it has performed `max_function_evalutions` evaluations.

```
max_function_evalutions {integer}
```

The `default` value is set to `10000`.

population_size

The `population_size` keyword is used to specify the size of the initial population for the genetic algorithm. The `population_size` keyword only sets the size of the initial population. The population size may vary according to the type of operators chosen for a particular optimization run. The syntax for this keyword is:

```
population_size {integer}
```

The `default` value is set to `500`.

num_parents

The `num_parents` keyword is used to specify the number of parents in random shuffle crossover. The syntax for this keyword is:

```
num_parents {integer}
```

The `default` value is set to `2`.

num_offspring

The `num_offspring` keyword is used to specify the number of offspring in random shuffle crossover. The syntax for this keyword is:

```
num_offspring {integer}
```

The `default` value is set to `2`.

mutation_type

The `mutation_type` keyword is used to specify the mutation method for the genetic algorithm. The syntax for this keyword is:

```
mutation_type {string}
```

The `default` mutation type is set to `replace_uniform`.

Mutation options

bit_random

The `bit_random` mutation introduces random variation by first converting a randomly chosen variable of a randomly chosen candidate into a binary string. It then flips a randomly chosen bit in the string from a 1 to a 0 or visa versa. In this scheme, the resulting value has more probability of being similar to the original value.

replace_uniform

The `replace_uniform` mutation introduces random variation by first randomly choosing a design variable of a randomly selected candidate and reassigning it to a random valid value for that variable. No consideration of the current value is given when determining the new value.

offset_normal

The `offset_normal` mutation introduces random variation by adding a Gaussian random amount to a variable value. The random amount has a standard deviation dependent on the mutation scale, which is set to a `default` value of `0.15`.

offset_cauchy

The `offset_cauchy` mutation introduces random variation by adding a Cauchy random amount to a variable value. The random amount has a standard deviation dependent on the mutation scale, which is set to a `default` value of `0.15`.

offset_uniform

The `offset_uniform` mutation introduces random variation by adding a uniform random amount to a variable value. The random amount depends on the mutation scale, which is set to a `default` value of `0.15`.

mutation_rate

The `mutation_rate` keyword is used to specify the probability for mutation. The syntax for this keyword is:

```
mutation_rate {value}
```

The `default` value is set to `0.1`.

crossover_type

The `crossover_type` keyword is used to specify the crossover method used by the genetic algorithm. The syntax for this keyword is:

```
crossover_type {string}
```

The `default` crossover type is set to `shuffle_random`.

Crossover options

multi_point_binary

The `multi_point_binary` crossover type performs a bit switching crossover at N crossover points in the binary encoded genome of two candidates. Thus, crossover may occur at any point along a solution chromosome.

multi_point_parameterized_binary

The `multi_point_parameterized_binary` crossover type performs a bit switching crossover routine at N crossover points. However, this crossover type performs crossover on each design variable individually. So the individual chromosomes are crossed at N locations.

multi_point_real

The `multi_point_real` performs a variable switching crossover routing at N crossover points in the real valued genome of two candidates. In this scheme, crossover only occurs between chromosomes.

shuffle_random

The `shuffle_random` crossover type performs by selecting chromosomes at random from a specified number of parents enough times that the requested number of offsprings are produced.

crossover_rate

The `crossover_rate` keyword is used to specify the probability for a crossover event. The syntax for this keyword is:

```
crossover_rate {value}
```

The `default` value is set to `0.8`.

moga_output

The `moga_output` keyword specifies how much diagnostic information from the genetic algorithm is written to the terminal and the output files. The syntax for this keyword is:

```
moga_output {string}
```

The `default` value is set to `normal`. Table 2.12.13 lists the supported output options.

Table 2.12.13: Supported output flags for the multi-objective genetic algorithm.

Keyword	Description
debug	extremely verbose output
verbose	more than normal output
normal	normal output behavior
quiet	less than normal output
silent	minimum output

moga_random_seed

The `moga_random_seed` keyword is used to specify the seed for the random number generator assigned to the genetic algorithm. The syntax for this keyword is:

```
moga_random_seed {integer}
```

The `default` seed is set to `10983`.

2.12.9 Output

Table 2.12.14 lists the keywords are used to enable study-related output of results and information.

Table 2.12.14: Keywords used to control study related output.

Required/Optional	Keyword	Description
Optional	*isosurface_output_frequency*	specifies rate at which the isosurface of the optimized design is output
Optional	*shape_study_output_frequency*	specifies rate at which the shape optimized design is output
Optional	*output_method*	specifies how results from a parallel run will be concatenated to a single output file
Optional	*verbose*	specifies whether to print verbose information to console during an optimization run

isosurface_output_frequency

The `isosurface_output_frequency` keyword is used to specify the rate at which the isosurface of the optimized design is output in between topology optimization iterations. The syntax for this keyword is:

```
isosurface_output_frequency {integer}
```

The `default` value is set to `5`.

shape_study_output_frequency

The `shape_study_output_frequency` keyword is used to specify the rate at which the shape optimized design is output. The syntax for this keyword is:

```
shape_study_output_frequency {integer}
```

The `default` value is set to `1`. The output file is an `exodus` mesh, which is written inside the run directory. The name of the file is `Iteration#.exo`, where the number character `#` in the name indicates which iteration it came from.

output_method

The `output_method` keyword is used to specify how results from a parallel run will be concatenated to a single output file. The syntax for this paramter is:

```
output_method {string}
```

The supported 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 into 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. The `default` method is set to `epu`.

verbose

The `verbose` keyword is used to specify whether to print verbose information to console during an optimization run. The syntax for this keyword is:

```
verbose {Boolean}
```

The `default` behavior is set to `false`. If set `true`, information about which stages and operations are being run will be printed to console.

2.12.10 Fixed Block

Table 2.12.15 lists the keywords used to set non-optimizable regions in the design space. These keywords are useful for topology optimization problems.

Table 2.12.15: Keywords used to set non-optimizable regions.

Required/Optional	Keyword	Description
Optional	*fixed_block_ids*	specifies the ids of the non-optimizable input deck block blocks
Optional	*fixed_exodus_nodeset_ids*	specifies the ids of the non-optimizable exodus nodesets
Optional	*fixed_exodus_sideset_ids*	specifies the ids of the non-optimizable exodus sidesets
Optional	*fixed_exodus_nodeset_names*	specifies the names of the non-optimizable exodus nodesets
Optional	*fixed_exodus_sideset_names*	specifies the names of the non-optimizable exodus sidesets

fixed_block_ids

The `fixed_block_ids` keyword is used to specify non-optimizable regions in the design domain. In *topology optimization problems*, the optimizer can add or remove material from the design volume at every material point. The

`fixed_block_ids` is used to specify regions in the design domain where the optimizer cannot remove material. For instance, a component will be mounted on top of another component. Therefore, the optimizer must not remove material from the mounting surface to allow the component to be mounted on the mounting surface.

The syntax for this keyword is:

```
fixed_block_ids {integer}{...}
```

where `{integer}{...}` is a list of the fixed input deck *block block* ids.

fixed_exodus_nodeset_ids

The `fixed_exodus_nodeset_ids` keyword is used to specify the ids of the non-optimizable exodus nodesets. The optimizer cannot add or remove material in these nodesets. The syntax for this keyword is:

```
fixed_exodus_nodeset_ids {integer}{...}
```

where `{integer}{...}` is a list of exodus nodeset ids.

fixed_exodus_nodeset_names

The `fixed_exodus_nodeset_names` keyword is used to specify the names of the non-optimizable exodus nodesets. The optimizer cannot add or remove material in these nodesets. The syntax for this keyword is:

```
fixed_exodus_nodeset_names {string}{...}
```

where `{string}{...}` is a list of exodus nodeset names.

fixed_exodus_sideset_ids

The `fixed_exodus_sideset_ids` keyword is used to specify the ids of the non-optimizable exodus sidesets. The optimizer cannot add or remove material in these sidesets. The syntax for this keyword is:

```
fixed_exodus_sideset_ids {integer}{...}
```

where `{integer}{...}` is a list of exodus sideset ids.

fixed_exodus_sideset_names

The `fixed_exodus_sideset_names` keyword is used to specify the names of the non-optimizable exodus sidesets. The optimizer cannot add or remove material in these sidesets. The syntax for this keyword is:

```
fixed_exodus_sideset_names {string}{...}
```

where `{string}{...}` is a list of exodus sideset names.

2.12.11 Restart

Table 2.12.16 lists the keywords used to enable the solution restart tools.

Table 2.12.16: Keywords used to enable the solution restart tools.

Required/Optional	Keyword	Description
Optional	*write_restart_file*	specifies weather to write the solution restart files
Required	*initial_guess_file_name*	specifies the name of the file containing the initial guess for the restart run
Required	*initial_guess_field_name*	specifies the name of the node field within the initial guess file that will serve as the initial design guess for the restart run
Required	*restart_iteration*	specifies the iteration in the initial guess file that will be used to extract the initial design guess for the restart run

write_restart_file

The `write_restart_file` keyword is used to specify whether to write the solution restart files. The syntax for this keyword 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. The `default` value is set to `false`. If the `write_restart_file` keyword is set to `true`, a restart file will be created inside the run directory, i.e., the directory from where the design study was submitted. The name of the restart exodus file is `engine_restart.exo`.

initial_guess_file_name

The `initial_guess_file_name` keyword is used to specify the name of the file containing the initial guess for the restart run. The syntax for this keyword 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 `engine_output.exo` output file that contains all of the output data from a previous run. This parameter **must** be set if the *write_restart_file* parameter is set to `true`.

initial_guess_field_name

The `initial_guess_field_name` keyword is used to specify the name of the nodal field within the *initial guess file* that contains the the initial design guess for the restart run. The syntax for this keyword is:

```
initial_guess_field_name {string}
```

This parameter **must** be set if the *write_restart_file* parameter is set to `true`.

restart_iteration

The `restart_iteration` keyword is used to specify the iteration in the *file containing the initial guess* that will be used to extract the *initial design guess* for the restart run. The syntax for this keyword is:

```
restart_iteration {integer}
```

This parameter **must** be set if the *write_restart_file* parameter is set to `true`.

2.13 Variables

The *variables block* begins and ends with the tokens `begin variables {string}` and `end variables`, respectively. The string following the `begin variables` expression is an identifier assigned to the *variables block*. Other blocks in the input deck will use this identifier to refer to the `variables block`.

The following text snippet is a typical `variables block` setup:

```
begin variables 1
  numvars 2
  type continuous_design
  initial_points 5e9  0.35
  lower_bounds   1e9  0.30
  upper_bounds   1e10 0.40
  descriptors     E     nu
end variables
```

The `variables block` **must** be defined when using one of these studies:

- *multidimensional parameter study*,
- *design of experiments*,
- *surrogate based design optimization*, or
- *shape optimization*.

The `variables block` is used in these use cases to set bounds and other relevant information pertaining to the optimization variables.

2.13.1 Variables block keywords

Table 2.13.1 lists the keywords used to define a `variables block`. These keywords can be specified in any other within the `variables block`

Table 2.13.1: Keywords used to define a variables block.

Required/Optional	Keyword	Description
Required	*numvars*	specifies the number of optimization variables
Required	*lower_bounds*	specifies the lower bounds for the optimization variables
Required	*upper_bounds*	specifies the upper bounds for the optimization variables
Required	*descriptors*	specifies identifiers assigned to the optimization variables
Required	*means*	specifies the mean values of the distributions used to sample the optimization variables
Required	*std_deviations*	specifies the standard deviation values of the distributions used to sample the optimization variables
Optional	*type*	specifies the type of the optimization variable
Optional	*initial_points*	specifies initial values for the optimization variables

numvars

The `numvars` keyword is used to specify the number of optimization variables for the design study. The syntax for this keyword is:

```
numvars {integer}
```

lower_bounds

The `lower_bounds` keyword is used to specify the lower bounds assigned to the optimization variables. The syntax for this keyword is:

```
lower_bounds {value}{...}
```

upper_bounds

The `upper_bounds` keyword is used to specify the upper bounds assigned to the optimization variables. The syntax for this keyword is:

```
upper_bounds {value}{...}
```

descriptors

The `descriptors` keyword is used to assign identifiers to the optimization variables. The syntax for this keyword is:

```
descriptors {string}{...}
```

The `descriptors` are used as placeholders for the value of the optimization variable in an optimization iteration. For instance, if the descriptors are placed in the `analyze` input deck, the optimizer will replaced the descriptor string with the value of the optimization variable associated with the descriptor. Thus, serving as an automated find and replace tool at runtime.

type

The `type` keyword is used to specify the type of the optimization variable. The syntax for this keyword is:

```
type {string}
```

The `default` value is set to `continuous_design`. The *Variable types* section lists the supported variable types.

initial_points

The `initial_points` keyword is used to specify initial values for the optimization variables. The syntax for this keyword is:

```
initial_points {value}{...}
```

The `default` values are set to the midpoints. The range used to compute the midpoints is defined by the *lower bounds* and the *upper bounds*.

means

The `means` keyword is used to specify the mean values for the optimization variables. The syntax for this keyword is:

```
means {value}{...}
```

The `means` keyword is used when the optimization variables are sampled from a distribution.

std_deviations

The `std_deviations` keyword is used to specify the standard deviations for the optimization variables. The syntax for this keyword is:

```
std_deviations {value}{...}
```

The `std_deviations` keyword is used when the optimization variables are sampled from a distribution.

2.13.2 Variable types

Table 2.13.2 lists the variable types supported within a `variables block`.

Table 2.13.2: Variable types supported within a variables block.

Required/Optional	Keyword	Description
Optional	*continuous_design*	specifies an optimization variable defined by a real interval
Optional	*discrete_design_range*	specifies an optimization variable defined on a range of integer values
Optional	*normal_uncertain*	specifies an optimization variable defined by a normal distribution
Optional	*uniform_uncertain*	specifies an optimization variable defined by an uniform distribution.

continuous_design

The `continuous_design` keyword is used to specify an optimization variable defined by a real interval. Table 2.13.3 lists the keywords used to define a `continuous_design` variable. These keywords can be specified in any order within the `variables block`.

Table 2.13.3: Keywords used to define a `continuous_design` variable.

Required/Optional	Keyword	Description
Required	*descriptors*	specifies the identifiers assigned to the optimization variables
Optional	*lower_bounds*	specifies the lower bounds on the optimization variables
Optional	*upper_bounds*	specifies the upper bounds on the optimization variables
Optional	*initial_points*	specifies the initial values assigned to the optimization variables

discrete_design_range

The `discrete_design_range` keyword is used to specify an optimization variable defined on a range of integer values from the specified lower bound to the specified upper bound. Table 2.13.4 lists the keywords used to define a `discrete_design_range` variable. These keywords can be specified in any order within the `variables block`.

Table 2.13.4: Keywords used to define a `discrete_design_range` variable.

Required/Optional	Keyword	Description
Required	*descriptors*	specifies the identifiers assigned to the optimization variables
Optional	*lower_bounds*	specifies the lower bounds on the optimization variables
Optional	*upper_bounds*	specifies the upper bounds on the optimization variables
Optional	*initial_points*	specifies the initial values assigned to the optimization variables

normal_uncertain

The `normal_uncertain` keyword is used to specify an optimization variable defined by the normal distribution. Table 2.13.5 lists the keywords used to define a `normal_uncertain` variable. These keywords can be specified in any order within the `variables block`. The density function for the normal distribution is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), \tag{2.13.1}$$

where μ is the mean and σ is the standard deviation of the normal distribution. If the normal distribution is bounded, the sampling occurs from the underlying distribution with the given mean and standard deviation, but samples are not taken outside the bounds, see [WJ98]. This can produce a different mean and the standard deviation for the sample data than the mean and standard deviation of the underlying distribution.

When a `normal_uncertain` variable is used with the the design of experiments or multidimensional parameter study methods, distribution bounds are inferred to be $[\mu - 3\sigma, \mu + 3\sigma]$.

Table 2.13.5: Keywords used to define a `normal_uncertain` variable.

Required/Optional	Keyword	Description
Required	*means*	specifies the means of the optimization variables
Required	*std_deviations*	specifies the standard deviations of the optimization variables
Required	*descriptors*	specifies the identifiers assigned to the optimization variables
Optional	*lower_bounds*	specifies the lower bounds on the optimization variables
Optional	*upper_bounds*	specifies the upper bounds on the optimization variables
Optional	*initial_points*	specifies the initial values assigned to the optimization variables

uniform_uncertain

The `uniform_uncertain` keyword is used to specify an optimization variable defined by an uniform distribution. Table 2.13.6 lists the keywords used to define a `uniform_uncertain` variable. These keywords can be specified in any order within the `variables block`. The uniform distribution has the density function:

$$f(x) = \frac{1}{U - L} \tag{2.13.2}$$

where U and L are the upper and lower bounds of the uniform distribution, respectively. The mean is computed as:

$$\mu = \frac{U - L}{2} \tag{2.13.3}$$

and the standard deviation is computed as:

$$\sigma = \frac{(U - L)^2}{12} \tag{2.13.4}$$

Table 2.13.6: Keywords used to define a `uniform_uncertain` variable.

Required/Optional	Keyword	Description
Required	*descriptors*	specifies the identifiers for the optimization variables
Required	*lower_bounds*	specifies the lower bounds on the optimization variables
Required	*upper_bounds*	specifies the upper bounds on the optimization variables

continues on next page

Table 2.13.6 – continued from previous page

Required/Optional	Keyword	Description
Optional	*initial_points*	specifies the initial values assigned to the optimization variables

2.14 Prune

The *prune block* begins and ends with the token `begin prune` and `end prune`, respectively. The following text snippet is a typical `prune block` setup:

```
begin prune
  spatial_dimensions 2
  number_of_refines 1
  number_buffer_layers 1
  prune_threshold 0.6
end prune
```

The input deck can only have one `prune block` defined. The *Prune block keywords* section describes the keywords used to define a `prune block`. These keywords can be specified in any order within the `prune block`.

The `prune` tool reduces computational cost while providing a means to resolve design features. Multiple levels of uniform mesh refinement can be requested to produce smooth and connected topology optimized solutions. Furthermore, the `prune` tool enables pruning of `void` elements, e.g., elements without stiffness, from a previous topology optimized solution. Therefore, a design solution from a previous topology optimization study must exists to use the `prune` tool.

Note: The prune tool at the moment does not work on meshes with mixed element types. For example, the prune tool will not operate on a mesh with tetrahedral and hexahedral elements or a mesh with linear and quadratic tetrahedral elements.

2.14.1 Absolute target versus relative target

The original mass or volume of the starting mesh will change after using the prune tool. If a *relative target mass or volume is used*, the *relative target* must be updated to reflect that the original mesh was pruned. Instead of using a *relative_target*, the *absolute_target* can be used instead. The *absolute_target* is used to specify an absolute target for the constraint. Using an *absolute target* will avoid the need to update the target constraint value every time the mesh is pruned. For example, when constraining the *design volume*, the absolute target volume can be calculated by multiplying the original design volume by the relative target volume, e.g., the volume fraction.

2.14.2 Number of processors

The *number_processors* keyword within the *service block* is used to set the number of processors assigned to the prune service. The following text snippet is a typical *service block* setup for the prune tool:

```
begin service 1
  code prune
  number_processors 4
end service
```

2.14.3 Prune block keywords

Table 2.14.1 lists the keywords that can be specified within a `prune block`.

Table 2.14.1: Keywords supported within the prune block.

Required/Optional	Keyword	Description
Required	*spatial_dimensions*	specifies the spatial dimensions of the parent mesh, i.e., mesh_to_be_pruned
Optional	*prune_path*	specifies the full path to the prune executable
Optional	*pruned_mesh*	specifies the name of the modified mesh produced by the prune tool
Optional	*mesh_to_be_pruned*	specifies the parent mesh from which the void elements will be removed
Optional	*initial_guess_mesh*	specifies the mesh file containing the initial guess for the density or level set field
Optional	*initial_guess_field_name*	specifies the initial guess for the density or level set field used in the prune and refine process
Optional	*number_of_refines*	specifies the number of uniform mesh refinements
Optional	*number_buffer_layers*	specifies the number of buffer element layers that will be added around the pruned mesh
Optional	*prune_threshold*	specifies threshold used to decide whether the element will be kept or pruned

mesh_to_be_pruned

The `mesh_to_be_pruned` keyword is used to specify the name of the parent mesh from which the modified mesh will be generated, i.e., the pruned and possibly refined mesh. The syntax for this keyword is:

```
mesh_to_be_pruned {string}
```

The `default` name is set to the name of the mesh file provided in the *mesh block* using the *name* keyword.

initial_guess_mesh

The `initial_guess_mesh` keyword is used to specify the name of the mesh file containing the initial guess for the density or level set field. The syntax for this keyword is:

```
initial_guess_mesh {string}
```

The `initial_guess_mesh` is typically the file containing the result from a previous design study such as a restart file or the `engine_output.exo` output file. The `default` mesh file is set to `engine_output.exo`.

pruned_mesh

The `pruned_mesh` keyword is used to specify the name of the modified mesh produced by the `prune` tool. The syntax for this keyword is:

```
pruned_mesh {string}
```

The `default` name is set to `pruned_mesh.exo`.

prune_path

The `prune_path` keyword is used to specify the name of the `prune` executable, including the full path to the executable. The syntax for this keyword is:

```
prune_path {string}
```

The `default` path is set to `prune_and_refine`, which assumes that the Morphorm environment is loaded.

initial_guess_field_name

The `initial_guess_field_name` keyword is used to specify the name of the field within the *initial guess file* that contains the density or level set field to be used as the initial guess for the prune and refine process. The syntax for this keyword is:

```
initial_guess_field_name {string}
```

The `default` name is set to `control`, which are the unfiltered design variables. If the initial guess field is set to `topology`, the filtered design variables will be used as the initial guess for the prune and refine process.

number_of_refines

The `number_of_refines` keyword is used to specify the number of uniform mesh refinements that will be done as part of the prune and refine operation. The syntax for this keyword is:

```
number_of_refines {integer}
```

The `default` number of uniform refinements is set to `0`, i.e., no refinement.

number_buffer_layers

The `number_buffer_layers` keyword is used to specify the number of buffer element layers that will be added around the pruned mesh during a prune and refine operation. The syntax for this keyword is:

```
number_buffer_layers {integer}
```

The `default` number of buffer element layers is set to `1`.

prune_threshold

The `prune_threshold` keyword is used to specify the threshold used to decide whether the element will be kept or pruned. Elements with values larger than the threshold will be kept while elements with values below the threshold will be pruned. The syntax for this keyword is:

```
prune_threshold {value}
```

The `default` threshold is set to `0.5`.

spatial_dimensions

The `spatial_dimensions` keyword is used to specify the spatial dimensions of the parent mesh, i.e., the mesh file provided in the *mesh_to_be_pruned* keyword. The syntax for this keyword is:

```
spatial_dimensions {integer}
```

The `default` behavior is to use the value assigned to the *dimensions* keyword within the *scenario block*. If the *prune method* is selected, the `spatial_dimensions` keyword within the `prune block` **must** be set.

2.15 Output

Each *output block* begins and ends with the tokens `begin output {string}` and `end output`, respectively. The string following the `begin output` expression specifies an identifier assigned to the *output block*. Other blocks in

the input deck will use this identifier to refer to the `output block`. One `output block` is required for each of the services writing output quantities of interest. The following is a typical `output block` setup:

```
begin output 1
   service 2
   data temperature
end output
```

Multiple `output blocks` are allowed. However, this advanced feature is rarely used. Typically, only one `output block` is required since only one *analysis service*, e.g., `analyze`, is used at runtime.

2.15.1 Output block keywords

Table 2.15.1 lists the keywords to write the output quantities of interest to the output files. These output keywords can be specified in any order within the `output block`.

Table 2.15.1: Keywords used to define an output block.

Required/Optional	Keyword	Description
Required	*service*	specifies the service block identifier of the service writing the output quantities of interest
Required	*data*	specifies the identifiers for the output quantities of interes
Optional	*writevars*	specifies the identifiers assigned to the writevars blocks
Optional	*output_frequency*	specifies the rate at which output data is written to file
Optional	*native_service_output*	specifies if the analysis service will write the output data
Optional	*native_service_output_directory*	specifies the directory where the analysis service will save the output files

service

The `service` keyword is used to specify the service block identifier (id) of the service computing the output quantities of interests. The syntax for this keyword is:

```
service {integer}
```

data

The `data` keyword is used to specify the output quantities of interests. The syntax for this keyword is:

```
data {string}{...}
```

Table 2.15.2 lists the identifiers used for the output quantity of interest. The `exodus` output file created by the Morphorm software can be visualized in any visualization tool that supports the exodus file format, e.g., Paraview.

Table 2.15.2: Identifiers used for the output quantities of interests.

Keyword	Description	Location
dispx	displacement in the x-direction	node field
dispy	displacement in the y-direction	node field
dispz	displacement in the z-direction	node field
temperature	temperature	node field
joule_heating	joule heating	node field
temperature_gradient	temperature gradient	node field
electric_potential	electric potential	node field
electric_field	electric field	node field
stress	stress tensor components	element field
strain	strain tensor components	element field
vonmises	von Mises stress	element field
thermal_flux	thermal flux	element field
electric_flux	electric flux	element field
current_density	current density	element field

writevars

The `writevars` keyword is used to specify the *writevars blocks* identifiers. The syntax for this keyword is:

```
writevars {string}{...}
```

Multiple writevars blocks can be defined in the input deck. However, only the *writevars blocks* requested in the `writevars` keyword will be considered for output.

native_service_output

The `native_service_output` keyword is used to specify if `analyze` will write the `exodus` output files containing the output quantities of interest. The syntax for this keyword is:

```
native_service_output {Boolean}
```

If the `native_service_output` keyword is set to `true`, the `exodus` output files are written to a directory. The new directory is created inside the working/run directory. The `default` name used for the new directory is `native_analyze_$ID_output`, where `$ID` denotes the *service block* identifier specify in the *output service* keyword. The name assigned to the `exodus` output files in the `native_analyze_$ID_output` directory used the following file name convention, `iteration$ITR`, where `$ITR` denotes the optimization iteration. The `native_service_output` is disabled by `default`, i.e., `native_service_output false`.

native_service_output_directory

The *native_service_output_directory* keyword is used to specify a suffix for the `default` output directory name, `native_analyze_$ID_output`, see the description for the *native_service_output* keyword.

```
native_service_output_directory {string}
```

For example, if the `native_service_output_directory` keyword is set tp `thermal_load`, the name of the output directory will be set to `native_analyze_$ID_output_thermal_load`, where `$ID` denotes the *service block* identifier specify in the *output service* keyword. The `default` value is an empty string.

output_frequency

The `output_frequency` keyword is used to specify how often the optimization results and the output quantities of interest will be written to file. The syntax for this keyword is:

```
output_frequency {integer}
```

Results are in the form of an `exodus` file and are saved in the *output directory*. The `default` value is set to `1`. Therefore, the optimization results and output quantities of interest are written at every major optimization iteration.

2.16 Writevars

Each *writevars block* begins and ends with the tokens `begin writevars {string}` and `end writevars`, respectively. The string following the `begin writevars` expression specifies an identifier assigned to the `writevars block`. Other blocks in the input deck will use this identifier to refer to the `writevars block`. The following is a typical `writevars block` setup:

```
begin writevars 1
  variable dispy
  entity_ids 8, 88, 108
end writevars
```

The `writevars block` is used to request that `analyze` writes one or more output quantities of interest at one or more *requested locations* in to text files. A text file will be created for each of the quantities of interests requested in the *variable* keyword.

2.16.1 Writevars block keywords

Table 2.16.1 lists the keywords used to define a `writevars block`. These keywords can be specified in any order within the `writevars block`.

Table 2.16.1: Keywords used to define a writevars block.

Required/Optional	Keyword	Description
Required	*variable*	specifies the output quantities of interest to be written to text files
Required	*entity_ids*	specifies the output location of interest to be written to text files

variable

The `variable` keyword is used to specify the output quantities of interest to be written to text files. The syntax for this keyword is:

```
variable {string}{...}
```

A text file will be created for each of the quantities of interests requested in the *variable* keyword. Table 2.16.2 lists the quantities of interest that can be written to a text file and the name assigned to the output text file based on the quantity of interest.

Table 2.16.2: Quantities of interest that can be written to a text file and the name assigned to the output text file.

Keyword	Description	Filename
dispx	displacement in the x-direction	displacement_x.dat
dispy	displacement in the y-direction	displacement_y.dat
dispz	displacement in the z-direction	displacement_z.dat
temperature	temperature	temperature.dat
electric_potential	electric potential	electric_potential.dat

entity_ids

The `entity_ids` keyword is used to specify the entity locations of interest, e.g., nodes, that will be written to the text files. The syntax for this keyword is:

```
entity_ids {integer}{...}
```

2.17 Mesh

The *mesh block* begins and ends with the tokens `begin mesh` and `end mesh`, respectively. The following text snippet is a typical `mesh block` setup:

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

where the *name* keyword is used to specified the name of the `exodus` mesh file. Table 2.17.1 list the supported element types for analysis.

Table 2.17.1: Supported element types for two- and three-dimensional problems.

Element	Spatial Dimensions	Description
TRI3	2D	linear triangular element
TRI6	2D	quadratic triangular element
QUAD4	2D	linear quadrilateral element
QUAD8	2D	quadratic quadrilateral element
TET4	3D	linear tetrahedral element
TET10	3D	quadratic tetrahedral element
HEX8	3D	linear hexahedral element
HEX27	3D	quadratic hexahedral element

2.17.1 name

The `name` keyword is used to specify the name of the `exodus` mesh file. The syntax for this keyword is:

```
name {string}
```

The `name` keyword **must** be defined.

2.18 Linear Solver

Each *linear solver block* begins and ends with the tokens `begin linear_solver {string}` and `end linear_solver`, respectively. The string following the `begin linear_solver` expression is an identifier (id) assigned to the *linear_solver block*. Other blocks in the input deck will use this identifier to refer to the `linear_solver block`. The following is a typical `linear_solver block` setup:

```
begin linear_solver 1
  solver_package tpetra
  max_iterations 500
  tolerance 1e-12
end linear_solver
```

The input deck can contain an arbitrary number of linear solver blocks. The `linear_solver block` *keywords* can be specified in any order within the `linear_solver block`. **The linear solver block is an optional input**. The `default` solver for serial and parallel analysis use cases are the Trilinos Tacho package and the Trilinos Tpetra package, respectively.

The `linear_solver block` identifier must be assigned to a *scenario block* using the *linear_solver* keyword. If the `linear_solver block` identifier is not assigned to a `scenario block`, the solver parameter values set within the linear solver block will not be written in the input deck of the *analyze service* evaluating the scenario. See the *Scenarios* section to learn how to assign the `linear_solver block` identifier to a `scenario block` using the *linear_solver* keyword.

2.18.1 Linear solver block keywords

Table 2.18.1 lists the keywords used to define a `linear_solver block`.

Table 2.18.1: Keywords used to set the solver package for analyze.

Required/Optional	Keyword	Description
Optional	*solver_package*	specifies the solver package used to solve the linear system of equations

solver_package

The `solver_package` keyword is used to specify the solver package used to solve the linear system of equations. The syntax for this keyword is:

```
solver_package {string}
```

The supported solver packages are the Trilinos Tpetra solver and the Trilinos Tacho solver. The `default` solver package for serial jobs is set to `tacho`. The `default` solver package for parallel jobs is set to `tpetra`.

Note: The Trilinos Tacho solver does not support distributed memory parallelism. Therefore, it can only be used to run serial jobs.

Tpetra Solver

Table 2.18.2 lists the keywords used to set the Tpetra solver parameters. These keywords can be specified in any order within the `linear_solver block`.

Table 2.18.2: Keywords used to set the Tpetra solver parameters.

Required/Optional	Keyword	Description
Optional	*max_iterations*	specifies the maximum number of iterations
Optional	*tolerance*	specifies the solver convergence tolerance
Optional	*preconditioner_package*	specifies the preconditioner package
Optional	*iterative_method*	specifies the iterative method use to solve the linear system of equations

Tpetra [Siefert] is "hybrid parallel," meaning that it uses at least two levels of parallelism:

- MPI (the Message Passing Interface) for distributed-memory parallelism
- Any of various shared-memory parallel programming models within an MPI process

Tpetra uses the Kokkos [Mackey *et al.*] package's shared-memory parallel programming model for data structures and computational kernels. Kokkos makes it easy to port Tpetra to new computer architectures, and to extend its use of parallel computational kernels and thread-scalable data structures. Kokkos supports several different parallel programming models, including

- OpenMP for shared-memory multi-processing programming
- Nvidia's CUDA programming model for graphics processing units (GPUs)

max_iterations

The `max_iterations` keyword is used to specify the maximum number of major linear solver iterations. The syntax for this keyword is:

```
max_iterations {integer}
```

The `default` value is set to `1000`.

tolerance

The `tolerance` keyword is used to specify the solver convergence tolerance. The syntax for this keyword is:

```
tolerance {value}
```

The iterative solver will iterate until the convergence tolerance is below this threshold. The `default` value is set to `1e-12`.

preconditioner_package

The `preconditioner_package` keyword is used to specify the preconditioner package. The syntax for this keyword is:

```
preconditioner_package {string}
```

The preconditioner package options are `muelu` and `ifpack2`. The `default` preconditioner package is set to `muelu`. Please consult the Trilinos documentation to learn more about the Trilinos preconditioner packages.

iterative_method

The `iterative_method` keyword is used to specify the iterative method use to solve the linear system of equations. The syntax for this keyword is:

```
iterative_method {string}
```

The supported iterative methods are `pseudoblock_gmres` and `pseudoblock_cg`. The `default` iterative method is set to `pseudoblock_gmres`. Please consult the Trilinos documentation to learn more about the Trilinos iterative methods.

Tacho Solver

The following keywords are used to control the performance of the Trilinos Tacho solver package. Tacho [Kim *et al.*] is a shared-memory parallel Cholesky factorization based direct solver that uses Kokkos based tasking.

factorization_type

The `factorization_type` keyword is used to specify the factorization method use to solve the linear system of equations. The syntax for this keyword is:

```
factorization_type {string}
```

The supported factorization methods are `cholesky`, `symlu`, and `ldlt`. The `default` factorization type is set to `cholesky`. Please consult the Trilinos documentation to learn more about the Trilinos factorization methods.

2.19 Newton-Raphson

The *newton_raphson block* begins and ends with the tokens `begin newton_raphson {string}` and `end newton_raphson`, respectively. The string following the `begin newton_raphson` expression denotes an identifier for the `newton_raphson block`. Other blocks in the input deck will use this identifier to refer to the `newton_raphson block`. The following is a typical `newton_raphson block` setup:

```
begin newton_raphson 1
  residual mechanical
  max_iterations 10
  residual_tolerance 1e-12
  increment_tolerance 1e-12
end newton_raphson
```

Default values are set for all the main `newton_raphson` parameters. However, these values can be adjusted.

In multi-physics use cases, a `newton_raphson block` **must** be defined for each residual that makes up the multi-physics analysis if the user wants to control the performance of each Newton-Raphson solver. The following text snippet is a typical multi-physics setup:

```
begin newton_raphson 1
  residual mechanical
  max_iterations 25
  residual_tolerance 1e-10
  increment_tolerance 1e-10
end newton_raphson

begin newton_raphson 2
```

(continues on next page)

(continued from previous page)

```
  residual thermal
  max_iterations 10
  residual_tolerance 1e-8
  increment_tolerance 1e-8
end newton_raphson
```

2.19.1 Newton-Raphson block keywords

The following keywords can be specified in any order within the `newton_raphson block`. Consult the *Scenarios* section to learn how to assign a `newton_raphson block` identifier to a `scenario block` using the *newton_raphson* keyword. Table 2.19.1 lists the keywords used to define a `newton_raphson block`. These keywords can be specified in any order within the `newton_raphson block`.

Table 2.19.1: Keywords used to define a newton_raphson block.

Required/Optional	Keyword	Description
Required	*residual*	specifies the service simulating the scenario
Optional	*max_iterations*	specifies the maximum number of iterations
Optional	*num_restarts*	specifies the number of restarts allowed for one force increment
Optional	*num_increments*	specifies the number of force increments
Optional	*residual_tolerance*	specifies the residual convergence tolerance
Optional	*increment_tolerance*	specifies the increment convergence tolerance

residual

The `residual` keyword is used to specify which residual is using the Newton-Raphson solver. The syntax for this keyword is:

```
residual {string}
```

Table 2.19.2 lists the identifiers used to refer to the residuals.

Table 2.19.2: Identifiers used to refer to the residuals.

Residual	Description
thermal	thermal residual
mechanical	mechanical residual
electrical	electrical residual

max_iterations

The `max_iterations` keyword is used to specify the maximum number of minor solver iterations permitted during a major Newton-Raphson iteration. The syntax for this keyword is:

```
max_iterations {integer}
```

A major iteration is associated with a solver restart iteration at a given force increment. The minor iteration is associated with a Newton-Raphson solver iteration. The `default` value is set to `20`.

num_restarts

The `num_restarts` keyword is used to specify the number of restarts allowed for one force increment. The syntax for this keyword is:

```
num_restarts {integer}
```

The `default` value is set to `5`.

num_increments

The `num_increments` keyword is used to specify the number of force increments. The syntax for this keyword is:

```
num_increments {integer}
```

The `default` value is set to `100`. An automatic force increment scheme is used by the Newton-Raphson solver. If the solution diverges, the solver starts again with the increment size set to 25% of its previous value. An attempt is then made at finding a converged solution with the smaller increment. If the increment still fails to converge, the solver reduces the increment size again. By `default`, the solver allows a maximum of `five` cutbacks, i.e., restart iterations, before stopping the analysis. If the increment `converges` in less than `five` Newton-Raphson iterations, the increment size is set to two times its previous value.

residual_tolerance

The `residual_tolerance` keyword is used to specify the residual convergence tolerance, which is defined as the Euclidean norm of the misfit between the internal and external forces. The syntax for this keyword is:

```
residual_tolerance {value}
```

The `default` value is set to `1e-8`.

increment_tolerance

The `increment_tolerance` keyword is used to specify the increment convergence tolerance, which is defined as the Euclidean norm of the misfit between the current and previous increments. The syntax for this keyword is:

```
increment_tolerance {value}
```

The `default` value is set to `1e-8`.

CHAPTER

THREE

DIAGNOSTICS

The *Diagnostics* chapter describes the tools used to monitor the progress of a study.

3.1 Simulation Study

Table 3.1.1 provides a brief description of the output files created by the Morphorm software when running a simulation.

3.1.1 Output Fields

The Morphorm software writes an exodus file in the run directory that contains the *output physical fields* computed by the simulation. The name of the output exodus file is `output_data_scenario_id_{scid}.exo`, where `{scid}` is the id assigned to the scenario. The output exodus file can be used to generate animations or images in a visualization tool that supports the exodus format, e.g., Paraview [Ahrens *et al.*] [Ayachit *et al.*].

Table 3.1.1: Output files created in the run directory when running a simulation.

File	Description
output_data_scenario_id_{scid}.exo	records the relevant physical fields computed by the simulation
output_scenario_id_{scid}_criteria_history.csv	records criteria values

3.1.2 Criteria

If one or more design criteria are defined in the input deck and the *evaluate_criteria* keyword is set to `true`, the simulation code will write a text file in the run directory with the criteria values computed by the simulation. The name of the output text file is `output_scenario_id_{scid}_criteria_history.csv`.

3.2 Optimization Study

The Morphorm software provides a collection of tools to monitor the progress of the optimization problem. Multiple data files in the form of text files are written in the run directory to monitor the criteria function evaluations, the convergence criteria for the gradient-based optimizer, and the evolution of the design variablesi and physical quantities of interest.

3.2.1 Criteria Histories

Table 3.2.1 lists the files created by the software to monitor the criteria and sub-criteria evaluations.

Table 3.2.1: Output data files holding the evaluation histories for the design criteria.

File	Description
opt_criteria_history.csv	records the evaluation histories for the design criteria
iteration{itr}_criteria_history.csv	records the evaluation histories for the design criteria

opt_criteria_history.csv

The `opt_criteria_history.csv` file records the evaluation histories for the objective function(s) and constraint function(s). The file is written in the run directory at the end of the optimization study.

iteration{itr}_criteria_history.csv

The `iteration{itr}_criteria_history.csv` file records the evaluation histories for the objective function(s) and constraint function(s) up to major optimization iteration `{itr}`. The `{itr}` key is replaced by the major optimization iteration. The file is written inside the `analyze_id_{svid}_output_scenario_id_{scid}` directory, which is created in the run directory by the optimizer at runtime. The `{svid}` and `{scid}` keys are replaced by the service id and the scenario id, respectively. A criterion **must** be evaluated by the `analyze service` to be recorded in the `iteration{itr}_criteria_history.csv` file.

The `default` behavior is to write the `iteration{itr}_criteria_history.csv` file at each major optimization iteration. The *output_frequency* keyword in the *output block* can be used to specify a rate at which the `iteration{itr}_criteria_history.csv` file will be written to disk.

3.2.2 Gradient-Based Optimizer

Table 3.2.2 lists the files created by the software to monitor the convergence criteria for the gradient-based optimizers.

Table 3.2.2: Progress reports for the gradient-based optimizers.

File Type	Description
morphorm_oc_optimizer_diagnostics.txt	progress report for the optimality criteria optimizer
morphorm_mma_optimizer_diagnostics.txt	progress report for the method of moving asymptotes optimizer
morphorm_umma_optimizer_diagnostics.txt	progress report for the unconstrained method of moving asymptotes optimizer

Keywords

Table 3.2.3 describes the keywords used in the progress reports to denote optimalizty and feasibility conditions.

Table 3.2.3: Keywords used to report convergence metrics for the gradient-based optimizers.

File Type	Description
Iter	optimization iteration
F-count	number of objective function evaluations
F(X)	objective function value
Norm(F`)	euclidean norm of the objective function gradient
H_i	constraint function value(s), where $i = 1, \ldots, N_h$ and N_h denotes the number of constraints
abs(dX)	maximum difference between two consecutive solutions, defined as $\mid \max(X_n - X_{n-1}) \mid$, where X is the solution and n is the current optimization iteration.
abs(dF)	maximum difference between two consecutive objective function evaluations, defined as $\mid \max(F_n - F_{n-1}) \mid$, where F is the objective function and n is the current optimization iteration.

Examples

Method of Moving Asymptotes

The following is a typical representation of the information presented in the progress report for the Method of Moving Asymptotes optimizer:

```
Iter   F-count          F(X)        Norm(F`)         H1(X)         abs(dX)         abs(dF)
0          1        1.509010e+00   2.182163e-04   8.500000e-01   1.000000e+00  1.
↪797693e+308
1          2        1.522258e+00   7.447184e-04   5.673034e-01   3.015659e-01   1.324748e-
↪02
2          3        1.572988e+00   3.810290e-03   2.846064e-01   3.015666e-01   5.073029e-
↪02
3          4        5.932018e+00   1.319001e+00  -5.462958e-02   3.618786e-01   4.
↪359030e+00
4          5        2.052045e+02   1.311323e+03  -4.028518e-02   4.999973e-01   1.
↪992725e+02
5          6        1.101845e+03   8.101313e+02  -6.948786e-02   5.000000e-01   8.
↪966404e+02

****** Optimization stopping due to exceeding maximum number of iterations. ******
```

Unconstrained Method of Moving Asymptotes

The following is a typical representation of the information presented in the progress report for the unconstrained Method of Moving Asymptotes (UMMA) optimizer:

```
Iter   F-count     G-count        F(X)         Norm(F`)         abs(dX)         abs(dF)
1          1           1      1.335326e+01   1.913219e+00   5.000000e-01   1.435326e+01
1          2           2      4.008026e+00   1.034668e+00   4.928680e-02   9.345231e+00
1          3           3      1.614877e+00   5.236304e-01   4.541994e-02   2.393149e+00
1          4           4      1.037318e+00   3.124713e-01   3.963437e-02   5.775599e-01
1          5           5      8.008150e-01   2.738359e-01   3.475477e-02   2.365026e-01
2          6           6      6.374185e-01   2.001995e-01   3.188472e-02   1.633965e-01
2          7           7      5.995462e-01   2.567796e-01   2.938482e-02   3.787226e-02
2          8           8      5.244458e-01   1.707753e-01   2.763170e-02   7.510035e-02
```

(continues on next page)

(continued from previous page)

```
2         9          9      5.283567e-01   2.475776e-01   2.803431e-02   3.910832e-03
2        10         10      4.881357e-01   1.738571e-01   2.607276e-02   4.022093e-02
:         :          :             :              :              :              :
:         :          :             :              :              :              :
38      186        186       2.126541e-01   6.022419e-02   1.248030e-03   1.700005e-04
38      187        187       2.124479e-01   6.029219e-02   1.140760e-03   2.062125e-04
38      188        188       2.122737e-01   6.003132e-02   8.741588e-04   1.742000e-04
38      189        189       2.121252e-01   5.980237e-02   7.884139e-04   1.484520e-04
38      190        190       2.119875e-01   6.017124e-02   7.752297e-04   1.376968e-04
38      191        191       2.118605e-01   5.990889e-02   8.343624e-04   1.270040e-04

****** Optimization stopping due to control (i.e. design variable) stagnation. ******
```

The UMMA optimizer is tailored for design studies modeling a large number of constraints, e.g., one or more constraints per element.

Optimality Criteria

The following is a typical representation of the information presented in the progress report for the Optimality Criteria optimizer:

```
Iter   F-count          F(X)        Norm(Fi`)         H1(X)         abs(dX)        ␣
↪abs(dF)
0          1       7.865749e-02   8.373826e-03   1.000000e-01   5.000000e-01   1.
↪797693e+308
1          2       8.516421e-02   8.775654e-03   9.264577e-06   2.000000e-01   6.506714e-
↪03
2          3       5.173293e-02   4.241214e-03  -1.401057e-05   2.000000e-01   3.343127e-
↪02
3          4       3.834720e-02   2.889830e-03  -9.099773e-06   2.000000e-01   1.338574e-
↪02
4          5       3.321951e-02   2.365102e-03   1.078016e-05   2.000000e-01   5.127687e-
↪03
5          6       3.079841e-02   2.156470e-03  -2.076419e-05   2.000000e-01   2.421106e-
↪03

****** Optimization stopping due to exceeding maximum number of iterations. ******
```

3.2.3 Visualization File

Table 3.2.4 provides a brief description of the visualization files created by the Morphorm software when running an optimization study. The optimizer records the evolution of the design variables in an exodus file. The name of the exodus file is `engine_output.exo`. For instance, in a density-based topology optimization study, the `engine_output.exo` file holds the history for the optimized material layout at each major optimization iteration. The material layout history can be used to generate animations in a visualization tool that supports the exodus format, e.g., Paraview [Ahrens *et al.*] [Ayachit *et al.*].

Table 3.2.4: Visualization files created when running an optimization study.

File	Description
engine_output.exo	holds the optimized material layout history
iteration{itr}.exo	holds the optimized material layout and the physical quantity of interest fields for the current major optimization iteration

In addition to the exodus file written by the optimizer in the run directory, the simulation code writes an additional exodus file to record the evolution of the design variables and the physical quantities of interests. The name of the of the exodus file written by the simulation code is `iteration{itr}.exo`, where `{itr}` key is replaced by the major optimization iteration. The `default` behavior is to write the exodus file at each major optimization iteration. However, an output frequency can be specified to manage the rate at which the simulation code writes the exodus file. Furthermore, the file is written inside the `analyze_id_{svid}_output_scenario_id_{scid}` directory, which is created in the run directory. The `{svid}` and `{scid}` keys are replaced by the service id and the scenario id, respectively.

CHAPTER

FOUR

BIBLIOGRAPHY

The *Bibliography* chapter lists the references used in this manuscript.

4.1 Bibliography

BIBLIOGRAPHY

[AGL05] James Ahrens, Berk Geveci, and Charles Law. ParaView: an end-user tool for large data visualization. In *Visualization Handbook*. Elesvier, 2005.

[ABG+15] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. Paraview catalyst: enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV 2015)*, 25–29. November 2015. doi:10.1145/2828612.2828624.

[GLAP21] Oliver Giraldo-Londono, Miguel A Aguilo, and Glaucio H Paulino. Local stress constraints in topology optimization of structures subjected to arbitrary dynamic loads: a stress aggregation-free approach. *Structural and Multidisciplinary Optimization*, 64:3287–3309, 2021.

[HD13] Robert Haimes and John Dannenhoffer. The engineering sketch pad: a solid-modeling, feature-based, web-enabled system for building parametric geometry. In *21st AIAA Computational Fluid Dynamics Conference*, 3073. 2013.

[KS99] Carl T Kelley and Ekkehard W Sachs. A trust region method for parabolic boundary control problems. *SIAM Journal on Optimization*, 9(4):1064–1081, 1999.

[KER18] Kyungjoo Kim, Harold C Edwards, and Sivasankaran Rajamanickam. Tacho: memory-scalable task parallel sparse cholesky factorization. Technical Report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018. URL: https://www.osti.gov/biblio/1515732, doi:10.1109/IPDPSW.2018.00094.

[MPS+22] Greg Mackey, Amy Powell, Daniel Sunderland, Mark Hoemmen, Sivasankaran Rajamanickam, Christian Trott, Stan Moore, Stan Bova, Harold Edwards, James Fourcar, Jeremiah Wilke, David Hollman, Kyungjoo Kim, Simon Hammond, Nicolas Morales, David Poliakoff, Brian Kelley, Nathan Ellingwood, Mehmet Deveci, Daniel Ibanez-Granados, Luc Berger-Vergiat, Jennifer Loe, Vinh Dang, Jeffery Miles, Seher Acer, Ichitaro Yamazaki, Dong Lee, Jan Ciesko, Evan Harvey, Kim Anne Liegeois, Carl Pearson, Francesco Rizzi, Cezary Skrzynski, Phil Miller, Nicholas Curtis, Nevin Liber, Damien Lebrun-Grandie, Bruno Turcksin, Daniel Arndt, Rahul Gayatri, Jonathan Madsen, Mikael Simberg, Chip Freitag, Advanced Micro Devices, Lawrence Berkley National Laboratory, AMD, Swiss National Supercomputing Centre, NexGen Analytics, Oak Ridge National Laboratory, Argonne National Laboratories, and USDOE. Kokkos v.4.0, version 4.0. Technical Report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 11 2022. URL: https://www.osti.gov/biblio/2372711, doi:10.11578/dc.20240607.1.

[Roz12] George IN Rozvany. *Structural design via optimality criteria: the Prager approach to structural optimization*. Volume 8. Springer Science & Business Media, 2012.

[Sie22] Christopher Siefert. Tpetra in fy23. Technical Report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2022. URL: https://www.osti.gov/biblio/2005730, doi:10.2172/2005730.

[Sja17] Gregory D. Sjaardema. Sandia engineering analysis code access system v. 2.0.1, version 00. 10 2017. URL: https://www.osti.gov//servlets/purl/1398883.

[SSD+22] Clint Stimpson, Gregory Sjaardema, Clifton Dudley, Robert Yorgason, Byron Hanks, David White, Robert Kerr, Scott Mitchell, Jason Shepherd, Timothy Tautges, Darryl Melander, Trevor Hensley, Paul Stallings, Teddy Blacker, Michael Borden, Brett Clark, Leslie Fortier, Corey McBride, Philippe Pebay, Matthew Staten, Craig Vineyard, Kevin Pendley, Steven Owen, Corey Ernst, William Quadros, Neal Grieb, Michael Plooster, Mark Richardson, Steve Storm, John Fowler, Mike Stephenson, Rammagy Yoeu, Karl Merkley, Ray Meyers, Mark Dewey, Sara Richards, Randy Morris, Ved Vyas, Boyd Tidwell, Erick Johnson, Mike Parrish, Adam Woodbury, John Malone, Jane Hu, Ben Aga, Andrew Rout, and USDOE. Cubit v.16.x, version 16.x. 8 2022. URL: https://www.osti.gov/biblio/1894127, doi:10.11578/dc.20221017.1.

[Sva87] Krister Svanberg. The method of moving asymptotes—a new method for structural optimization. *International journal for numerical methods in engineering*, 24(2):359–373, 1987.

[WB06] Andreas Wachter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 2006.

[WJ98] Gregory D Wyss and Kelly H Jorgensen. A users guide to lhs: sandias latin hypercube sampling software. Technical Report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 1998.