SANDIA REPORT
SAND2007-3749
Unlimited Release
Printed May 2007

Presto User's Guide 2.7 (Version 1)

J. Richard Koteras, Arne S. Gullerud, Nathan K. Crane, Jason D. Hales, and Rhonda K. Reinert

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2007-3749xxxx
Unlimited Release
Printed May 2007

Presto Users Guide 2.7 (Version 1)

J. Richard Koteras, Arne S. Gullerud, Nathan K. Crane,
Jason D. Hales, and Rhonda K. Reinert
Computational Solid Mechanics and Structural Dynamics Department
Engineering Sciences Center
Sandia National Laboratories
Box 5800
Albuquerque, NM 87185-0380

Abstract

Presto is a Lagrangian, three-dimensional explicit, transient dynamics code for the analysis of solids subjected to large, suddenly applied loads. Presto is designed for problems with large deformations, nonlinear material behavior, and contact. There is a versatile element library incorporating both continuum and structural elements. The code is designed for a parallel computing environment. This document describes the input for the code that gives users access to all of the current functionality in the code. Presto is built in an environment that allows it to be coupled with other engineering analysis codes. The input structure for the code, which uses a concept called scope, reflects the fact that Presto can be used in a coupled environment. This guide describes the scope concept and the input from the outermost to the innermost input scopes. Within a given scope, the descriptions of input commands are grouped based on code functionality. For example, all material input command lines are described in a section of the user's guide for all of the material models in the code.

Document History

Version	Author	Description	Date
1.0	J. Richard Koteras, Arne S. Gullerud	Original version of Presto User's Guide for release 1.0 of Presto.	October 2001
Stable 1.03	J. Richard Koteras, Arne S. Gullerud	Updated version of Presto User's Guide for stable 1.03 of Presto.	April 2002
Stable 1.04	J. Richard Koteras, Arne S. Gullerud	Updated version of Presto User's Guide for stable 1.04 of Presto. Corrected errors in stable 1.03.	July 2002
Stable 1.05	J. Richard Koteras, Arne S. Gullerud	Updated version of Presto User's Guide for stable 1.05 of Presto.	February 2003
2.1	J. Richard Koteras, Arne S. Gullerud	Rough draft to assist users with newest release (2.1) of the code.	October 29, 2004
2.2	J. Richard Koteras, Arne S. Gullerud	Rough draft to assist users with newest release (2.2) of the code.	March 30, 2005
2.3	J. Richard Koteras, Arne S. Gullerud, Nathan K. Crane, Jason D. Hales	Updated version of Presto User's Guide for release 2.3 of Presto.	June 2, 2005
2.3	J. Richard Koteras, Arne S. Gullerud, Nathan K. Crane, Jason D. Hales	Revision of Presto User's Guide for release 2.3 of Presto.	July 28, 2005
2.6	J. Richard Koteras, Arne S. Gullerud, Nathan K. Crane, Jason D. Hales	Updated version of Presto User's Guide for release 2.6 of Presto.	April 2006 and September 20, 2006
2.7	J. Richard Koteras, Arne S. Gullerud, Nathan K. Crane, Jason D. Hales, Rhonda K. Reinert	Updated version of Presto User's Guide for release 2.7 of Presto.	May 2007

Acknowledgments

The authors would like to thank James V. Cox for providing information about the Shared Interface MODels (SIMOD) library and editing the SIMOD section in this document. We also would like to thank Gerald W. Wellman for providing information about the material models, Jakob T. Ostien for providing references for the BCJ model, and Greg Sjaardema for reviewing the chapter on Presto output. Finally, we would like to thank Rhonda Reinert of Technically Write for her careful review and extensive editing of this document and her many valuable suggestions for improving the organization and clarity of this document.

Intentionally Left Blank

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Presto is a three-dimensional transient dynamics code with a versatile element library, nonlinear material models, large deformation capabilities, and contact. It is built on the SIERRA Framework [1]. SIERRA provides a data management framework in a parallel computing environment that allows the addition of capabilities in a modular fashion. Contact capabilities are parallel and scalable; these capabilities are provided by ACME [2].

Presto User's Guide 2.7 (Version 1) provides information about the functionality in Presto and the command structure required to access this functionality in a user input file. This document is divided into chapters based primarily on functionality. For example, command structure related to the use of various element types is grouped in one chapter; descriptions of material models are grouped in another chapter.

One of the key concepts for the command structure in the input file is a concept referred to as *scope*. A detailed explanation of scope is provided in Section 1.2. Most of the command lines in Chapter 2 are related to a certain scope rather than to some particular functionality.

1.1 Document Overview

This document describes how to create an input file for Presto. Highlights of the document contents are as follows:

- Chapter 1 presents the overall structure of the input file, including conventions for the command descriptions, style guidelines for file preparation, and naming conventions for input files that reference the Exodus II database [3]. The chapter also gives an example of the general structure of an input file that employs the concept of scope.

- Chapter 2 explains some of the commands that are general to various applications based on the SIERRA Framework. These commands let you define scopes, functions, and coordinate systems, and they let you set up some of the main time control parameters (begin time, end time, time blocks) for your analysis. (Time control and time step control are discussed in detail in Chapter 3.) Other capabilities documented in this chapter are available for estimating error and for activating and deactivating functionality at different times throughout an analysis.

- Chapter 3 describes how to set the start time, end time, and time blocks for an analysis. This chapter also discusses various options for controlling the critical time step in Presto.

- Chapter 4 describes various material models that can be used in conjunction with the elements in Presto. This chapter also discusses the application of temperature to a mesh, the computation of thermal strains (isotropic and anisotropic), and the deposition of energy for the equation-of-state material models.

- Chapter 5 lists the various elements in Presto and describes how to set up commands to use the various options for the elements. This chapter also includes descriptions of the commands for mass property calculations, element death, and mesh rebalancing. Two ”element-like” capabilities are discussed in Chapter 5—torsional springs and rigid bodies. Although torsional springs and rigid bodies exhibit element-like behavior, they are really implemented as boundary conditions. From a user viewpoint, it is best to discuss the torsional-spring and rigid-body capabilities with elements.

- Chapter 6 documents how to use kinematic boundary conditions, force boundary conditions, initial conditions, and some specialized boundary conditions. Constraints are also discussed in this chapter.

- Chapter 7 discusses how to define interactions of contact surfaces, including the use of the ACME contact package in Presto.

- Chapter 8 details the various options for obtaining output from Presto.

- Chapter 9 provides an overview of the user subroutine functionality.

- Chapter 10 describes how to access a number of surface-physics models using the Shared Interface MODels (SIMOD) library. SIMOD is a third-party library that allows for the addition of surface-physics modeling to various codes in a modular manner.

- Chapter 11 provides a sample input file from an analysis of 16 lead spheres being crushed together inside a steel box.

- Chapter 12 gives all of the permissible Presto input lines in their proper scope.
- The index allows you to find information about the command blocks and command lines in Presto. In general, single-level entries identify the page where the command syntax appears, with discussion following soon thereafter—on the same page or on a subsequent page. Page ranges are not provided in this index. Some entries consist of two or more levels. Such entries are typically based on context, including such information as the command blocks in which a command line appears, the location of the discussion related to a particular command line, and tips on usage. The PDF version of this document contains hyperlinked entries from the page numbers listed in the index to the text in the body of the document.

Note that all references cited within the text of Chapters 1 through 12 are listed at the end of the respective chapters rather than in a separate references chapter.

1.2 Overall Input Structure

Presto is only one of the codes built on the SIERRA Framework. The SIERRA Framework provides the capability to perform multiphysics analyses using a number of codes built on the SIERRA Framework. Input files may be constructed for analyses using only Presto, or input files may be constructed for analyses using Presto and some other analysis code built on the SIERRA Framework. For example, you might run Adagio [4], the quasti-static structural response code, to compute a stress state, and then pass the results of this analysis to Presto as initial conditions for the Presto analysis. For a multiphysics analysis using Presto and Adagio, the time-step control, the mesh-related definitions, and the boundary conditions for both Presto and Adagio will all be in the same input file. Therefore, the input for Presto reflects the fact that a Presto analysis could be part of a multiphysics analysis. (Note that not all codes built on the SIERRA Framework can be coupled. Consult with the authors of this document to learn about the codes that can be coupled with Presto.)

To create files defining multiphysics analyses, the input files use a concept called "scope." Scope is used to group similar commands; a scope can be nested inside another scope. The broadest scope in the input file is the domain scope. The domain scope contains information that can be shared among different physics. Examples of physics information that can be shared are definitions of functions and materials. Thus, in our above example of a Presto/Adagio multiphysics analysis, both Adagio and Presto could reference functions to define such things as time histories for boundary conditions or stress-strain curves. Some of the functions could even be shared by these two applications. Both Presto and Adagio could also share information about materials.

Within the domain scope are two other important scopes: the procedure scope and the region scope. The region is nested inside the procedure, and the procedure is nested inside the domain. For a multiphysics analysis, the domain scope could contain several different procedures and several different regions. For Presto, the procedure scope controls the overall analysis from the start time to the end time. The region scope controls a single time step.

Inside the procedure scope (but outside of the region scope) are commands that set the start time and end time for the analysis.

Inside the region scope for Presto are such things as definitions for boundary conditions and contact. In a multiphysics analysis, there would be more than one region. In our Presto/Adagio example, there would be both a Presto region and an Adagio region, each within its respective procedures. The definitions for boundary conditions and contact and the mesh specification for Presto would appear in the Presto region; the definitions for boundary conditions and contact and the mesh specification for Adagio would appear in the Adagio region.

The input for Presto consists of command blocks and command lines. The command blocks define a scope. These command blocks group command lines or other command blocks that share a similar functionality. A command block will begin with an input line that has the word "begin"; the command block will end with an input line that has the word "end". The domain scope, for example, is defined by a command block that begins with an input line of the form

```
BEGIN SIERRA my_problem .
```

Note that the space and the period following `my_problem` *are not part of the input for the above command line.* As explained in Section 1.3, command lines and command blocks, like equations, are punctuated as if they are part of the text in this document.

The two character strings `BEGIN` and `SIERRA` are the key words for this command block. An input line defining a command block or command line will have one or more key words. The string `my_problem` is a user-specified name for this domain scope. The domain scope is terminated by an input line of the form

```
END SIERRA my_problem ,
```

where `END` and `SIERRA` are the key words to end this command block. The domain scope can also be terminated simply by using

```
END .
```

This abbreviated command line will be discussed in more detail in later chapters. There are similar input lines used to define the procedure and region scopes. Boundary conditions are another example where a scope is defined. A particular instance of a boundary condition for a prescribed displacement boundary condition is defined with a command block. The command block for the boundary condition begins with an input line of the form

```
BEGIN PRESCRIBED DISPLACEMENT
```

and ends with an input line of the form

```
END PRESCRIBED DISPLACEMENT
```

or just simply

```
END .
```

Command lines appear within the command blocks. The command lines typically have the form `keyword = value`, where `value` can be a real, an integer, or a string. In the previous example of the prescribed displacement boundary condition, there would be command lines inside the command block that are used to set various values. For example, the boundary condition might apply to all nodes in node set 10, in which case there would be a command line of the form

```
NODE SET = nodelist_10 .
```

If the prescribed displacement were to be applied along a given component direction, there would be a command line of the form

```
COMPONENT = X ,
```

which would specify that the prescribed displacement would be in the x-direction. Finally, if the displacement magnitude is described by a time history function with the name `cosine_curve`, there would be a command line of the form

```
FUNCTION = cosine_curve .
```

The command block for the boundary condition with the appropriate command lines would appear as follows:

```
BEGIN PRESCRIBED DISPLACEMENT
  NODE SET = nodelist_10
  COMPONENT = X
  FUNCTION = cosine_curve
END PRESCRIBED DISPLACEMENT
```

It is possible to have a command line with the same key words appearing in different scopes. For example, we might have a command line identified by the word `TYPE` in two or more different scopes. The command line would perform different functions based on the scope in which it appeared, and the associated value could be different in the two locations.

The input lines are read by a parser that searches for recognizable key words. If the key words in an input line are not in the list of key words used by Presto to describe command blocks and command lines, the parser will generate an error. A set of key words defining a command line or command block for Presto that is not in the correct scope will also cause a parser error. For example, the key words `STEP INTERVAL` define a valid command line in the scope of the `TIME CONTROL` command block. However, if this command line was to appear in the scope of one of the boundary conditions, it would not be in the proper scope, and the parser would generate an error. Once the parser has an input line with any recognizable key words in the proper scope, a method can be called that will handle the input line.

There is an initial parsing phase that checks only the parser syntax. If the parser encounters a command line it cannot parse within a certain scope, the parser will indicate it cannot recognize the command line and will list the various command lines that can appear within that scope. The initial parsing phase will catch errors such as the one described in the previous paragraph (a command line in the wrong scope). It will also catch misspelled key words. The initial parsing does not catch some other types of errors, however. If you have specified a value on a command line that is out of a specified range for that command line, the initial parsing will not catch this error. If you have some combination of command lines within a command block that is not allowed, the initial parsing will not catch this error. These other errors are caught after the initial parsing phase and are handled one error at a time.

1.3 Conventions for Command Descriptions

The conventions below are used to describe the input commands for Presto. NOTE: In this document, all of the sentences containing input lines are punctuated correctly. For example, the function command line in a sentence such as this one would appear as

```
FUNCTION = <string>function_name .
```

The space after `function_name` indicates that the period following this space is not a part of the command line but is the correct punctuation in the text. The above command line in the input file would NOT have a period.

A number of the individual command lines discussed in the text appear on several text lines. In the text of this document, the continuation symbols that are used to continue lines in an actual input file (`\#` and `\$`, Section 1.4.2) are not used for those instances where the description of the command line appears on several text lines. The description of command lines will clearly indicate all of the key words, delimiters, and values that constitute a complete command line. As an example, the `DEFINE POINT` command line (Section 2.1.6) is presented in the text as

```
DEFINE POINT <string>point_name WITH COORDINATES
  <real>value_1 <real>value_2 <real>value_3 .
```

If the `DEFINE POINT` command line were used as a command line in an input file and spread over two input lines, it would appear, with actual values, as

```
DEFINE POINT center WITH COORDINATES \#
10.0 144.0 296.0 ,
```

where the `\#` symbol implies the first line is continued onto the second line.

1.3.1 Key Words

The key word or key words for a command are shown in uppercase letters. For actual input, you can use all uppercase letters for the key words, all lowercase letters for the key words, or some combination of uppercase and lowercase letters for the key words.

1.3.2 User-Specified Input

The input that you supply is typically shown in lowercase letters. (Occasionally, uppercase letters may be used for user input for purposes of clarity or in examples.) The user-supplied input may be a real number, an integer, a string, or a string list. For the command descriptions, a type appears before the user input. The type (real, integer, string, string list) description is enclosed by angle brackets, <>, and precedes the user-supplied input. For example,

```
<real>value
```

indicates that the quantity `value` is a real number. For the description of an input command, you would see

```
FUNCTION = <string>function_name .
```

Your input would be

```
FUNCTION = my_name
```

if you have specified a function name called `my_name`.

Valid user input consists of the following:

`<integer>`	Integer data is a single integer number.
`<real>`	Real data is a single real number. It may be formatted with the usual conventions, such as `1234.56` or `1.23456e+03`.
`<string>`	String data is a single string.
`<string list>`	A string list consists of multiple strings separated by white space, a comma, a tab, or white space combined with a comma or a tab.

1.3.3 Optional Input

Anything in an input line that is enclosed by square brackets, [ ], represents optional input within the line. Note, however, that this convention is not used to identify optional input lines. Any command line that is optional (in its entirety) will be described as such within the text.

1.3.4 Default Values

A value enclosed by parentheses, (), appearing after the user input denotes the default value. For example,

```
SCALE FACTOR = <real>scale_factor(1.0)
```

implies the default value for `scale_factor` is 1.0. Any value you specify will overwrite the default.

For your actual input file, you may simply omit a command line if you want to use the default value associated with the command line. For example, there is a `TIME`

`STEP SCALE FACTOR` command line used to set one of the time control parameters; the parameter for this command line has a default value of 1.0. If you want to use the default value of 1.0 for this parameter, you do not have to include the `TIME STEP SCALE FACTOR` command line in the `TIME CONTROL` command block.

1.3.5 Multiple Options for Values

Quantities separated by the | symbol indicate that one and only one of the possible choices must be selected. For example,

```
EXPANSION RADIUS = <string>SPHERICAL|CYLINDRICAL
```

implies that expansion radius must be defined as `SPHERICAL` or `CYLINDRICAL`. One of the values must appear. This convention also applies to some of the command options within a begin/end block. For example,

```
SURFACE = <string>surface_name |
  NODE SET = <string>nodelist_name
```

in a command block specifies that either a surface or a node set must be specified.

Quantities separated by the `/` symbol can appear in any combination, but any one quantity in the sequence can appear only once. For example,

```
COMPONENTS = <string>X/Y/Z
```

implies that components can equal any combination of X, Y, and Z. Any value (X or Y or Z) can appear at most once, and at least one value of X, Y, or Z must appear. Some examples of valid expressions in this case are

```
COMPONENTS = Z  ,
```

```
COMPONENTS = Z X  ,
```

```
COMPONENTS = Y X Z  ,
```

and

```
COMPONENTS = Z Y X  .
```

An example of an invalid expression would be

```
COMPONENTS = Y Y Z  .
```

1.3.6 Set of Command Lines

In some of the command blocks, it may be possible to select from a set of command lines to activate some functionality within the command block. In the boundary conditions, for example, a boundary condition may be applied to a group of nodes.

This group of nodes can be defined by some collection of command lines that are Boolean operations to add or delete nodes to the group. The command lines that are the Boolean operations to define the group of nodes are described in detail in the introduction to the chapter on boundary conditions, i.e., in Section 6.1. In the description of the command blocks for the boundary conditions, we denote this set of command lines by enclosing a name for the set of command lines in curly braces, {}. Therefore, the description of a command block for a kinematic boundary condition may have a line of the form

```
{node set commands}
```

to indicate that the user can insert some arbitrary combination of command lines from the set of command lines defined as `node set commands`.

We will use {} to enclose some named set of command lines or some description of a set of command lines.

1.4 Style Guidelines

This section gives information that will affect the overall organization and appearance of your input file. It also contains recommendations that will help you construct input files that are readable and easy to proof.

1.4.1 Comments

A comment is anything between the `#` symbol or the `$` symbol and the end-of-line. If the first nonblank character in a line is a `#` or `$`, the entire line is a comment line. You can also place a `#` or `$` (preceded by a blank space) after the last character in an input line used to define a command block or command line.

1.4.2 Continuation Lines

An input line can be continued by placing a \# pair of characters (or `\$`) at the end of the line. The following line is then taken to be a continuation of the preceding line that was terminated by the \# or `\$`. Note that everything after the line-continuation pair of characters is discarded, including the end-of-line.

1.4.3 Case

Almost all of the character strings in the input lines are case insensitive. For example, the `BEGIN SIERRA` key words could appear as

```
BEGIN SIERRA
```

or

```
begin sierra
```

or

```
Begin Sierra .
```

You could specify a `SIERRA` command block with

```
BEGIN SIERRA BEAM
```

and terminate the command block with

```
END SIERRA beam .
```

Case is important only for file name specifications. If you have defined a restart file with uppercase and lowercase letters and want to use this file for a restart, the file name you use to request this restart file must exactly match the original definition you chose.

1.4.4 Commas and Tabs

Commas and tabs in input lines are ignored.

1.4.5 Blank Spaces

We highly recommend that everything be separated by blank spaces. For example, a command line of the form

```
node set = nodelist_10
```

is recommended over

```
node set= nodelist_10
```

or

```
node set =nodelist_10
```
.

Both of the above two lines are correct, but it is easier to check the first form (the equal sign surrounded by blank space) in a large input file.

The parser will accept the line

```
BEGIN SIERRABEAM
```
,

but it is harder to check this line for the correct spelling of the key words and the intended domain name than the line

```
BEGIN SIERRA BEAM
```
.

It is possible to introduce hard-to-detect errors because of the way in which the blank spaces are handled by the command parser. Suppose you type

```
begin definition for functions my_func
```

rather than the correct form, which is

```
begin definition for function my_func
```
.

For the incorrect form of this command line (in which `functions` is used rather than `function`), the parser will generate a string name of

```
s my_func
```

for the function name rather than the expected name of

```
my_func
```
.

If you attempt to use a function named `my_func`, the parser will generate an error because the list of function names will include `s my_func` but not `my_func`.

1.4.6 General Format of the Command Lines

In general, command lines have the form

```
keyword = value .
```

This pattern is not always followed, but it describes the vast majority of the command lines.

1.4.7 Delimiters

The delimiter used throughout this document is "`=`" (the equal sign). Typically, but not always, the `=` sign separates key words from input values in a command line. For example, in the command line

```
COMPONENTS = X ,
```

the key word `COMPONENTS` is separated from its value, a string in this case, by the `=` sign. Some command lines do allow for other delimiters. The use of these alternate delimiters is not consistent, however, throughout the various command lines. (This lack of consistency has the potential for introducing errors in this document as well as in your input.) The `=` sign provides a strong visual cue for separating key words from values. By using the `=` sign as a delimiter, it is much easier to proof your input file. It also makes it easier to do "cut and paste" operations. If you accidently delete an `=` sign, it is much easier to detect than accidently removing part of one of the other delimiters that could be used.

1.4.8 Order of Commands

There are no requirements for ordering the commands. Both the input sequence

```
BEGIN PRESCRIBED DISPLACEMENT
  NODE SET = nodelist_10
  COMPONENT = X
  FUNCTION = cosine_curve
END PRESCRIBED DISPLACEMENT
```

and the input sequence

```
BEGIN PRESCRIBED DISPLACEMENT
  FUNCTION = cosine_curve
  COMPONENT = X
  NODE SET = nodelist_10
END PRESCRIBED DISPLACEMENT
```

are valid, and they produce the same result. Remember, however, that command lines and command blocks must appear in the proper scope.

1.4.9 Abbreviated END Specifications

It is possible to terminate a command block without including the key word or key words that identify the block. For example, you could define a specific instance of the prescribed displacement boundary condition with

```
BEGIN PRESCRIBED DISPLACEMENT
```

and terminate it simply with

```
END
```

as opposed to

```
END PRESCRIBED DISPLACEMENT .
```

Both the short termination (`END` only) and the long termination (`END` followed by identification, or name, of the command block) are valid. It is recommended that the long termination be used for any command block that becomes large. For example, the `RESULTS OUTPUT` command block described in later chapters can become fairly lengthy, so this is probably a good place to use the long termination. For most boundary conditions, the command block will typically consist of five lines. In such cases, the short termination can be used. Using the long termination for the larger command blocks will make it easier to proof your input files. If you use the long termination, the text following the `END` key word must exactly match the text following the `BEGIN` key word. You could not, for example, have `BEGIN PRESCRIBED DISPLACEMENT` paired with an `END PRESCRIBED DISPL` to define the beginning and ending of a command block.

1.4.10 Indentation

When constructing an input file, it is useful to indent a scope that is nested inside another scope. Command lines within a command block should also be indented in relation to the lines defining the command block. This will make it easier to construct the input file with everything in the correct scope and with all of the command blocks in the correct structure.

1.5 Naming Conventions Associated with the Exodus II Database

When the mesh file has an Exodus II format, there are three basic conventions that apply to user input for various command lines. First, for a mesh file with the Exodus II format, the Exodus II side set is referenced as a surface. In SIERRA, a surface consists of element faces plus all of the nodes and edges associated with these faces. A surface definition can be used not only to select a group of faces but also to select a group of edges or a group of nodes that are associated with those faces. In the case of boundary conditions, a surface definition can be used not only to apply boundary conditions that typically use surface specifications (pressure) but also to apply boundary conditions for what are referred to as nodal boundary conditions (fixed displacement components). For nodal boundary conditions that use the surface specification, all of the nodes associated with the faces on a specific surface will have this boundary condition applied to them. The specification for a surface identifier in the following chapters is `surface_name`. It typically has the form `surface_integerid`, where `integerid` is the integer identifier for the surface. If the side set number is 125, the value of `surface_name` would be `surface_125`. It is also possible to generate an alias for the side set and use this for `surface_name`. If `surface_125` is aliased to `outer_skin`, then `surface_name` becomes `outer_skin` in the actual input line.

Second, for a mesh file with the Exodus II format, the Exodus II node set is still referenced as a node set. A node set can be used only for cases where a group of nodes needs to be defined. The specification for a node set identifier in the following chapters is `nodelist_name`. It typically has the form `nodelist_integerid`, where `integerid` is the integer identifier for the node set. If the node set number is 225, the value of `nodelist_name` would be `nodelist_225`. It is also possible to generate an alias for the node set and use this for `nodelist_name`. If `nodelist_225` is aliased to `inner_skin`, then `nodelist_name` becomes `inner_skin` in the actual input line.

Third, an element block is referenced as a block. The specification for an element block identifier in the following chapters is `block_name`. It typically has the form `block_integerid`, where `integerid` is the integer identifier for the block. If the element block number is 300, the value of `block_name` would be `block_300`. It is also possible to generate an alias for the block and use this for `block_name`. If `block_300` is aliased to `big_chunk`, then `block_name` becomes `big_chunk` in the actual input line.

A group of elements can also be used to select other mesh entities. In SIERRA, a block consists of elements plus all of the faces, edges, and nodes associated with the elements. The block and surface concepts are similar in that both have associated derived quantities. Chapter 6 and Chapter 7 show how this concept of derived quantities is used in the input command structure.

1.6 Major Scope Definitions for a Presto Input File

The typical Presto input file will have the structure shown below. The major scopes—domain, procedure, and region—are delineated with input lines for command blocks. Comment lines are included that indicate some of the key scopes that will appear within the major scopes. Note the indentation used for this example.

```
BEGIN SIERRA <string>some_name
  #
  # All command blocks and command lines in the domain
  # scope appear here. The PROCEDURE PRESTO command
  # block is the beginning of the next scope.
  #
  # function definitions
  # material descriptions
  # description of mesh file
  #
  BEGIN PROCEDURE PRESTO <string>procedure_name
    #
    # time step control
    #
    BEGIN REGION PRESTO <string>region_name
      #
      # All command blocks and command lines in the
      # region scope appear here
      #
      # specification for output of result
      # specification for restart
      # boundary conditions
      # definition of contact
      #
    END [REGION PRESTO <string>region_name]
  END [PROCEDURE PRESTO <string>procedure_name]
END [SIERRA <string>some_name]
```

1.7 Input/Output Files for Presto

The primary user input to Presto is the input file introduced in this chapter. Throughout this document, we explain how to construct a valid input file. It is important to be aware, however, that Presto also processes a number of other types of input files and produces a variety of output files. These additional files are also discussed in this document where applicable. Figure 1.1 presents a simple schematic diagram of the various input and output files in Presto.

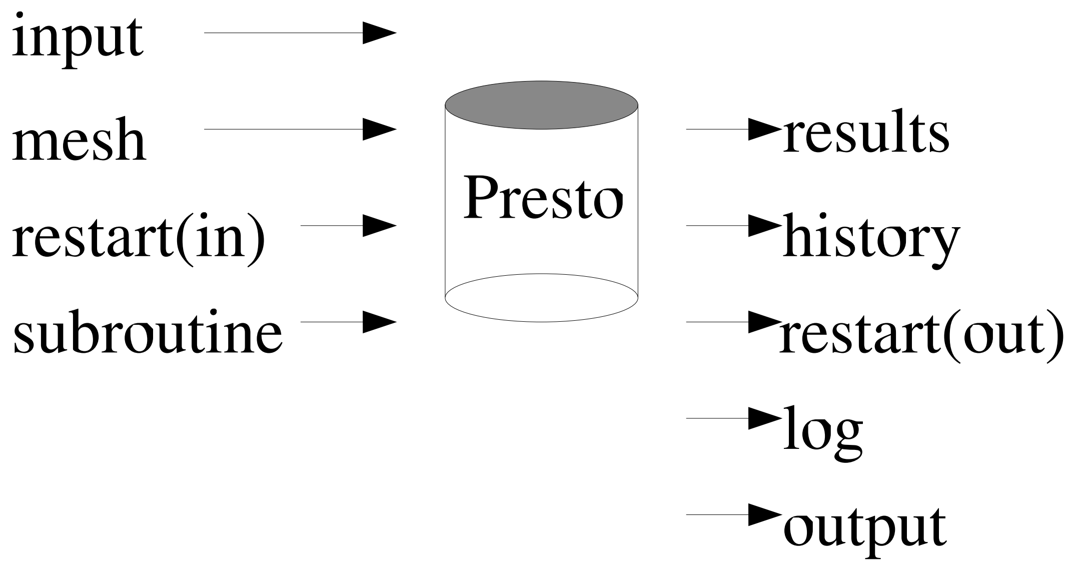


Figure 1.1: Input/output files for Presto.

As shown in Figure 1.1, Presto uses the input file, mesh files, restart files, and user subroutine files. The input file, which is required, is a set of valid Presto command lines. Another required input is a mesh file, which provides a description of the finite element mesh for the object being analyzed. Restart and user subroutine files are optional inputs. The restart functionality lets you break an analysis from the start time to the termination time into a sequence of runs. The files generated by the restart functionality contain a complete state description for a problem at various analysis times, which we will refer to as restart times. You can restart Presto at any of these restart times because the complete state description is known (see Chapter 8). The user subroutine files let you build and incorporate specialized functionality into Presto (Chapter 9).

As also shown in Figure 1.1, Presto can generate a number of files. These include results files, history files, restart files, a log file, and an output file. Typically, only the log file and the output file are produced automatically. Generation of the other types of files is based on user settings in the input file for the particular kinds of output desired. Results files provide the values of global variables, element variables, and node variables at specified times (see Chapter 8). History files will also provide values of global variables, element variables, and node variables at specified times (see Chapter 8). History files are set up to provide a specific value at a specific node, for example, whereas results files provide a nodal value for large subsets of nodes or, more typically, all nodes. History files provide a much more limited set of information than results files. As noted above, restart files can be generated at various analysis times. The log file contains a variety of information such as the Presto version number, a listing of the input file, initialization information, some model information (mass, critical time steps for element blocks, etc.), and information at various time steps. At every nth step, where n is user selected, the log file gives the current analysis time; the current time step; the kinetic, internal, and external energies; the error in the energy; and computing time information. You can monitor step information in the log file to gain information about how your analysis is progressing. The output file contains error information.

1.8 References

1. Edwards, H. C., and J. R. Stewart. "SIERRA: A Software Environment for Developing Complex Multi-Physics Applications." In *First MIT Conference on Computational Fluid and Solid Mechanics*, edited by K. J. Bathe, 1147–1150. Amsterdam: Elsevier, 2001.

2. Brown, K. H., R. M. Summers, M. W. Glass, A. S. Gullerud, M. W. Heinstein, and R. E. Jones. *ACME: Algorithms for Contact in a Multiphysics Environment*, API Version, 1.0. Albuquerque, NM: Sandia National Laboratories, October 2001.

3. Schoof, L. A., and V. R. Yarberry. *EXODUS II: A Finite Element Data Model*, SAND92-2137. Albuquerque, NM: Sandia National Laboratories, September 1994.

4. Mitchell, J. A., A. S. Gullerud, W. M. Scherzinger, J. R. Koteras, and V. L. Porter. "ADAGIO: Non-Linear Quasi-Static Structural Response Using the SIERRA Framework." In *First MIT Conference on Computational Fluid and Solid Mechanics*, edited by K. J. Bathe, 361–364. Amsterdam: Elsevier, 2001.

Chapter 2

General Commands

The commands described in this section appear in the domain or procedure scope or control some general functionality in Presto.

2.1 Domain Scope

These commands are used to set up some of the fundamentals of the Presto input. The commands are physics independent, or at least can be shared between physics. The commands lie in the domain scope, not in the procedure or region scope.

2.1.1 SIERRA Command Block

```
BEGIN SIERRA <string>name
  #
  # All other command blocks and command lines
  # appear within the domain scope defined by
  # begin/end sierra.
  #
END [SIERRA <string>name]
```

All input commands for Presto must occur within a `SIERRA` command block. The syntax for beginning the command block is

```
BEGIN SIERRA <string>name
```

and for terminating the command block is

`END [SIERRA <string>name]` ,

where name is a name for the `SIERRA` command block. All other commands for the analysis must be within this command block structure. The name for the `SIERRA` command block is often a descriptive name that identifies the analysis. The name is not currently used anywhere else in the file and is completely arbitrary.

2.1.2 Title

```
    TITLE <string list>title
```

To permit a fuller description of the analysis, the Presto input has a `TITLE` command line for the analysis, where `title` is a text description of the analysis. The title is transferred to the results file.

2.1.3 Restart Control

The restart capability in Presto allows a user to run an analysis up to a certain time, stop the analysis at that time, and then restart the analysis at this stop time. Restart can be used to break a long-running analysis into several smaller runs so that the user can examine intermediate results before proceeding with the next step. Restart can also be used in case of abnormal termination. If a restart file has been written at various intervals throughout the analysis up to the point where the abnormal termination has occurred, you can pick some restart time before the abnormal termination and restart the problem from there. Thus, users do not have to go back to the beginning of the analysis, but can continue the analysis at some time well into the analysis. With the restart capability, you will generate a sequence of restart runs. Each run can have its own set of restart, results, and history files.

When using the restart capability, you can reset a number of the parameters in the input file. Not all parameters, however, can be reset. Users should exercise care in resetting parameters in the input file for a restart. You will likely want to change parameters if you have encountered an abnormal termination. You may want to change certain parameters, hourglass control for example, to see whether you can prevent the abnormal termination and continue the analysis past the abnormal termination time you had previously encountered.

The use of the restart capability involves commands in **both the domain scope and the region scope**. One of two restart command lines, `RESTART` or `RESTART TIME`, appears in the domain scope. A command block in the region scope, the `RESTART DATA` command block, specifies restart file names and the frequency at which the restart files will be written. The `RESTART DATA` command block is described in Section 8.3. This section gives a brief discussion of the command lines that appear in the domain scope. For a full discussion of all the command lines used for restart, consult with Chapter 8. The use of some of the command lines in the `RESTART DATA` command block depends on the command line, either `RESTART` or `RESTART TIME`, you select in the domain scope.

If you specify a time from a specific restart file for the restart, you will use the `RESTART TIME` command line described in Section 2.1.3.1. If you select the automatic restart option, you will use the `RESTART` command line described in Section 2.1.3.2. The command lines for both of these methods are in the **domain scope**. All other commands for restart are in the **region scope** in the `RESTART DATA` command block.

For restarts specified with a restart time from a specific restart file, you will have to be concerned about overwriting information in existing files. The issue of overwriting information is discussed in Chapter 8. In general, you will want to have a restart file (or files in the case of parallel runs) for each run in a sequence of runs you create with the restart option. You will want to preserve all restart files you have written prior to any given run in a sequence of restart runs. The easiest way to preserve prior restart information is with the use of the `RESTART` command line. How you preserve previous restart information is discussed in detail in Chapter 8.

The amount of data written at a restart time is quite large. The restart data written at a given time is a complete description of the state for the problem at that time. The restart data includes not only information such as displacement, velocity, and acceleration, but also information such as element stresses and all the state variables for the material model associated with each element.

2.1.3.1 Restart Time

```
RESTART TIME = <real>restart_time
```

The `RESTART TIME` command line is used to specify a time from a specific restart file for the restart run. This restart option will pick the restart time on the restart file that is closest to the user-specified time on the `RESTART TIME` command line. If the user specifies a restart time greater than the last time written to a restart file, then the last time written to the restart file is picked as the restart time. Use of this command line can result in previous restart information being overwritten. To prevent the overwriting of existing restart files, you can specify both an input restart file and an output restart file (and rename the results and history files) for the various restarts. The use of the `RESTART TIME` command line requires the user to be more active in the management of the file names to prevent the overwriting of restart, results, and history files. The automatic restart feature (i.e., the `RESTART` command line in Section 2.1.3.2) prevents the overwriting of restart, results, and history files. Consult with Section 8.3 for a full discussion of implementing the restart capability.

2.1.3.2 Automatic Restart

```
RESTART = AUTOMATIC
```

The RESTART command line automatically selects for restart the last restart time written to the last restart file. The automatic restart feature lets the user restart runs with minimal changes to the input file. The only quantity that must be changed to move from one restart to another is the termination time. The RESTART command line manages the restart files so as not to write over any previous restart files. It also manages the results and history files so as not to write over any previous results or history files. Consult with Section 8.3 for a full discussion of implementing the restart capability.

2.1.4 User Subroutine Identification

```
USER SUBROUTINE FILE = <string>file_name
```

This command line is a part of a set of commands that are used to implement the user subroutine functionality. The string file_name identifies the name of the file that contains the FORTRAN code of one or more user subroutines.

To understand how this command line is used, see Chapter 9.

2.1.5 Functions

```
BEGIN DEFINITION FOR FUNCTION <string>function_name
  TYPE = <string>CONSTANT|PIECEWISE LINEAR|
    ANALYTIC
  ABSCISSA = <string>abscissa_label
  ORDINATE = <string>ordinate_label
  BEGIN VALUES
    <real>value_1   [<real>value_2
    <real>value_3    <real>value_4
    ...              <real>value_n]
  END [VALUES]
  EVALUATE EXPRESSION = <string>analytic_expression1;
    analytic_expression2;...
END [DEFINITION FOR FUNCTION <string>function_name]
```

A number of Presto features are driven by a user-defined description of the dependence of one variable on another. For instance, the prescribed displacement boundary condition requires the definition of a time-versus-displacement relation, and the thermal strain computations require the definition of a thermal-strain-versus-temperature relation. SIERRA provides a general method of defining these relations as functions using the `DEFINITION FOR FUNCTION` command block, as shown above.

There is no limit to the number of functions that can be defined. All function definitions must appear within the domain scope.

A description of the various parts of the `DEFINITION FOR FUNCTION` command block follows:

- The string `function_name` is a user-selected name for the function that is unique to the function definitions within the input file. This name is used to refer to this function in other locations in the input file.
- The `TYPE` command line has three options to define the type of function. The string value can be `CONSTANT`, `PIECEWISE LINEAR`, or `ANALYTIC`.
- The `ABSCISSA` command line provides a descriptive label for the independent variable (x-axis) with the string `abscissa_label`. This command line is optional.
- The `ORDINATE command` line provides a descriptive label for the dependent variable (y-axis) with the string `ordinate_label`. This command line is optional.
- The `VALUES` command block consists of the real values `value_1` through `value_n`, which describe the function. This command block must be used if the value on the `TYPE` command line is `CONSTANT` or `PIECEWISE LINEAR`. For a `CONSTANT` function, only one value is needed. For a `PIECEWISE LINEAR` function, the values are (x, y) pairs of data that describe the function. The values are nested inside the `VALUES` command block.

  In the case of a `PIECEWISE LINEAR` function, for any abscissa value passed to the function that is greater than the last abscissa value in the `VALUES` command block, the last ordinate value is used for the function value. For example, suppose we have a piecewise linear function that indicates a function `my_func` describes a time history for a pressure load where the pressure increases from 0 to 50,000 psi from time 0.0 sec to time 1.0×10^{-3} sec. The last time specified in the function is 1.0×10^{-3}. Now, suppose our final analysis time is 2.0×10^{-3} sec. Then, from the time 1.0×10^{-3} to the time 2.0×10^{-3}, the value for this function (`my_func`) will be 50,000 psi.
- The `EVALUATE EXPRESSION` command line consists of one or more user-supplied algebraic expressions. This command line must be used if the value on the `TYPE`

command line is `ANALYTIC`. See the rules and options for composing algebraic expressions discussed below.

Importantly, a `DEFINITION FOR FUNCTION` command block cannot contain both a `VALUES` command block and an `EVALUATE EXPRESSION` command line.

Rules and options for composing algebraic expressions. If you choose to use the `EVALUATE EXPRESSION` command line, you will need to write the algebraic expressions. The algebraic expressions are written using a C-like format. Each algebraic expression is terminated by a semicolon. The entire set of algebraic expressions, whether a single expression or several, is enclosed in a single set of double quotes.

An expression is evaluated with `x` as the independent variable. We first provide several simple examples and then list the options available in the algebraic expressions.

Example: Return `sin(x)` as the value of the function.

```
begin definition for function fred
  type is analytic
  evaluate expression is ‘‘sin(x);’’
end definition for function fred
```

Example: In this example, the commented out table is equivalent to the evaluated expression.

```
begin definition for function pressure
  type is analytic
  evaluate expression is ‘‘x <= 0.0 ? 0.0 : (x < 0.5 ? x*200.0
    : 100.0);’’
    #     begin values
    #       0.0      0.0
    #       0.5    100.0
    #       1.0    100.0
    #     end values
end definition for function pressure
```

The following functionality is currently implemented for the expressions:

Operators

```
+ - * / == != > < >= <= ! & | && || ? :
```

Parentheses

```
()
```

Math functions

```
abs(x), absolute value of x
mod(x, y), modulus of x|y
ipart(x), integer part of x
fpart(x), fractional part of x
```

Power functions

```
pow(x, y), x to the y power
pow10(x), x to the 10 power
sqrt(x), square root of x
```

Trigonometric functions

```
acos(x), arccosine of x
asin(x), arcsine of x
atan(x), arctangent of x
atan2(y, x), arctangent of y/x, signs of x and y
  determine quadrant (see atan2 man page)
cos(x), cosine of x
cosh(x), hyperbolic cosine of x
sin(x), sine of x
sinh(x), hyperbolic sine of x
tan(x), tangent of x
tanh(x), hyperbolic tangent of x
```

Logarithm functions

```
log(x), natural logarithm of x
ln(x), natural logarithm of x
log10(x), the base 10 logarithm of x
exp(x), e to the x power
```

Rounding functions

```
ceil(x), smallest integral value not less than x
floor(x), largest integral value not greater than x
```

Random functions

```
rand(), random number between 0.0 and 1.0, not including 1.0
randomize(), random number between 0.0 and 1.0, not
  including 1.0
srand(x), seeds the random number generator
```

Conversion functions

```
deg(x), converts radians to degrees
rad(x), converts degrees to radians
recttopolr(x, y), magnitude of vector x, y
recttopola(x, y), angle of vector x, y
poltorectx(r, theta), x coordinate of angle theta at
  distance r
poltorecty(r, theta), y coordinate of angle theta at
  distance r
```

Constants. There are two predefined constants that may be used in an expression. These two constants are `e` and `pi`.

e = e = 2.7182818284...
pi = π = 3.1415926535...

2.1.6 Axes, Directions, and Points

```
DEFINE POINT <string>point_name WITH COORDINATES
  <real>value_1 <real>value_2 <real>value_3
DEFINE DIRECTION <string>direction_name WITH VECTOR
  <real>value_1 <real>value_2 <real>value_3
DEFINE AXIS <string>axis_name WITH POINT
  <string>point_1 POINT <string>point_2
DEFINE AXIS <string>axis_name WITH POINT
  <string>point DIRECTION <string>direction
```

A number of Presto features require the definition of geometric entities. For instance, the prescribed displacement boundary condition requires a direction definition, and the cylindrical velocity initial condition requires an axis definition. Currently, Presto input permits the definition of points, directions, and axes. Definition of these geometric entities occurs in the domain scope.

The `DEFINE POINT` command line is used to define a point:

```
DEFINE POINT <string>point_name WITH COORDINATES
  <real>value_1 <real>value_2 <real>value_3
```

where

- The string `point_name` is a name for this point. This name must be unique to all other points defined in the input file.

- The real values `value_1`, `value_2`, and `value_3` are the x, y, and z coordinates of the point.

The `DEFINE DIRECTION` command line is used to define a direction:

```
DEFINE DIRECTION <string>direction_name WITH VECTOR
  <real>value_1 <real>value_2 <real>value_3
```

where

- The string `direction_name` is a name for this direction. This name must be unique to all other directions defined in the input file.
- The real values `value_1`, `value_2`, and `value_3` are the x, y, and z magnitudes of the direction vector.

There are two command lines that can be used to define an axis. The first `DEFINE AXIS` command line uses two points:

```
DEFINE AXIS <string>axis_name WITH POINT
  <string>point_1 POINT <string>point_2
```

where

- The string `axis_name` is a name for this axis. This name must be unique to all other axes defined in the input file.
- The strings `point_1` and `point_2` are the names for two points defined in the input file via a `DEFINE POINT` command line.

The second `DEFINE AXIS` command line uses a point and a direction:

```
DEFINE AXIS <string>axis_name WITH POINT
  <string>point DIRECTION <string>direction
```

where

- The string `axis_name` is a name for this axis. This name must be unique to all other axes defined in the input file.
- The string `point` is the name of a point defined in the input file via a `DEFINE POINT` command line.
- The string `direction` is the name of a direction defined in the input file via a `DEFINE DIRECTION` command line.

2.1.7 Orientation

```
BEGIN ORIENTATION <string>orientation_name
  SYSTEM = <string>RECTANGULAR|Z RECTANGULAR|CYLINDRICAL|
    SPHERICAL(RECTANGULAR)
  #
  POINT A = <real>global_ax <real>global_ay <real>global_az
  POINT B = <real>global_bx <real>global_by <real>global_bz
  #
  ROTATION ABOUT <integer> 1|2|3(1) = <real>theta(0.0)
END [ORIENTATION <string>orientation_name]
```

The `ORIENTATION` command block is currently used in Presto to define a local coordinate system for output of shell stresses. In the future, the `ORIENTATION` command block will be used with other functionality in Presto.

A local coordinate system is defined at the particular location at which it is required. For example, suppose we want to define a local coordinate system for a shell element. This local coordinate system will be used for output of stresses in the element. For shell elements, the centroid of the element is where we want to define the local coordinates for output of stresses. When we use orientation with a shell element, the centroid of the shell element becomes the particular location at which we want to define a local coordinate system. (When we associate an orientation with a block of shell elements, the orientation will generate a local coordinate system for each element in the block.)

The `SYSTEM` command line gives you several options for constructing a local coordinate system. The options on this command line are `RECTANGULAR`, `Z RECTANGULAR`, `CYLINDRICAL`, and `SPHERICAL`. The `SYSTEM` command line is optional. If you do not include a `SYSTEM` command line in the `ORIENTATION` command block, the default system is the `RECTANGULAR` system.

The `ORIENTATION` command actually generates two local coordinate systems. The first local system constructed at a particular location will always be a Cartesian system designated as $X'Y'Z'$. How this system is constructed depends on the choice for the `SYSTEM` option. Regardless of what system option you choose, the command lines `POINT A` and `POINT B` are required. The details of constructing a local coordinate system for each of the different `SYSTEM` options is described below:

- `RECTANGULAR` Option: The command line `POINT A` defines a point a that lies on the X'-axis. The command line `POINT B` defines a point b that lies in the $X'Y'$-plane. Let the coordinates of a define a vector $\vec{A}$ and the coordinates of b define a vector $\vec{B}$. The normalized value of $\vec{A}$, $\vec{A}/\parallel\vec{A}\parallel$, defines a unit vector along the X'-axis, which we denote as $\vec{X'}$. The normalized cross-product of $\vec{A} \times \vec{B}$ is a unit vector defining the Z'-axis, which we denote as $\vec{Z}'$. We can obtain a unit vector along the Y'-axis, $\vec{Y}'$, from a cross-product of $\vec{Z}'$ and $\vec{X}'$. The three unit vectors $\vec{X}'$, $\vec{Y}'$, and $\vec{Z}'$ give us our local coordinate system $X'Y'Z'$.

- `Z RECTANGULAR` Option: The command line `POINT A` defines a point a that lies on the Z'-axis. The command line `POINT B` defines a point b that lies in the $X'Z'$-plane. Let the coordinates of a define a vector $\vec{A}$ and the coordinates of b define a vector $\vec{B}$. The normalized value of $\vec{A}$, $\vec{A}/ \parallel \vec{A} \parallel$, defines a unit vector along the Z'-axis, which we denote as $\vec{Z}'$. The normalized cross-product of $\vec{A} \times \vec{B}$ is a unit vector defining the Y'-axis, which we denote as $\vec{Y}'$. We can obtain a unit vector along the X'-axis, $\vec{X}'$, from a cross-product of $\vec{Y}'$ and $\vec{Z}'$. The three unit vectors $\vec{X}'$, $\vec{Y}'$, and $\vec{Z}'$ give us our local coordinate system $X'Y'Z'$.

- `CYLINDRICAL` Option: The point a defined by the command line `POINT A` and the point b defined by the command line `POINT B` define a cylindrical axis. The local coordinate system always has the Z'-axis parallel to this cylindrical axis and in the direction from a to b. The vector $\vec{Z}'$ is a unit vector defining the Z'-axis. The X'-axis lies along a line that is normal to the cylindrical axis and passes through the origin of our local coordinate system. (Example: If we are defining a local system for shell stress output, the origin of our local system is the centroid of the element.) The vector $\vec{X}'$ is a unit vector defining the X'-axis. We can obtain the Y'-axis from the cross-product of the $\vec{Z}'$ and $\vec{X}'$ vectors. The three unit vectors $\vec{X}'$, $\vec{Y}'$, and $\vec{Z}'$ give us our local coordinate system $X'Y'Z'$.

- `SPHERICAL` Option: The point a, from the `POINT A` command line, defines the center of a sphere. The point b, from the `POINT B` command line, defines a polar axis for the sphere. The X'-axis lies along a line passing through the origin of the sphere, point a, and the origin of our local coordinate system. (Example: If we are defining a local system for shell stress output, the origin of our local system is the centroid of the element.) The vector $\vec{X}'$ is a unit vector defining the X'-axis. A cross product of the polar axis for the sphere and the vector $\vec{X}'$ gives the $\vec{Y}'$ vector. The vector $\vec{Y}'$ is a unit vector defining the Y'-axis. We can obtain the Z'-axis from the cross-product of the $\vec{X}'$ and $\vec{Y}'$ vectors. The three unit vectors $\vec{X}'$, $\vec{Y}'$, and $\vec{Z}'$ give us our local coordinate system $X'Y'Z'$.

The second local coordinate system constructed at a particular location is defined by use of the `ROTATION` command line. This second local coordinate system is always a Cartesian system that is designated as $X''Y''Z''$. The `ROTATION` command line has the form:

```
    ROTATION ABOUT 1|2|3(1) = <real>theta(0.0) .
```

The second local coordinate system, $X''Y''Z''$, is obtained by specifying some rotation, the `theta` parameter, about an axis, which is specified with an integer 1, 2, or 3. The parameter `theta` has units of degrees. The manner in which the $X''Y''Z''$ is generated is as follows:

- Rotation about axis 1: If the `ROTATION` command line specifies a rotation about the 1 axis, then the $X''Y''Z''$ coordinate system is obtained by a transformation that rotates the $X'Y'Z'$ coordinate system by `theta` degrees about the X'-axis. The local origin for $X''Y''Z''$ is the same as that for $X'Y'Z'$.
- Rotation about axis 2: If the `ROTATE` command line specifies a rotation about the 2 axis, then the $X''Y''Z''$ coordinate system is obtained by a transformation that rotates the $X'Y'Z'$ coordinate system by `theta` degrees about the Y'-axis. The local origin for $X''Y''Z''$ is the same as that for $X'Y'Z'$.
- Rotation about axis 3: If the `ROTATE` command line specifies a rotation about the 3 axis, then the $X''Y''Z''$ coordinate system is obtained by a transformation that rotates the $X'Y'Z'$ coordinate system by `theta` degrees about the Z'-axis. The local origin for $X''Y''Z''$ is the same as that for $X'Y'Z'$.

If no `ROTATION` command line is used, the $X''Y''Z''$ coordinate system is generated with rotation about the 1-axis (X') of zero degree (`theta` set to zero `0.0`).

2.2 Presto Procedure and Region

The Presto procedure scope is nested within the domain scope, and the Presto region scope is nested within the procedure scope. (See Section 1.2 for more information about scope.) To create the scope for the Presto procedure and Presto region, use the following commands:

```
BEGIN PRESTO PROCEDURE <string>presto_procedure_name
  #
  # TIME CONTROL command block
  #
  BEGIN PRESTO REGION <string>presto_region_name
    #
    # command blocks and command lines that appear in the
    # region scope
    #
END [PRESTO REGION <string>presto_region_name]
```

Currently, only the `TIME CONTROL` command block appears within the `PRESTO PROCEDURE` command block and outside of the `PRESTO REGION` command block. These three command blocks (procedure, time control, and region) are discussed below.

Many command blocks and command lines fall within the region scope. These command blocks and command lines are described in other sections of this document.

2.2.1 Presto Procedure

The analysis time, from the initial time to the termination time, is controlled within the procedure scope defined by the `PRESTO PROCEDURE` command block. The command block begins with

```
BEGIN PRESTO PROCEDURE <string>presto_procedure_name
```

and is terminated with

```
END [PRESTO PROCEDURE <string>presto_procedure_name] .
```

The string `presto_procedure_name` is the name for the Presto procedure.

2.2.2 Time Control

Within the procedure scope, there is a `TIME CONTROL` command block. This command block lets the user set the initial time and the termination time for an analysis. This block also gives the user some control over the size of the time step.

Because Presto is an explicit, transient dynamics code, it must run at a time step that is less than the critical time for the problem at any given instant. Typically, this global critical time step is based on a critical time step estimate calculated for each element. With the `TIME CONTROL` command block, the user can set an initial time step, scale the element-based time step estimate, and control the growth of the element-based estimate for the critical time step.

In addition to the element-based method for estimating the time step, Presto also offers a Lanczos-based method and a node-based method for determining a critical time step estimate. The command blocks for these two methods are in the region scope. There is also a mass scaling technique that will influence the magnitude of the critical time step. If you use mass scaling, you must use the node-based method to obtain a critical time step estimate.

The estimation of the time step is a key part of any Presto analysis. Time step determination and control is discussed in detail in Chapter 3 of this document. The `TIME CONTROL` command block with its associated command lines are described in detail in Chapter 3. Consult Chapter 3 to determine how to specify command lines associated with the `TIME CONTROL` command block and how the `TIME CONTROL` command block fits into the overall scheme for time step control in Presto. Also consult with Chapter 3 to learn about the other methods for estimating the critical time step and the mass scaling technique.

2.2.3 Presto Region

Individual time steps are controlled within the region scope. The region scope is defined by a `PRESTO REGION` command block that begins with

```
BEGIN PRESTO REGION <string>presto_region_name
```

and is terminated with

```
END [PRESTO REGION <string>presto_region_name] .
```

The string `presto_region_name` is the name for the Presto region.

The region, as indicated previously, determines what happens at each time step. In the procedure, we set the begin time and end time for the analysis. Time is incremented in the region. It is in the region where we set information about what occurs at various time steps. The output of results, for example, is set by command blocks in the region. If we want results output at certain times or certain steps in the analysis, this information is set in command blocks in the region. The region also contains command blocks for the boundary conditions. A boundary condition can have a time-varying component. The region determines the value of the component for the current time step.

Two of the major types of command blocks, those for results output and boundary conditions, have already been mentioned. Other major types of command blocks in the region are those for restart control and contact. The region is also where the user selects the analysis model (finite element mesh).

The region makes use of information in the procedure and domain. For example, the specific element type used for an element block in the analysis model is defined in the domain. This information about the element type is collected into an analysis model. The region then references this analysis model. As another example, the boundary condition command blocks can reference a function. The function will be defined in the domain.

2.3 Use Finite Element Model

```
USE FINITE ELEMENT MODEL <string>model_name
```

The model specification occurs within the region scope. To specify the model (finite element mesh), use this command line. The string `model_name` must match a name used in a `FINITE ELEMENT MODEL` command block described in Section 5.1. If one of these command blocks uses the name penetrator in the command-block line and this is the model we wish to use in the region, then we would enter the command line as

```
USE FINITE ELEMENT MODEL penetrator .
```

2.4 Error Estimation

Presto incorporates a number of user-defined error estimators. These error estimators can be used to help assess the quality of the solution as the mesh evolves through time. A selected error estimator will calculate the specified error metric on every supported element of the mesh.

The following sections describe the use of the error estimators in Presto.

2.4.1 Error Estimation Controller

```
BEGIN ERROR ESTIMATION CONTROLLER <string>err_name
  ERROR ESTIMATOR = <string>DISTORTION
  COMPUTE METRIC = <string>ASPECT_RATIO/SOLID_ANGLE
    /PERIMETER_RATIO
  COMPUTE STEP INTERVAL = <integer>step_int
  COMPUTE AT OUTPUT
END [ERROR ESTIMATION CONTROLLER <string>err_name]
```

The full definition of an error estimation method is given in an `ERROR ESTIMATION CONTROLLER` command block. This block must be defined in the domain scope, i.e., at the same level as material models and functions. Note that there can be multiple `ERROR ESTIMATION CONTROLLER` command blocks defined in the domain scope. The user-defined error estimation methods are available for use within any Presto region through inclusion of the `USE ERROR ESTIMATION CONTROLLER` command line in the region scope (see Section 2.4.2).

The command block begins with the input line

```
BEGIN ERROR ESTIMATION CONTROLLER <string>err_name
```

and ends with the input line

```
END [ERROR ESTIMATION CONTROLLER <string>err_name] ,
```

where name is a user-selected name for the `ERROR ESTIMATION CONTROLLER` command block. The command lines within the block define what type of error metric to calculate, how to calculate the metric, and when to calculate the metric. The valid commands within this block are described next in Section 2.4.1.1 through Section 2.4.1.3.

2.4.1.1 Error Estimator Class

```
ERROR ESTIMATOR = <string>DISTORTION
```

This command line specifies the type of error metric to calculate. Currently, there is only a single class of error estimators available to Presto, the `DISTORTION` class. The distortion error metric measures the distortion of mesh elements as they deform and distort through time. If an element becomes inverted, the analysis will abort. As an element nears inversion, the solution generally becomes poor. The distortion error metric measures how close an element is to inversion.

2.4.1.2 Distortion Metrics

```
COMPUTE METRIC = <string>ASPECT_RATIO/SOLID_ANGLE
  /PERIMETER_RATIO
```

There are three formulations for error metrics, or error estimates, available within the distortion class: aspect ratio, solid angle, and perimeter ratio. Selection of a metric (or metrics) may be specified by the `COMPUTE METRIC` command line.

If the `ASPECT_RATIO` option is specified, the aspect ratio is computed for tetrahedral elements. A perfect equilateral tetrahedron has an aspect ratio of 1.0. A degenerate zero-volume tetrahedron has an aspect ratio of zero. An inverted tetrahedron has a negative aspect ratio. A very thin element can have very large aspect ratios. The `ASPECT RATIO` option will only work on tetrahedral elements.

If the `SOLID_ANGLE` option is specified, the minimal or maximal angle between edges of an element is computed. The optimal solid angle for tetrahedrons and triangles is 60 degrees; for hexahedrons and quadrilaterals, it is 90 degrees. An element in which all angles are optimal has an error metric of 1, whereas a degenerate element has an error metric of 0 and an inverted element has a negative solid angle. Severely distorted or twisted elements will have poor (near 0) solid angles. The `SOLID ANGLE` option operates on any two-dimensional or three-dimensional element type in Presto.

If the `PERIMETER_RATIO` option is specified, the ratio of the deformed perimeter of an element to the undeformed perimeter of the element is computed. If we take the ratio of the perimeter of an undeformed element to the perimeter of the undeformed element, obviously, we will get a value of one. If, however, we take the ratio of the perimeter of a deformed element to the perimeter of the element in the undeformed state, the perimeter ratio for the deformed element may have a value either larger or smaller than one, depending on the amount of deformation. The `PERIMETER_RATIO` option will only work on three- and four-node two-dimensional elements.

If you want to examine two or three error estimates on the same mesh, you can include any combination of `ASPECT_RATIO`, `SOLID_ANGLE`, and `PERIMETER_RATIO` in the command line. A multiple error estimate request produces all the results that are applicable based on the range of element types. For example, if you want to examine both the aspect ratio and the solid angle on a mesh, you could include both `ASPECT_RATIO` and `SOLID_ANGLE` in the `COMPUTE METRIC` command line. If the mesh consisted solely of tetrahedral and hexahedral elements, an aspect ratio and a solid angle would be computed for all the tetrahedral elements in the mesh; only a solid angle would be computed for the hexahedral elements. The results are automatically stored in output variables when computed, one variable for each metric.

`ASPECT_RATIO`, `SOLID_ANGLE`, and `PERIMETER_RATIO` are treated as element variables. Thus, you can request that the results computed for any of the specified variables be output for a Presto run by specifying an `ELEMENT VARIABLES` command line (Section 8.1.1.4) in the `RESULTS OUTPUT` command block (Section 8.1.1) for each metric for which the results are of interest. As an example, suppose you have specified the `ASPECT_RATIO` option. You can write the values for the aspect ratio for the appropriate elements by including a command line of the form

```
ELEMENT VARIABLES = ASPECT_RATIO as aspect
```

in a `RESULTS OUTPUT` command block. (See Section 8.1.1 for more details about obtaining results output.) In the above command line, the element variable `ASPECT_RATIO` is assigned the name `aspect` on the results output file.

The three distortion metrics can also be used for element death (Section 5.5). As an example, suppose you have specified the `SOLID_ANGLE` option. You can use the solid angle value of an element as an element death criterion by including the following command line:

```
CRITERION IS ELEMENT VALUE OF SOLID_ANGLE < 30.0
```

Any element with a solid angle value less than 30.0 degrees will be killed. For more information about the use of the distortion metrics as an element variable for element death, see Section 5.5.2.2.

2.4.1.3 Utilities

```
COMPUTE STEP INTERVAL = <integer>step_int
COMPUTE AT OUTPUT
```

To control the frequency and output of error metrics, an `ERROR ESTIMATION CONTROLLER` command block may contain one or both of the command lines above.

The `COMPUTE STEP INTERVAL` command line specifies how often the error metrics are computed. The metric will be computed every `step_int` time steps.

The `COMPUTE AT OUTPUT` command line specifies that error estimators should only be computed immediately prior to results output.

An `ERROR ESTIMATION CONTROLLER` block can include the `COMPUTE STEP INTERVAL` command line and the `COMPUTE AT OUTPUT` command line. If both are specified, the error would be computed every n time steps. Additionally, the error would be computed immediately prior to writing an output file to ensure that output values are correct for visualization. Specifying both command lines is reasonable in many analyses. For example, a user may wish to view the correct current error estimate on the mesh when it is output and use the error estimate to compute element death. The `COMPUTE STEP INTERVAL` command line can be used to ensure the error estimator is updated sufficiently often to steer the calculation, but not so often as to incur a major computational cost due to error estimation.

2.4.2 Use Error Estimation Controller

```
USE ERROR ESTIMATION CONTROLLER <string>err_name
```

The activation of an error estimation controller occurs within the region scope. To specify the controller, use this command line. The string `err_name` must match a name used in an `ERROR ESTIMATION CONTROLLER` command block described in Section 2.4.1. If, for example, one of these command blocks uses the name `estim1` in the command-block line and this is the controller we wish to use in the region, then we would enter the command line as

```
USE ERROR ESTIMATION CONTROLLER estim1 .
```

Each Presto region may use at most one of the defined error estimation methods via the `USE ERROR ESTIMATION CONTROLLER` command line.

2.5 Activation/Deactivation of Functionality

```
ACTIVE PERIODS = <string list>period_names
```

The `ACTIVE PERIODS` command line can be used to activate or deactivate functionality in the code at various points during an analysis. This functionality can include such things as boundary conditions, element blocks, and user subroutines. In the command line, the string list `period_names` is a list of the time periods defined in `TIME STEPPING BLOCK` command blocks (see Section 3.1) during which the particular functionality is considered to be active. Each such `period_name` must match a name used in a `TIME STEPPING BLOCK` command block, i.e., `time_block_name`. Each defined time period runs from that period's start time to the next period's start time. Note that if the `ACTIVE PERIODS` command line is present, the functionality will be treated as inactive for any time periods that are not listed. If this command line is absent, then by default, the functionality is active during all time periods. Various other command blocks in Presto will indicate whether they can be used with the `ACTIVE PERIODS` command line.

Intentionally Left Blank

Chapter 3

Time Step Control in Presto

This chapter discusses time control in Presto. We begin with a broad overview of time control in Presto and then describe the options that are available to users for time control.

Time control in Presto begins with the user setting a start time and a termination time for an analysis. The analysis is typically carried out with a large number of time steps, each time step being much smaller than the analysis time. Because Presto is an explicit, transient dynamics code, the time step must be less than some critical value. Presto has a number of schemes to compute an estimate for the critical time step. These will be discussed in detail in later sections.

The primary time control utilizes a `TIME CONTROL` command block that appears in the procedure scope. Use of the `TIME CONTROL` command block gives the user, by default, access to an element-based method for estimating the critical time step. The user can access two other methods (node-based and Lanczos-based) for estimating the critical time step with command blocks in the region scope. These other methods (node-based and Lanczos-based) tend to give better (larger) estimates for the critical time step. Selection of the node-based and Lanczos-based methods is made with command blocks in the region scope.

We begin this chapter with the description of the `TIME CONTROL` command block. Next, we discuss the node-based and Lanczos-based methods for the critical time step estimate. You should read the introductory material for the node-based and Lanczos-based methods and understand it thoroughly before you attempt to use these two methods. Although the node-based and Lanczos-based methods give larger time step estimates than the element-based scheme, they may not result in a net reduction of central processor unit (CPU) time for an analysis unless they are used properly. In those sections dealing with the Lanczos-based and node-based methods, we discuss how to use these two methods in a cost-effective manner. (As we gain more experience with the Lanczos-based method and the node-based method, we may implement an improved time control method that defaults to the node-based or Lanczos-based method rather than to the element-based method.)

Finally, there is a method for adjusting the time step known as mass scaling. Mass scaling is a much different approach for adjusting the time step when compared to the methods (element-, node-, Lanczos-based) we have just been discussing. The last part of this chapter discusses mass scaling. The command block for mass scaling is in the region scope.

3.1 Procedure Time Control

As indicated previously, the primary time control in Presto utilizes a `TIME CONTROL` command block in the procedure scope. The user sets the start time and the termination time for the analysis in this `TIME CONTROL` command block. The analysis time can be subdivided into a number of time blocks. If the total analysis time is from time 0 to time T and there are three blocks, then the first block is defined from time 0 to time t_1, the second block is defined from time t_1 to time t_2, and the third block is defined from time t_2 to time T. (The times t_1 and t_2 are set by the user.) If we sum all of the times for each block, the sum will be T. The different time periods defined by each block can be referenced so that we can turn certain functionality on or off throughout an analysis. For example, we may want to have a certain boundary condition turned off during our first time period and activated for the second time period. (Most analyses require only one block.) Use the `ACTIVE PERIODS` command line (Section 2.5) to activate and deactivate functionality.

If only the `TIME CONTROL` command block is used in an analysis (and the node-based and Lanczos-based methods are not invoked in region command blocks), Presto relies on the element-based critical time step estimate. At every time step, an element-based calculation is performed to determine a critical time step. You have some control over the actual time step that is used with either one of two techniques. We discuss these two techniques in the following paragraphs. The specifics for using these techniques are described in Section 3.1.1.

First, you can set an initial time step that is smaller than the element-based critical time step estimate. Presto will start the analysis with your initial time step value rather than with the element-based critical time step (as long as your value is less than the element-based critical time step). You can then control the rate at which the time step increases from your initial value.

- If you set a time step increase factor equal to one, then the initial value you specified will be used throughout the analysis (provided the initial time step is never smaller than the element-based critical time step throughout the computations).

- If you set a time step increase factor to some value greater than one, the time step will grow (from the initial value) at each time step until it reaches the value of the element-based critical time step estimate. From then on, the element-based critical time step estimate will essentially control the time step.

Second, you can manipulate the element-based estimate with either a scale factor or a time step increase factor.

- The element-based estimate for the critical time step is usually smaller than some maximum theoretical value for your model. It may therefore be possible to scale the element-based critical time step by some factor greater than one. (Your scaled value must remain below the theoretical maximum limit, however. We discuss ways to obtain a critical time step close to the theoretical maximum in later sections of this chapter.)
- If there are stability problems with a particular problem, it may be necessary to scale the element-based estimate with a factor less than one.
- You can also control the rate at which the time step can increase for an analysis. By specifying a time step increase factor, you can limit the increase in the size of the time step so that it does not increase too rapidly from one step to the next. For certain problems, the element-based critical time step estimate may increase rapidly from one step to the next. You may want to limit the increase in the time step size for reasons of either stability or accuracy.

Now that we have presented an overview of the functionality in the `TIME CONTROL` command block, we will discuss the actual command lines.

3.1.1 Command Blocks for Time Control and Time Stepping

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK <string>time_block_name
    START TIME = <real>start_time_value
    BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
      #
      # Time control parameters specific to PRESTO
      # are set in this command block.
      #
    END [PARAMETERS FOR PRESTO REGION <string>region_name]
  END [TIME STEPPING BLOCK <string>time_block_name]
  TERMINATION TIME = <real>termination_time ,
END [TIME CONTROL]
```

Presto time control resides in a `TIME CONTROL` command block. The command block begins with

```
BEGIN TIME CONTROL
```

and terminates with

```
END [TIME CONTROL] .
```

Within the `TIME CONTROL` command block, a number of `TIME STEPPING BLOCK` command blocks can be defined. Each `TIME STEPPING BLOCK` command block contains the time at which the time stepping starts and a number of parameters that set time-related values for the analysis. Each `TIME STEPPING BLOCK` command block terminates at the start time of the following command block. The start times for the `TIME STEPPING BLOCK` command blocks must be in increasing order. Otherwise, an error will be generated by Presto. (The example in Section 3.1.6 shows the overall structure of the `TIME CONTROL` command block.)

In the above input lines, the values are as follows:

- The string `time_block_name` is a name for the `TIME STEPPING BLOCK` command block. This name must be unique to the other command blocks of this type. The string `time_block_name` can be referenced on an `ACTIVE PERIODS` command line to activate and deactivate functionality (see Section 2.5).
- The real value `start_time_value` is the start time for this `TIME STEPPING BLOCK` command block. Values set by the block apply from the start time for this block until the next start time or the termination time.
- The string `region_name` is the name of the Presto region affected by the parameters (see Section 2.2).

The final termination time for the analysis is given by the command line

```
TERMINATION TIME = <real>termination_time ,
```

where `termination_time` is the time at which the analysis will be stopped. The `TERMINATION TIME` command line appears inside the `TIME CONTROL` scope but outside of any `TIME STEPPING BLOCK` command block.

The `TERMINATION TIME` command line can appear before the first `TIME STEPPING BLOCK` command block or after the last `TIME STEPPING BLOCK` command block. Note that it is permissible to have `TIME STEPPING BLOCK` command blocks with start times after the termination time; in this case, those command blocks that have start times after the termination time are not executed. Only one `TERMINATION TIME` command line can appear. If more than one of these command lines appears, Presto gives an error.

Nested inside the `TIME STEPPING BLOCK` command block is a `PARAMETERS FOR PRESTO REGION` command block containing parameters that control the time stepping.

```
BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
  INITIAL TIME STEP = <real>initial_time_step_value
  TIME STEP SCALE FACTOR = <real>time_step_scale_factor(1.0)
  TIME STEP INCREASE FACTOR =
    <real>time_step_increase_factor(1.1)
  STEP INTERVAL = <integer>nsteps(100)
END [PARAMETERS FOR PRESTO REGION <string>region_name]
```

These parameters are specific to a Presto analysis.

The command block begins with

```
BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
```

and is terminated with

```
END [PARAMETERS FOR PRESTO REGION <string>region_name] .
```

As noted previously, the string `region_name` is the name of the Presto region affected by the parameters. The command lines nested inside the `PARAMETERS FOR PRESTO REGION` command block are described next.

3.1.2 Initial Time Step

```
INITIAL TIME STEP = <real>initial_time_step_value
```

By default, Presto computes a critical time step for the analysis and uses this as the initial time step. To directly specify a different initial time step, use the `INITIAL TIME STEP` command line, where `initial_time_step_value` is the size of the initial time step. This command is only valid if it is in the first `TIME STEPPING BLOCK` command block in the problem.

The value for the initial time step will overwrite the calculated critical time step. If you specify an initial time step larger than the critical time step, the time step is set to the calculated critical time step.

3.1.3 Time Step Scale Factor

```
TIME STEP SCALE FACTOR = <real>time_step_scale_factor(1.0)
```

During the element computations, Presto computes a minimum time step required for stability of the computation (the critical time step). Using the `TIME STEP SCALE FACTOR` command line, you can provide a scale factor to modify the critical time step. Note that a value greater than 1.0 for `time_step_scale_factor` will cause the time step to be greater than the computed critical time step, and thus the problem will likely go unstable. By default, the scale factor is 1.0.

3.1.4 Time Step Increase Factor

```
TIME STEP INCREASE FACTOR =
  <real>time_step_increase_factor(1.1)
```

During an analysis, the computed critical time step may change as elements deform, are killed, etc. By using the `TIME STEP INCREASE FACTOR` command line, you can limit the amount that the time step can increase between two adjacent time steps. The value `time_step_increase_factor` is a factor that multiplies the previous time step. The current time step can be no larger than the product of the previous time step and the scale factor.

Note that an increase factor less than 1.0 will cause the time step to continuously decrease. The default value for this factor is 1.1, i.e., a time step cannot be more than 1.1 times the previous step.

3.1.5 Step Interval

```
STEP INTERVAL = <integer>nsteps(100)
```

Presto can output data about the current time step, the current internal and external energy, and the kinetic energy throughout an analysis. The `STEP INTERVAL` command line controls the frequency of this output, where `nsteps` is the number of time steps between output. The default value for `nsteps` is 100.

The output at any given step (read from left to right) is

- step number,
- time,
- time increment,
- kinetic energy,
- internal energy,

- external energy (work done on boundary),
- error in energy balance,
- cpu time, and
- wall clock time.

The time is at the current time, step n, and the time increment is the previous time step increment from step $n-1$ to step n.

The error in the energy balance is computed from the relation

```
energy balance error = (kinetic energy + internal energy
  - external energy) / external energy * 100 .
```

The above expression gives a percent error for the energy balance.

3.1.6 Example

The following is a simple example of a `TIME CONTROL command` block:

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK p1
    START TIME = 0.0
    BEGIN PARAMETERS FOR PRESTO REGION presto_region
      INITIAL TIME STEP = 1.0e-6
      STEP INTERVAL = 50
    END
  END
  BEGIN TIME STEPPING BLOCK p2
    START TIME = 0.5e-3
    BEGIN PARAMETERS FOR PRESTO REGION presto_region
      TIME STEP SCALE FACTOR = 0.9
      TIME STEP INCREASE FACTOR = 1.5
      STEP INTERVAL = 10
    END
  END
  TERMINATION TIME = 1.0e-3
END
```

The first `TIME STEPPING BLOCK`, `p1`, begins at time 0.0, the initial start time, and terminates at time 0.5×10^{-3}. The second `TIME STEPPING BLOCK`, `p2`, begins at time 0.5×10^{-3} and terminates at time 1.0×10^{-3}, the time listed on the `TERMINATION TIME` command line. The `TIME STEPPING BLOCK` names `p1` and `p2` can be referenced by `ACTIVE PERIODS` command lines (Section 2.5) to activate and deactivate functionality.

3.2 Other Critical Time Step Methods

At present, there are three methods for calculating a critical time step for Presto. First, there is the traditional element-based method. We know that, in general, the element-based time step in Presto can give a fairly conservative estimate for the time step. Second, there is a node-based method for giving a critical time step estimate. Depending on the problem, the node-based method may or may not give an estimate for the critical time step that approaches the theoretical maximum value for a particular model. Although the node-based method will give a larger critical time step estimate than the element-based method in most cases, the node-based estimate may still be significantly lower than the maximum theoretical time step for a problem. Third, there is the use of the Lanczos method to obtain an estimate for the critical time step. The Lanczos method can give an accurate estimate of the maximum eigenvalue for a problem using a very small number of Lanczos vectors compared to the total number of degrees of freedom in a problem. From the maximum eigenvalue, it is possible to derive the theoretical maximum critical time step for a problem.

Use of a critical time step from the node-based method or the Lanczos method is desirable because the larger critical time steps produced by these methods (compared to the element-based method) reduce CPU time. Both methods, however, are not cost-effective if they are called every time step to give a critical time step estimate. The cost of doing one node-based estimate or one Lanczos-based estimate for the critical time step will not offset the cost benefit of the increase to the critical time step (compared to the element-based time step estimate) over a single time step. Hence, there must be some scheme for

- calling these methods only periodically throughout a calculation and
- maintaining a larger estimate (than the element-based estimate) for the critical time step in between these calls

if we are to gain a net benefit from the increase in the critical time step these methods can produce.

We begin with a discussion of the Lanczos method and then follow with a discussion of the node-based method.

3.2.1 Lanczos Method

The Lanczos method is a more attractive method for computing the critical time step in an explicit, transient dynamics code than typical element-based methods in that the Lanczos method can give a significantly larger time step estimate. However, the Lanczos method represents a more expensive method for calculating a critical time step than element-based methods. Over one time step, it is not possible to recoup the cost of the Lanczos calculations with the increase in the time step size. To use the Lanczos method for estimating the critical time step in an explicit, transient dynamics code, some methodology must be developed so that the Lanczos method is used in a cost-effective manner. The following sections outline the development of a cost-effective approach to using the Lanczos method in an explicit, transient dynamics code.

3.2.1.1 Lanczos Method with Constant Time Steps

To understand how we can effectively use the Lanczos method in an explicit, transient dynamics code, we begin with a simple case study. In this case study, we compute the critical time step using the Lanczos method at some time step and then assume this critical time step value remains constant for a subsequent number of time steps n_L. We only call the Lanczos method once during the n_L time steps. (In reality, the critical time step in an explicit, transient dynamics code like Presto changes with each time step. We address this issue of the changing time step when we present the details for cost-effective use of the Lanczos method.)

As indicated previously, the cost of a Lanczos vector is approximately the cost of an internal force calculation. Over a given time step, the cost of the internal force calculation is the major computational cost. (This assumes no contact. The addition of contact introduces another computationally expensive process into a time step. We address the issue of contact later on.) For this case study, we will assume that the computational cost of an element-based estimate for the critical time step is part of the cost of an internal force calculation. The cost of the element-based time step estimate is a small part of the overall internal force calculations.

Assume that the Lanczos method computes a global estimate for the critical time step of Δt_L, which is the value to be used for n_L time steps. At the end of the n_L time steps, the analysis time for the code has been incremented by an amount ΔT, which is computed simply as

$$\Delta T = n_L \Delta t_L \,. \tag{3.1}$$

If the element-based estimate for the time step is Δt_e and the number of time steps required to increment the analysis time by ΔT is n_e, then, for the element-based time step, we have

$$\Delta T = n_e \Delta t_e \,. \tag{3.2}$$

Because the Lanczos estimate for the critical time step is larger than the element-based estimate, we know that $n_e > n_L$. Let us define the ratio r as

$$r = \Delta t_L / \Delta t_e = n_e / n_L \,. \tag{3.3}$$

The ratio r is greater than one.

Now that we have determined the relation between the number of steps required for a Lanczos-based critical time step estimate versus the element-based critical time step estimate to achieve the same analysis time increment, let us examine the computational costs for these two cases in terms of CPU time. Designate the CPU cost for a time step as Δt_{IF}. If the number of Lanczos vectors required to obtain the critical time step estimate is N_L, then the total computational cost of the Lanczos method and the n_L time steps is

$$n_L \Delta t_{IF} + N_L \Delta t_{IF} \,. \tag{3.4}$$

The total computational cost, if we use the element-based method, is

$$n_e \Delta t_{IF} \;. \tag{3.5}$$

Recall that we have chosen n_L and n_e so that we have the same analysis time increment ΔT even though we have different critical time steps. Now determine the the point at which the computational cost for the Lanczos-based critical time step calculations is the same as the cost for the element-based critical time step calculations. This is simply the point at which

$$n_e \Delta t_{IF} = n_L \Delta t_{IF} + N_L \Delta t_{IF} \,. \tag{3.6}$$

If we rearrange the above equation to eliminate Δt_{IF} and make use of the ratio r, then we obtain

$$n_e = \frac{N_L}{1 - 1/r} \,. \tag{3.7}$$

Consider the case of $r = 1.25$ and $N_L = 20$. The above equations show that for $n_L = 80$ and $n_e = 100$, the calculations with the Lanczos-based critical time step and the calculations with the element-based time step give the same analysis time for the same computational expense. If we use the Lanczos-based critical time step Δt_L for more than 80 iterations, then the Lanczos-based approach becomes cost-effective. Our above equations have established the "break-even" point at which it becomes cost-effective to use the Lanczos method to reduce computational costs by overcoming the initial cost of the Lanczos calculations with the larger critical time step.

We can build on what we have done so far to account for contact. Suppose that the computational cost of contact over a time step is some multiple m of the computational cost of the internal force calculation Δt_{IF}. Then the point at which the computational cost for the Lanczos-based calculations is the same as the computational cost for the element-based calculations is

$$(1+m)n_L\Delta t_{IF} + N_L\Delta t_{IF} = (1+m)n_e\Delta t_{IF}\,. \tag{3.8}$$

For the case with contact,

$$n_e = \frac{N_L}{(1+m)(1-1/r)}\,. \tag{3.9}$$

Again, consider the case of $r = 1.25$ and $N_L = 20$. Assume the computational cost of contact calculations is the same as an internal force calculation ($m = 1$). For these values, the break-even point is $n_L = 40$ and $n_e = 50$. The added computational cost of the contact calculations results in reaching break-even with a smaller number of iterations when compared to the case with no contact.

The above derivations let us calculate a break-even point based on our assumptions of a constant critical time step. Considering that a typical analysis will run for tens of thousands of time steps, something on the order of 100 steps represents a reasonable number of steps to recoup the cost of the Lanczos calculations. Whether or not the cost of the Lanczos calculations can be recouped in something on the order of 100 calculations depends heavily upon N_L. If N_L is sufficiently small, we can recoup the cost of the Lanczos calculations in a reasonable number of steps.

Some computational studies indicate that N_L is in an acceptable range for many problems. The Lanczos method computes a good estimate for the maximum eigenvalue with a small number of Lanczos vectors, N_L, compared to the number of degrees of freedom in a problem. Some component studies show that, for a problem with between 250,000 and 350,000 degrees of freedom, one can obtain a good estimate for the maximum eigenvalue with only 20 Lanczos vectors. A large-scale study of a model involving 1.7 million nodes (5.1 million degrees of freedom) showed that only 45 Lanczos vectors were required to obtain a good estimate of the maximum eigenvalue. These examples show that the number of Lanczos vectors required for a good maximum eigenvalue estimate is very small when compared to the number of degrees of freedom for a problem. When N_L is in the range of 20 to 45, Equations 3.6 and 3.9 show that, with an increase in the time step on the order of 1.2 to 1.25, we can recoup the cost of the Lanczos method in a reasonable number of time steps.

Now that we have determined we can recoup the cost of the Lanczos calculations in a reasonable number of time steps, let us look a the issue of reusing a Lanczos-based estimate in some manner.

3.2.1.2 Controls for Lanczos Method

The above discussion indicates, that, if we can perform a Lanczos calculation and reuse the Lanczos-based estimate for the critical time step in some way over a number of subsequent time steps, then the Lanczos method can be cost-effective for use in an explicit, transient dynamics code. This section presents a method for reusing a Lanczos-based estimate over a number of time steps so that we maintain a critical time step estimate that is close to the theoretical maximum value in between the calls to the Lanczos method. The method we discuss here makes use of the element-based critical time step estimate at each time step.

We start our method with a Lanczos calculation to determine the maximum eigenvalue. The Lanczos method converges to the maximum eigenvalue from below, which means that the method underestimates the maximum eigenvalue. Because the critical time step depends on the inverse of the maximum eigenvalue, we overestimate the critical time step. It is necessary, therefore, to scale back the critical time step estimate from the Lanczos method so that the calculations in the explicit time-stepping scheme do not become unstable. The method to determine a scaled-back value for the maximum critical time step makes use of the element-based time step estimate. Again, let Δt_L be the critical time step estimate from the Lanczos method and Δt_e be the critical time step estimate from the element-based calculations. The scaled-back estimate for the critical time step, Δt_s, is computed from the equation

$$\Delta t_s = \Delta t_e + f_s(\Delta t_L - \Delta t_e)\,, \tag{3.10}$$

where f_s is a scale factor. (A reasonable value for f_s ranges from 0.9 to 0.95 for our problems.) This value of f_s puts Δt_s close to and slightly less than a theoretical maximum critical time step.

Once Δt_s is determined, the ratio

$$t_r = \Delta t_s / \Delta t_e \tag{3.11}$$

is computed. This ratio is then used to scale subsequent element-based estimates for the critical time step. If $\Delta t_{e(n)}$ is the n^{th} element-based critical time step after the time step where the Lanczos calculations are performed, then the n^{th} time step after the Lanczos calculations, $\Delta t_{(n)}$, is simply

$$\Delta t_{(n)} = t_r \Delta t_{e(n)}\,. \tag{3.12}$$

The ratio t_r is used until the next call to the Lanczos method. The next call to the Lanczos method is controlled by one of two mechanisms. First, the user can set the frequency with which the Lanczos method is called. The user can set a parameter so that the Lanczos method is called only once every n time steps. This number remains fixed throughout an analysis. Second, the user can control when the Lanczos method is called based on changes in the element-based critical time step. For this second method, the change in the element-based critical time step estimate is tracked. Suppose the element-based critical time step at the time the Lanczos method was called is Δt_e. At the n^{th} step after the call to the Lanczos method, the element-based critical time step is $\Delta t_{e(n)}$. If the value

$$\Delta t_{lim} = \frac{|\Delta t_{e(n)} - \Delta t_e|}{\Delta t_e} \tag{3.13}$$

is greater than some limit set by the user, then the Lanczos method will be called. If there is a small, monotonic change in the element-based critical time step over a large number of time steps, this second mechanism will result in the Lanczos method being called. Or, if there is a large, monotonic change in the element-based critical time step over a few time steps, the Lanczos method will also be called.

These two mechanisms for calling the Lanczos method can be used together. For example, suppose the second mechanism, the mechanism based on a change in the element-based time step, results in a call to the Lanczos method. This resets the counter for the first mechanism, the mechanism using a set number of time steps between calls to the Lanczos method.

This method for reusing a Lanczos-based time step estimate has been implemented in Presto, and it has been used for a number of studies. One of the component studies, as indicated previously, used the same scale factor for $n_L = 1700$ iterations. The break-even point for this problem is $n_e = 45$ time steps (not accounting for contact, which was a part of the component modeling). For this particular problem, the extended use of the Lanczos estimate reduced the computational cost to 56% of what it would have been with the element-based time step.

Not all problems will lend themselves to reuse of one Lanczos-based estimate for thousands of time steps. However, if it is possible to use the Lanczos-based estimate for two to three times the number of time steps required for break-even, we begin to see a noticeable reduction in the total number of time steps required for a problem.

3.2.1.3 Scale Factor for Lanczos Method

In addition to understanding how to use the Lanczos method in a cost-effective manner, one must also be aware that the Lanczos method makes use of the action of the global stiffness matrix $\mathbf{K}$ on a Lanczos vector $\mathbf{q}_j$. This action of $\mathbf{K}$ on $\mathbf{q}_j$ is the internal force computation. We do not construct a $\mathbf{K}$ matrix, but simply provide a $\mathbf{q}_j$ vector for the internal force calculations in Presto. The internal force calculations give us the desired matrix $\times$ vector product. You, the user, do not see the

$\mathbf{q}_j$ vectors. The $\mathbf{q}_j$ must be scaled so that they represent velocities associated with small strain. When these scaled vectors are sent to the internal force calculation, the internal force calculation becomes a matrix $\times$ vector product with a constant tangent stiffness matrix $\mathbf{K}_T$.

A number of tests have established a scale factor that works well for a range of models encountered here at Sandia National Laboratories. There is a default value for this scale factor, but the default value can be changed by the user. The scale factor must not be too small, as this will create round off problems and give a bad estimate for the critical time step. If the scale factor is too large, we violate the above restriction of a constant tangent stiffness matrix $\mathbf{K}_T$.

The user can always determine whether a particular scale factor is suitable for a particular problem. Take a scale factor v_{sf}, plus values on either side of it, say $0.9 \times v_{sf}$ and $1.1 \times v_{sf}$. If all three of these scale factor values produce very nearly the same estimate for the critical time step for a particular model, then the value for v_{sf} meets the criteria for an acceptable scale factor.

Work is under way to develop an automated scheme to determine a good value for the scale factor v_{sf}.

3.2.1.4 Lanczos Parameters Command Block

```
BEGIN LANCZOS PARAMETERS <string>lanczos_name
  NUMBER EIGENVALUES = <integer>num_eig(20)
  STARTING VECTOR = <string>STRETCH_X|STRETCH_Y|STRETCH_Z
    (STRETCH_X)
  VECTOR SCALE = <real>vec_scale(1.0e-5)
  TIME SCALE = <real>time_scale(0.9)
  STEP INTERVAL = <integer>step_int(500)
  INCREMENT INTERVAL = <integer>incr_int(5)
  TIME STEP LIMIT = <real>step_lim(0.10)
END [LANCZOS PARAMETERS <string>lanczos_name]
```

If you use the Lanczos method to compute a critical time step, there should be only one `LANCZOS PARAMETERS` command block, and it should appear in the region. If you use the Lanczos method to compute the critical time step, you should set the time step scale factor to 1.0 and the time step increase factor to a large number (5.0 is an acceptable value for the time step increase factor). If you have a `LANCZOS PARAMETERS` command block, you may not specify a command control block for the node-based method.

The `NUMBER EIGENVALUES` command line selects the number of eigenvalues to be computed by the Lanczos method. We are really only interested in the maximum eigenvalue computed by the Lanczos method, as this is what is used for the critical time step computation. The more eigenvalues computed, the better the estimate for the maximum eigenvalue, and, hence, the critical time step. The Lanczos method can compute an accurate value for the maximum eigenvalue with a very small number of total eigenvalues computed compared to the number of degrees of freedom in a problem. There are examples of problems with 250,000 to 350,000 degrees of freedom where we obtain a good estimate of the critical time step with 20 eigenvalues (the default value). There is an example of a problem with 5.1 million degrees of freedom where we obtain a good estimate of the critical time step with 45 eigenvalues.

The Lanczos method requires some type of starting vector. This is determined by the `STARTING VECTOR` command line. The various options will generate a ”displacement” vector that stretches your model in the x-, y-, or z-direction. If your model has its longest dimension in the x-direction, you should use `STRETCH_X` (the default) for the starting vector. The Lanczos method appears to be fairly insensitive to the choice of a starting vector. However, by choosing a starting vector that reflects the geometry of the model, you may gain a slight amount of accuracy for the critical time step estimate for a given number of eigenvalues.

The `VECTOR SCALE` command line sets the scale factor v_{sf} discussed in the preceding section. Please consult with the previous section to determine how this scale factor is set. The default value is 1.0e-5.

The `TIME SCALE` command line sets the factor f_s in Equation (3.10). The value for f_s is set to 0.9. One can probably run with a slightly higher value for f_s, 0.95, for most problems. More experience with the Lanczos method will help us determine the appropriate value for f_s.

The `STEP INTERVAL` command line sets the number of step intervals at which the Lanczos method is called. If `step_int` is set to 1000, the Lanczos method will be called every 1000 steps to compute an estimate for the critical time step. (The default is 500.) This control mechanism interacts with the control established by the `TIME STEP LIMIT` command line. Suppose we have set `step_int` to 1000, and we have computed 800 steps since the last call to the Lanczos method. If Δt_{lim} has been set to 0.15 and we exceed this value at step 800, then the change in the element-based time step will result in the Lanczos method being called. The counter for keeping track of the number of step intervals since the last Lanczos computation will be reset to zero. The next call to the Lanczos method will then be in 1000 steps, unless we again exceed the change in the element-based time step.

The `INCREMENT INTERVAL` command line determines how many steps are used to transition from an element-based critical time step estimate to the Lanczos-based estimate at the beginning of an analysis. The user may want to increase from the element-based estimate to the Lanczos-based estimate over a number of time steps if the difference between these two estimates is large. The `INCREMENT INTERVAL` defaults to 5.

The `TIME STEP LIMIT` command line sets the value for Δt_{lim} in Equation (3.13). If the change in the element-based critical time step estimate as given by Equation (3.13) exceeds the value for Δt_{lim}, then the Lanczos method is called for a new estimate for the critical time step. The default value for Δt_{lim} is 0.10.

3.2.2 Node-Based Method

Now that we have developed a scheme to make the Lanczos method a cost-effective tool for the estimation of the critical time step, let us examine the node-based scheme.

The node-based method in Presto will give an estimate for the critical time step that is greater than or equal to the element-based estimate. It may or may not give an estimate for the time step that is close to the maximum value associated with the maximum eigenvalue for the problem. In general, we assume the node-based method will give us an estimate larger than the element-based estimate, but not significantly larger.

If the node-based scheme is used to determine the critical time step for a block of uniform elements (all the same material), then the estimate from the node-based method will be the same as that for the element-based estimate. The node-based estimate begins to diverge from (become larger than) the element-based estimate as the differences in aspect ratios of the elements attached to a node become larger. We can assume, therefore, that if the stiffest part of our structure has a relatively uniform mesh, the node-based method will not give a significantly larger estimate than the element-based method.

The node-based method costs only a fraction of an internal force calculation. However, the node-based method makes use of element time step estimates. Therefore, in order to do the node-based procedure, one must also do the element-based procedure. The cost of the node-based procedure is in addition to the element-based procedure. The modest increase in the critical step estimate from the node-based procedure is unlikely to offset the added cost of the node-based procedure.

To make the node-based method cost-effective, we can use the same procedures that are employed for the Lanczos method. Let t_b be the estimate for the critical time step from the node-based method. We define the ratio of the node-based estimate to the element-based estimate as

$$t_r = \frac{\Delta t_b}{\Delta t_e}. \tag{3.14}$$

This ratio is then used to scale subsequent element-based estimates for the critical time step. If $\Delta t_{e(n)}$ is the n^{th} element-based critical time step after the time step where the node-based calculations are performed, then the n^{th} time step after the node-based calculations, $\Delta t_{(n)}$, is simply

$$\Delta t_{(n)} = t_r \Delta t_{e(n)} . \qquad (3.15)$$

The ratio t_r is used until the next call to the node-based method. The next call to the node-based method is controlled by one of two mechanisms described in the Lanczos discussion. The node-based method is called after a set number of times or a significant change in the element-based estimate for the critical time step.

3.2.2.1 Node-Based Parameters Command Block

```
BEGIN NODE BASED TIME STEP PARAMETERS <string>nbased_name
  INCREMENT INTERVAL = <integer>incr_int(5)
  STEP INTERVAL = <integer>step_int(500)
  TIME STEP LIMIT = <real>step_lim(0.10)
END [NODE BASED TIME STEP PARAMETERS <string>nbased_name]
```

If you use the node-based method to compute a critical time step, there should be only one `NODE BASED TIME STEP PARAMETERS` command block, and it should appear in the region. If you use the node-based method to compute the critical time step, you should set the time step scale factor to 1.0 and the time step increase factor to a large number (5.0 is an acceptable value for the time step increase factor). If you have a `NODE BASED TIME STEP PARAMETERS` command block, you may not specify a command control block for the Lanczos method.

The `STEP INTERVAL` command line sets the number of step intervals at which the node-based method is called. If `step_int` is set to 1000, the node-based method will be called every 1000 steps to compute an estimate for the critical time step. (The default is 500.) This control mechanism interacts with the control established by the `TIME STEP LIMIT` command line. Suppose we have set `step_int` to 1000, and we have computed 800 steps since the last call to the node-based method. If Δt_{lim} has been set to 0.15 and we exceed this value at step 800, then the change in the element-based time step will result in the node-based method being called. The counter for keeping track of the number of step intervals since the last node-based computation will be reset to zero. The next call to the node-based method will then be in 1000 steps, unless we again exceed the change in the element-based time step.

The `INCREMENT INTERVAL` command line determines how many steps are used to transition from an element-based critical time step estimate to the node-based estimate at the beginning of an analysis. The user may want to increase from the element-based estimate to the node-based estimate over a number of time steps if the difference between these two estimates is large. The `INCREMENT INTERVAL` defaults to 5.

The `TIME STEP LIMIT` command line sets the value for Δt_{lim} in Equation (3.13). If the change in the element-based critical time step estimate as given by Equation (3.13) exceeds the value for Δt_{lim}, then the node-based method is called for a new estimate for the critical time step. The default value for Δt_{lim} is 0.10.

3.3 Mass Scaling

3.3.1 What is Mass Scaling?

WARNING: USE OF MASS SCALING WILL INTRODUCE ERROR INTO YOUR ANALYSIS. THE AMOUNT OF ERROR INCURRED IS UNBOUNDED AND CAN BE UNPREDICTABLE. IT IS ENTIRELY UP TO THE ANALYST TO DECIDE WHETHER MASS SCALING CAN BE USED IN A WAY THAT DOES NOT DISTORT THE RESULTS OF INTEREST.

Mass scaling allows for arbitrarily increasing the mass of certain nodes in order to increase the global estimate for the critical time step. The nodes where the mass is increased must be associated with those elements that have the minimum time step. By increasing the mass at any node for an element, we have effectively raised the critical time step estimate for that element.

Note that mass scaling does not adjust the value for the density used in the element calculations. Mass scaling only adjusts the mass at nodes. The net effect of the mass scaling makes it appear, however, as if we have modified the density of selected elements (even though no adjustment has been made to element densities).

Mass scaling can be useful in a number of circumstances, as listed below. However, in all of these circumstances, error will be introduced into the calculations. The user must be extremely careful not to introduce excessive error.

- Quasi-static or rigid-body motion: If the model or part of a model is undergoing what is basically quasi-static or rigid-body motion, then adding mass may have little effect on the end result.

- Disparate sizes in element geometry: Some models may contain elements for some portion of the model that are much smaller than the majority of elements in the rest of the model. For example, a model might include screws or gears that are modeled in detail. The elements for the screw threads or gear teeth could be much smaller than elements in other portions of the model. If the dynamics of these parts modeled with small elements (compared to the rest of the mesh) are relatively unimportant, adding mass to them might not affect the quantities of interest.

- Increasing time step for "unimportant" sections of the mesh: For some problems, you may not want part of the mesh to control the time step. Consider a car-crash problem in which the bumper is the first part to strike an object. The crumpling of the bumper could greatly reduce the time step in some elements of the bumper, and these elements would control the time step for the problem. At a later time in the analysis, the effect of the bumper on the overall crash dynamics may not be significant. Mass scaling could be applied to ensure that this now noncritical part (the bumper) is no longer controlling the global time step.

3.3.2 Mass Scaling Command Block

```
BEGIN MASS SCALING
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list> nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  TARGET TIME STEP = <real>target_time_step
  ALLOWABLE MASS INCREASE RATIO = <real>mass_increase_ratio
  #
  # additional command
  ACTIVE PERIODS = <string list>periods
END MASS SCALING
```

The `MASS SCALING` command block controls mass scaling for a specified set of nodes. This command block contains one or more command lines to specify the node set. It also contains two command lines that determine how the actual mass scaling will be applied to the nodes in the node set. In addition to the command lines in the two command groups, there is an additional command line: `ACTIVE PERIODS`. The `ACTIVE PERIODS` command line is used to active or deactivate the mass scaling for certain time periods.

Multiple `MASS SCALING` command blocks can exist to apply different criteria to different portions of the mesh at different times. For any given set of `MASS SCALING` command blocks, mass will only be added to a node if doing so will allow increasing the global time step. The amount of artificial mass added to a node will vary in time as the mesh deforms and moves. The added mass computation is redone every time the nodal-based time step estimate is recomputed.

NOTE: Mass scaling must be used in conjunction with the node-based time step estimation method. Consult with the preceding sections for a description of the node-based method for estimating the critical time step.

Following are descriptions of the different command groups and the `ACTIVE PERIODS` command line.

3.3.2.1 Node Set Commands

The {`node set commands`} portion of the `MASS SCALING` command block specifies the nodes associated with the boundary condition. This portion of the command block can include some combination of the following lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list> nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes for mass scaling. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

3.3.2.2 Mass Scaling Commands

The `MASS SCALING` command block may contain either a

```
TARGET TIME STEP = <real>target_time_step
```

command line or an

```
ALLOWABLE MASS INCREASE RATIO = <real>mass_increase_ratio
```

command line, or both of these command lines can appear in the input block.

The `TARGET TIME STEP` command lines sets the maximum time step for a set of nodes. The parameter `target_time_step` is the maximum time step for all of the nodes specified in the command block.

The `ALLOWABLE MASS INCREASE RATIO` sets an upper limit on the mass scaling at a node. The `mass_increase_ratio` limits the ratio of the mass at a node, as set by mass scaling, to the original mass at the node. (The original mass of the node is determined only by the element contributions.) This ratio must be a factor greater than or equal to one. If m_s is the scaled mass at a node and m_0 is the original mass at the node due only to element contributions, then the ratio $m_s \div m_0$ will not exceed `mass_increase_ratio`.

Mass scaling will add mass to nodes until the target time step is reached, the mass added to some node reaches the allowable mass increase ratio, or the current set of nodes no longer controls the global analysis time step.

The amount of mass added due to mass scaling is stored in the nodal variable `mass_scaling_added_mass`. This variable can be output and postprocessed to determine how much mass is being added at a given time. See Section 8.1.1 regarding the output of nodal variables to a results file.

3.3.2.3 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `MASS SCALING` command block:

```
ACTIVE PERIODS = <string list>periods
```

This command line determines when mass scaling is active. See Section 2.5 for more information about this command line.

Chapter 4

Materials

The chapter summarizes the input for various material models available in Presto. It also describes the specifications for activating thermal strains and for implementing energy deposition for energy-dependent materials.

The material models described in the following sections are, in general, applicable to solid elements. The structural elements (shells, beams, etc.) have a much more limited set of material models. You should consult with Chapter 5, the chapter on the element library in Presto, to determine what material models are available for the various elements. The introduction to Chapter 5 provides a summary of all the element types in Presto. For each element type, there is a list of available material models.

When using the nonlinear material models, you may want to output state variables associated with these models. Reference Section 8.5.2 to learn how to output the state variables for the various nonlinear material models.

4.1 Property Specification

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  #
  # Command blocks and command lines for material
  # models appear in this scope.
  #
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

`PROPERTY SPECIFICATION FOR MATERIAL` command blocks appear in the domain scope in the general form shown above. These command blocks are physics independent in the sense that the information in them can be shared by more than one application. For example, the `PROPERTY SPECIFICATION FOR MATERIAL` command blocks contain density information that can be shared among several applications.

The command block begins with the input line

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
```

and is terminated with the input line

```
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name] ,
```

where the string `mat_name` is a user-specified name for the command block.

Within a `PROPERTY SPECIFICATION FOR MATERIAL` command block, there will be other command blocks and command lines that describe particular material models. These material models are described by a set of material-model command blocks that follow the naming convention of `PARAMETERS FOR MODEL model_name`, where `model_name` identifies a particular material model, such as elastic, elastic-plastic, or orthotropic crush. Each such command block contains all the parameters needed to describe a particular material model. NOTE: More than one material-model command block can appear within a `PROPERTY SPECIFICATION FOR MATERIAL` command block. Suppose we have a `PROPERTY SPECIFICATION FOR MATERIAL` command block called `steel`. It would be possible to have two material-model command blocks within this command block. One of the material-model command blocks would provide an elastic model for `steel`; the other material-mode command block would provide an elastic-plastic model for `steel`. The general form of a property command block would be as follows:

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  #
  {General material commands}
  #
  BEGIN PARAMETERS FOR MODEL <string>model_name1
    {Parameters for material model model_name1}
  END PARAMETERS FOR MODEL <string>model_name1
  #
  BEGIN PARAMETERS FOR MODEL <string> model_name2
    {Parameters for material model model_name2}
  END PARAMETERS FOR MODEL <string> model_name2
  #
  {Additional model command blocks if required}
  #
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

In the above general form for a `PROPERTY SPECIFICATION FOR MATERIAL` command block, the string `model_name1` could be `ELASTIC` and the string `model_name2` could be `ORTHOTROPIC CRUSH`. Most `PROPERTY SPECIFICATION FOR MATERIAL` command blocks will have only one `PROPERTY SPECIFICATION FOR MATERIAL` command block.

Both `PROPERTY SPECIFICATION FOR MATERIAL` command blocks and `PARAMETERS FOR MODEL` command blocks are referenced by the element-block command block (also known as the `FINITE ELEMENT MODEL` command block), which is described in Section 5.1. It is up to you, the user, to select the material model that will be called for a particular material.

The material models that are most useful in Presto are described in Section 4.1.2 through Section 4.1.18. These models do not constitute the entire set of material models that are implemented in the SIERRA Framework. There are material models implemented in the SIERRA Framework that are useful for other solid mechanics codes, but not for Presto.

As indicated in the introductory material, not all of the material models available are applicable to all of the element types. As one example, there is a one-dimensional elastic material model that is used for a truss element but is not applicable to solid elements such as hexahedra or tetrahedra. For this particular example, the specific material-model usage is hidden from the user. If the user specifies a linear elastic material model for a truss, the one-dimensional elastic material model is used. If the user specifies a linear elastic material model for a hexahedron, a full three-dimensional elastic material model is used. As another example, the energy-dependent material models cannot be used for a one-dimensional element such as a truss. The energy-dependent material models can only be used for solid elements such as hexahedra and tetrahedra. (Chapter 5 has information to indicate what material models are available for which element models.)

For each material model, the parameters needed to describe that model are listed in the section pertinent to that particular model. Solid models with elastic constants require only two elastic constants. These two constants are then used to generate all of the elastic constants for the model. For example, if the user specifies Young's modulus and Poisson's ratio, then the shear modulus, bulk modulus, and lambda are calculated. If the shear modulus and lambda are specified, then Young's modulus, Poisson's ratio, and the bulk modulus are calculated.

The various nonlinear material models have state variables. Reference Section 8.5.2 to learn how to output the state variables for the nonlinear material models.

Only brief descriptions of the material models are presented in this chapter. For a detailed description of the various material models, you will need to consult a variety of references. Specific references are identified in Section 4.1.2 through Section 4.1.18 for most of the material models in Presto.

4.1.1 Thermal Strains

```
THERMAL STRAIN FUNCTION = <string>thermal_strain_function
# or all three of the following
THERMAL STRAIN X FUNCTION =
  <string>thermal_strain_x_function
THERMAL STRAIN Y FUNCTION =
  <string>thermal_strain_y_function
THERMAL STRAIN Z FUNCTION =
  <string>thermal_strain_z_function
```

It is possible to specify either an isotropic thermal-strain field using the command line `THERMAL STRAIN FUNCTION` or an orthotropic thermal-strain field using the command lines `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION`. For any of these command lines, the user supplies a thermal strain function (via a `DEFINITION FOR FUNCTION` command block), which defines the thermal strain as a function of temperature. The computed thermal strain is then subtracted from the strain passed to the material model.

A thermal strain can be applied to any two-dimensional or three-dimensional element, regardless of material type. For a three-dimensional element such as a hexahedron or tetrahedron, the thermal strains are applied to the strain in the global XYZ coordinate system. For the isotropic case, the thermal strains are the same in the X-direction, Y-direction, and Z-direction. For the anistropic case, the thermal strains can be different in each of the three global directions—X, Y, and Z. For a two-dimensional element, shell or membrane, the thermal strain corresponding to `THERMAL STRAIN X FUNCTION` is applied to the strain in the shell (or membrane) r-direction. (Reference Section 5.2.2 for a discussion of the shell rst coordinate system.) The thermal strain corresponding to `THERMAL STRAIN Y FUNCTION` is applied to the strain in the shell (or membrane) s-direction. For two-dimensional elements, the current implementation of orthotropic thermal strains is limited, for practical purposes, to special cases—flat sheets of uniform shell elements lying in one of the global planes, i.e., XY, YZ, or ZX. The current orthotropic thermal strain capability has limited use for shells and membranes in the current release of Presto. Tying the orthotropic thermal strain functionality to the shell orientation functionality (Section 5.2.2) in the future will provide much more useful orthotropic thermal-strain functionality for two-dimensional elements.

If an isotropic thermal-strain field is to be applied, the `THERMAL STRAIN FUNCTION` command line is placed in the root material scope in the material model, which is where the `DENSITY` command is located in some of the material models. The isotropic thermal strain is thus a general property of any material, not a property of any specific material formulation. The `THERMAL STRAIN FUNCTION` command line can go either before or after the `DENSITY` command line. The input value of `ther-`

`mal_strain_function` is the name of the function that is defined in a `DEFINITION FOR FUNCTION` command block containing the thermal strain and temperature values applicable to the specific model. For more information on how to set Presto to compute thermal strains and how to apply temperatures, see Section 4.2.

If an orthotropic thermal-strain field is to be applied, all three of the command lines `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION` are placed in the root material scope in the material model, which is where the `DENSITY` command is located in some of the material models. The orthotropic thermal strain is thus a general property of any material, not a property of any specific material formulation. The `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION` command lines can go either before or after the `DENSITY` command line. The input values of `thermal_strain_x_function`, `thermal_strain_y_function`, `thermal_strain_z_function` are the names of the functions that are defined in `DEFINITION FOR FUNCTION` command blocks containing the thermal strain and temperature values applicable to the specific model. For more information on how to set Presto to compute thermal strains and how to apply temperatures, see Section 4.2.

The `THERMAL STRAIN FUNCTION` command line and the `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION` command lines are not used for the elastic three-dimensional orthotropic model (Section 4.1.11) or the elastic laminate model (Section 4.1.18). See Section 4.2 for further information. Note that specification of a thermal strain is identified in the material model descriptions using the notation “{`thermal strain option`}.”

4.1.2 Elastic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ELASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
  END [PARAMETERS FOR MODEL ELASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

An elastic material model is used to describe simple linear elastic behavior of materials. This model is generally valid for small deformations.

For an elastic material, an elastic command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ELASTIC
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ELASTIC] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.

- The bulk modulus is defined with the `BULK MODULUS` command line.
- The shear modulus is defined with the `SHEAR MODULUS` command line.
- Lambda is defined with the `LAMBDA` command line.

For information about the elastic model, consult Reference 1.

4.1.3 Elastic Fracture Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ELASTIC_FRACTURE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    MAX STRESS = <real>max_stress
    CRITICAL STRAIN = <real>critical_strain
  END [PARAMETERS FOR MODEL ELASTIC_FRACTURE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

An elastic fracture material is a simple failure model based on linear elastic behavior. The model uses a maximum-principal-stress failure criterion. The stress decays isotropically based on the component of strain parallel to the maximum principal stress. The value of the component of strain over which the stress is decayed to zero is a user-defined parameter for the model. This strain parameter can be adjusted so that failure is mesh independent.

For an elastic material, an elastic command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ELASTIC_FRACTURE
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ELASTIC_FRACTURE] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.

- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The maximum principal stress at which failure occurs is defined with the `MAX STRESS command` line.
- The component of strain over which the stress decays to zero is defined with the `CRITICAL STRAIN` command line. This component of strain is aligned with the maximum-principal-stress direction at failure.

For information about the elastic fracture model, contact William Scherzinger at Sandia National Laboratories (Sandia) in Albuquerque, NM. His phone number is 505-284-4866, and his email address is wmscher@sandia.gov.

4.1.4 Elastic-Plastic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    HARDENING MODULUS = <real>hardening_modulus
    BETA = <real>beta_parameter(1.0)
  END [PARAMETERS FOR MODEL ELASTIC_PLASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The elastic-plastic linear hardening models are used to model materials, generally metals, undergoing plastic deformation at finite strains. Linear hardening generally refers to the shape of a uniaxial stress-strain curve where the stress increases linearly with the plastic, or permanent, strain. In a three-dimensional framework, hardening is the law that governs how the yield surface grows in stress space. If the yield surface grows uniformly in stress space, the hardening is referred to as isotropic hardening. When `BETA` is 1.0, we have only isotropic hardening.

Because the linear hardening model is relatively simple to integrate, there is also the ability to define a yield surface that not only grows, or hardens, but also moves in stress space. This is known as kinematic hardening. When `BETA` is 0.0, we have only kinematic hardening. The elastic-plastic linear hardening model allows for isotropic hardening, kinematic hardening, or a combination of the two.

For an elastic-plastic material, an elastic-plastic command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ELASTIC_PLASTIC] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The yield stress is defined with the `YIELD STRESS` command line.
- The hardening modulus is defined with the `HARDENING MODULUS` command line.
- The beta parameter is defined with the `BETA command` line.

For information about the elastic-plastic model, consult Reference 1.

4.1.5 Elastic-Plastic Power-Law Hardening Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL EP_POWER_HARD
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    HARDENING CONSTANT = <real>hardening_constant
    HARDENING EXPONENT = <real>hardening_exponent
    LUDERS STRAIN = <real>luders_strain
  END [PARAMETERS FOR MODEL EP_POWER_HARD]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

A power-law hardening model for elastic-plastic materials is used for modeling metal plasticity up to finite strains. The power-law hardening model, as opposed to the linear hardening model, has a power law fit for the uniaxial stress-strain curve that has the stress increase as the plastic strain raised to some power. The power-law hardening model also has the ability to model materials that exhibit Luder's strains after yield. Due to the more complicated yield behavior, the power-law hardening model can only be used with isotropic hardening.

For an elastic-plastic power-law hardening material, an elastic-plastic power-law hardening command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL EP_POWER_HARD
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL EP_POWER_HARD] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The yield stress is defined with the `YIELD STRESS` command line.
- The hardening constant is defined with the `HARDENING CONSTANT` command line.
- The hardening exponent is defined with the `HARDENING EXPONENT` command line.
- The Luder's strain is defined with the `LUDERS STRAIN` command line.

For information about the elastic-plastic power-law hardening model, consult Reference 1.

4.1.6 Elastic-Plastic Power-Law Hardening Model with Failure

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL DUCTILE_FRACTURE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    HARDENING CONSTANT = <real>hardening_constant
    HARDENING EXPONENT = <real>hardening_exponent
    LUDERS STRAIN <real>luders_strain
    CRITICAL TEARING PARAMETER = <real>crit_tearing
    CRITICAL CRACK OPENING STRAIN = <real>crit_crack
  END [PARAMETERS FOR MODEL DUCTILE_FRACTURE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

This model is identical to power-law hardening with the addition of a failure criterion and a postfailure isotropic decay of the stress to zero within the constitutive model. The point at which failure occurs is defined by a critical tearing parameter. The critical tearing parameter t_p is related to the plastic strain at failure ε_f by the evolution integral

$$t_p = \int_0^{\varepsilon_f} \langle \frac{2\sigma_{\max}}{3\left(\sigma_{\max} - \sigma_m\right)} \rangle^4 d\varepsilon_p \, . \tag{4.1}$$

In Equation (4.1), $\sigma_{\max}$ is the maximum principal stress, and σ_m is the mean stress. The quantity in the angle brackets, the expression

$$\frac{2\sigma_{\max}}{3\left(\sigma_{\max} - \sigma_m\right)}, \tag{4.2}$$

is nonzero only if it evaluates to a positive value. This quantity is set to zero if it has a negative value.

The stress decays isotropically based on the component of strain parallel to the maximum principal stress. The value of the component of strain over which the stress is decayed to zero is a user-defined parameter for the model. This strain parameter can be adjusted so that failure is mesh independent.

For an elastic-plastic power-law hardening material with failure, the command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL DUCTILE_FRACTURE
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL DUCTILE_FRACTURE] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The yield stress is defined with the `YIELD STRESS` command line.
- The hardening constant is defined with the `HARDENING CONSTANT` command line.
- The hardening exponent is defined with the `HARDENING EXPONENT` command line.
- The Luder's strain is defined with the `LUDERS STRAIN` command line.
- The critical tearing parameter is defined with the `CRITICAL TEARING PARAMETER` command line.

- The component of strain over which the stress decays to zero is defined with the `CRITICAL CRACK OPENING STRAIN` command line. This component of strain is aligned with the maximum-principal-stress direction at failure.

For information about the elastic-plastic power-law hardening model with failure, consult Reference 1.

4.1.7 Multilinear Elastic-Plastic Hardening Model with Failure

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ML_EP_FAIL
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    BETA = <real>beta_parameter(1.0)
    HARDENING FUNCTION = <real>hardening_function_name
    YOUNGS MODULUS FUNCTION = <real>ym_function_name
    POISSONS RATIO FUNCTION = <real>pr_function_name
    YIELD STRESS FUNCTION = <real>yield_stress_function_name
    CRITICAL TEARING PARAMETER = <real>crit_tearing
    CRITICAL CRACK OPENING STRAIN = <real>crit_crack
  END [PARAMETERS FOR MODEL ML_EP_FAIL]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

This model is similar to the power-law hardening model except the hardening behavior is described with a piecewise-linear curve as opposed to a power law. This model incorporates a failure criterion and a postfailure isotropic decay of the stress to zero within the constitutive model. The point at which failure occurs is defined by a critical tearing parameter. The critical tearing parameter t_p is related to the plastic strain at failure ε_f by the evolution integral

$$t_p = \int_0^{\varepsilon_f} \left\langle \frac{2\sigma_{\max}}{3\left(\sigma_{\max} - \sigma_m\right)} \right\rangle^4 d\varepsilon_p \, . \tag{4.3}$$

In Equation (4.3), $\sigma_{\max}$ is the maximum principal stress, and σ_m is the mean stress.

The quantity in the angle brackets, the expression

$$\frac{2\sigma_{\max}}{3\left(\sigma_{\max}-\sigma_m\right)}, \tag{4.4}$$

is nonzero only if it evaluates to a positive value. This quantity is set to zero if it has a negative value.

The stress decays isotropically based on the component of strain parallel to the maximum principal stress. The value of the component of strain over which the stress is decayed to zero is a user-defined parameter for the model. This strain parameter can be adjusted so that failure is mesh independent.

For a multilinear elastic-plastic hardening material with failure, the command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ML_EP_FAIL
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ML_EP_FAIL] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The yield stress is defined with the `YIELD STRESS` command line.
- The beta parameter is defined with the `BETA` command line.
- The `HARDENING FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the domain scope that describes the hardening behavior of the material as a stress versus equivalent plastic strain.

- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the domain scope that describes Young's modulus as a function of temperature.
- The `POISSONS RATIO FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the domain scope that describes Poisson's ratio as a function of temperature.
- The `YIELD STRESS FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the domain scope that describes the yield stress as a function of temperature.
- The critical tearing parameter is defined with the `CRITICAL TEARING PARAMETER` command line.
- The component of strain over which the stress decays to zero is defined with the `CRITICAL CRACK OPENING STRAIN` command line. This component of strain is aligned with the maximum-principal-stress direction at failure.

For information about the multilinear elastic-plastic hardening model with failure, contact William Scherzinger at Sandia National Laboratories (Sandia) in Albuquerque, NM. His phone number is 505-284-4866, and his email address is wmscher@sandia.gov.

4.1.8 BCJ Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL BCJ
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    C1 = <real>c1
    C2 = <real>c2
    C3 = <real>c3
    C4 = <real>c4
    C5 = <real>c5
    C6 = <real>c6
    C7 = <real>c7
    C8 = <real>c8
    C9 = <real>c9
    C10 = <real>c10
    C11 = <real>c11
    C12 = <real>c12
    C13 = <real>c13
    C14 = <real>c14
    C15 = <real>c15
    C16 = <real>c16
    C17 = <real>c17
    C18 = <real>c18
    C19 = <real>c19
    C20 = <real>c20
    DAMAGE EXPONENT = <real>damage_exponent
    INITIAL ALPHA_XX = <real>alpha_xx
    INITIAL ALPHA_YY = <real>alpha_yy
    INITIAL ALPHA_ZZ = <real>alpha_zz
```

```
        INITIAL ALPHA_XY = <real>alpha_xy
        INITIAL ALPHA_YZ = <real>alpha_yz
        INITIAL ALPHA_XZ = <real>alpha_xz
        INITIAL KAPPA = <real>initial_kappa
        INITIAL DAMAGE = <real>initial_damage
        YOUNGS MODULUS FUNCTION = <string>ym_function_name
        POISSONS RATIO FUNCTION = <string>pr_function_name
        SPECIFIC HEAT = <real>specific_heat
        THETA OPT = <integer>theta_opt
        FACTOR = <real>factor
        RHO = <real>rho
        TEMP0 = <real>temp0
      END [PARAMETERS FOR MODEL BCJ]
    END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The BCJ plasticity model is a state variable model for describing the finite deformation behavior of metals. It uses a multiplicative decomposition of the deformation gradient into elastic, volumetric plastic, and deviatoric parts. The model considers the natural configuration defined by this decomposition and its associated thermodynamics. The model incorporates strain rate and temperature sensitivity, as well as damage, through a yield-surface approach in which state variables follow a hardening-minus-recovery format.

Because the BCJ model has such an extensive list of parameters, we will not present the usual synopsis of parameter names with command lines. As with most other material models in Presto, the {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures. In addition, only two of the five elastic constants are required. The user should consult References 2, 3, and 4 for a description of the various parameters. The parameters `SPECIFIC HEAT`, `THETA OPT`, `FACTOR`, `RHO`, and `TEMP0` are recent additions to the parameter set for the BCJ model. These parameters were added to accommodate changes to the model for heat generation due to plastic dissipation. For coupled solid/thermal calculations, the plastic dissipation rate is stored as a state variable and passed to a thermal code as a heat source term. For uncoupled calculations, temperature is stored as a state variable and temperature increases due to plastic dissipation are calculated within the material model.

If temperature is provided from an external source, `THETA OPT` is set to 0. If the temperature is calculated by the BCJ model, `THETA OPT` is set to 1.

If you wish to know more about using the BCJ model, contact Michael L. Chiesa at Sandia in Livermore, CA. His phone number is 925-294-2103, and his email address is chiesa@sandia.gov.

4.1.9 Soil and Crushable Foam Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL SOIL_FOAM
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    A0 = <real>const_coeff_yieldsurf
    A1 = <real>lin_coeff_yieldsurf
    A2 = <real>quad_coeff_yieldsurf
    PRESSURE CUTOFF = <real>pressure_cutoff
    PRESSURE FUNCTION = <string>function_press_volstrain
  END [PARAMETERS FOR MODEL SOIL_FOAM]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The soil and crushable foam model is a plasticity model that can be used for modeling soil or crushable foam. Given the right input, the model is a Drucker-Prager model.

For the soil and crushable foam model, the yield surface is a surface of revolution about the hydrostat in principal stress space. A planar end cap is assumed for the yield surface so that the yield surface is closed. The yield stress, σ_{yd}, is specified as a polynomial in pressure, p. The yield stress is given as

$$\sigma_{yd} = a_0 + a_1 p + a_2 p^2 \, , \tag{4.5}$$

where p is positive in compression. The determination of the yield stress from Equation (4.5) places severe restrictions on the admissible values of a_0, a_1, and a_2. There are three valid cases for the yield surface. In the first case, there is an elastic–perfectly plastic deviatoric response, and the yield surface is a cylinder oriented along the hydrostat in principal stress space. In this case, a_0 is positive, and a_1 and a_2 are zero.

In the second case, the yield surface is conical. A conical yield surface is obtained by setting a_2 to zero and using appropriate values for a_0 and a_1. In the third case, the yield surface has a parabolic shape. For the parabolic yield surface, all three of the coefficients in Equation (4.5) are nonzero. The coefficients are checked to determine that a valid negative tensile-failure pressure can be derived based on the specified coefficients.

For the case of the cylindrical yield surface (e.g., $a_0 > 0$ and $a_1 = a_2 = 0$), there is no tensile-failure pressure. For the other two cases, the computed tensile-failure pressure may be too low. To handle the situations where there is no tensile-failure pressure or the tensile-failure pressure is too low, a pressure cutoff can be defined. If a pressure cutoff is defined, the tensile-failure pressure is the larger of the computed tensile-failure pressure and the defined cutoff pressure.

The plasticity theories for the volumetric and deviatoric parts of the material response are completely uncoupled. The volumetric response is computed first. The mean pressure p is assumed to be positive in compression, and a yield function ϕ_p is written for the volumetric response as

$$\phi_p = p - f_p\left(\varepsilon_V\right) , \tag{4.6}$$

where $f_p\left(\varepsilon_V\right)$ defines the volumetric stress-strain curve for the pressure. The yield function ϕ_p determines the motion of the end cap along the hydrostat.

For a soil and crushable foam material, a soil and crushable foam command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL SOIL_FOAM
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL SOIL_FOAM] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.

  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.

- The constant in the equation for the yield surface is defined with the `A0` command line.

- The coefficient for the linear term in the equation for the yield surface is defined with the `A1` command line.

- The coefficient for the quadratic term in the equation for the yield surface is defined with the `A2` command line.

- The user-defined tensile-failure pressure is defined with the `PRESSURE CUTOFF` command line.

- The pressure as a function of volumetric strain is defined with the function named on the `PRESSURE FUNCTION` command line.

For information about the soil and crushable foam model, consult with the Pronto3d document listed as Reference 5. The soil and crushable foam model in Presto is the same as the soil and crushable foam model in Pronto3d. The Pronto3d model is based on a material model developed by Krieg [6]. The Krieg version of the soil and crushable foam model was later modified by Swenson and Taylor [7]. The soil and crushable foam model developed by Swenson and Taylor is the model in both Pronto3d and Presto.

4.1.10 Foam Plasticity Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL FOAM_PLASTICITY
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    PHI = <real>phi
    SHEAR STRENGTH  = <real>shear_strength
    SHEAR HARDENING = <real>shear_hardening
    SHEAR EXPONENT  = <real>shear_exponent
    HYDRO STRENGTH  = <real>hydro_strength
    HYDRO HARDENING = <real>hydro_hardening
    HYDRO EXPONENT  = <real>hydro_exponent
    BETA = <real>beta
  END [PARAMETERS FOR MODEL FOAM_PLASTICITY]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The foam plasticity model was developed to describe the response of porous elastic-plastic materials like closed-cell polyurethane foam to large deformation. Like solid metals, these foams can exhibit significant plastic deviatoric strains (permanent shape changes). Unlike metals, these foams can also exhibit significant plastic volume strains (permanent volume changes). The foam plasticity model is characterized by an initial yield surface that is an ellipsoid about the hydrostat.

When foams are compressed, they typically exhibit an initial elastic regime followed by a plateau regime in which the stress needed to compress the foam remains nearly constant. At some point in the compression process the densification regime is reached, and the stress needed to compress the foam further begins to rapidly increase.

The foam plasticity model can be used to describe the response of metal foams and many closed-cell, polymeric foams to large deformation (including polyurethane, polystyrene bead, etc.). This model is not appropriate for flexible foams that return to their undeformed shape after loads are removed.

For a foam plasticity material, a foam plasticity command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL FOAM_PLASTICITY
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL FOAM_PLASTICITY] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio is defined with the `POISSONS RATIO` command line.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The initial volume fraction of solid material in the foam, φ, is defined with the `PHI` command line. For example, solid polyurethane weighs 75 pounds per cubic foot (pcf); uncompressed 10 pcf polyurethane foam would have a φ of $0.133 = 10/75$.
- The shear (deviatoric) strength of uncompressed foam is defined with the `SHEAR STRENGTH` command line.
- The shear hardening modulus for the foam is defined with the `SHEAR HARDENING` command line.
- The shear hardening exponent is defined with the `SHEAR EXPONENT` command line. The deviatoric strength is given by `(SHEAR STRENGTH)` + `(SHEAR HARDENING) * PHI**(SHEAR EXPONENT)`.
- The hydrostatic (volumetric) strength of the uncompressed foam is defined with the `HYDRO STRENGTH` command line.

- The hydrodynamic hardening modulus for the foam is defined with the `HYDRO HARDENING` command line.

- The hydrodynamic hardening exponent for the foam is defined with the `HYDRO EXPONENT` command line. The hydrostatic strength is given by `(HYDRO STRENGTH)` + `(HYDRO HARDENING) * PHI**(HYDRO EXPONENT)`.

- The prescription for nonassociated flow, β, is defined with the `BETA` command line. When $\beta = 0.0$, the flow direction is given by the normal to the yield surface (associated flow). When $\beta = 1.0$, the flow direction is given by the stress tensor. Values of between 0.0 and 0.95 are recommended.

For information about the foam plasticity model, contact William Scherzinger at Sandia National Laboratories (Sandia) in Albuquerque, NM. His phone number is 505-284-4866, and his email address is wmscher@sandia.gov.

4.1.11 Elastic Three-Dimensional Orthotropic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
    YOUNGS MODULUS AA = <real>Eaa_value
    YOUNGS MODULUS BB = <real>Ebb_value
    YOUNGS MODULUS CC = <real>Ecc_value
    POISSONS RATIO AB = <real>NUab_value
    POISSONS RATIO BC = <real>NUbc_value
    POISSONS RATIO CA = <real>NUca_value
    SHEAR MODULUS AB = <real>Gab_value
    SHEAR MODULUS BC = <real>Gbc_value
    SHEAR MODULUS CA = <real>Gca_value
    COORDINATE SYSTEM = <string>coordinate_system_name
    DIRECTION FOR ROTATION = <real>1|2|3
    ALPHA = <real>alpha_in_degrees
    SECOND DIRECTION FOR ROTATION = <real>1|2|3
    SECOND ALPHA = <real>second_alpha_in_degrees
    THERMAL STRAIN AA FUNCTION = <string>ethaa_function_name
    THERMAL STRAIN BB FUNCTION = <string>ethbb_function_name
    THERMAL STRAIN CC FUNCTION = <string>ethcc_function_name
  END [PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The elastic three-dimensional orthotropic model describes the linear elastic response of an orthotropic material where the orientation of the principal material directions can be arbitrary. These principal axes are here denoted as A, B, and C. Thermal strains are also given along the principal material axes. The specification of these material axes is accomplished by selecting a user-defined coordinate system that can then be rotated twice about one of its current axes to give the final desired directions.

For an elastic three-dimensional orthotropic model, the command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC] .
```

In the above command blocks:

- The density is defined with the `DENSITY` command line.
- The Youngs moduli corresponding to the principal material axes A, B, and

C are given by the `YOUNGS MODULUS AA`, `YOUNGS MODULUS BB`, and `YOUNGS MODULUS CC` command lines.

- The Poisson's ratio defining the BB normal strain when the material is subjected only to AA normal stress is given by the `POISSONS RATIO AB` command line.
- The Poisson's ratio defining the CC normal strain when the material is subjected only to BB normal stress is given by the `POISSONS RATIO BC` command line.
- The Poisson's ratio defining the AA normal strain when the material is subjected only to CC normal stress is given by the `POISSONS RATIO CA` command line.
- The shear moduli for shear in the AB, BC, and CA planes are given by the `SHEAR MODULUS AB`, `SHEAR MODULUS BC`, and `SHEAR MODULUS CA` command lines, respectively.
- The specification of the principal material directions begins with the selection of a user-specified coordinate system given by the `COORDINATE SYSTEM` command line. This initial coordinate system can then be rotated twice to give the final material directions.
- The rotation of the initial coordinate system is defined with the `DIRECTION FOR ROTATION` and `ALPHA` command lines. The axis for rotation of the initial coordinate system is specified by the `DIRECTION FOR ROTATION` command line, while the angle of rotation is given by the `ALPHA` command line. This gives an intermediate specification of the material directions.
- The rotation of the intermediate coordinate system is defined with the `SECOND DIRECTION FOR ROTATION` and `SECOND ALPHA` command lines. The axis for rotation of the intermediate coordinate system is specified by the `SECOND DIRECTION FOR ROTATION` command line, while the angle of rotation is given by the `SECOND ALPHA` command line. The resulting coordinate system gives the final specification of the material directions.
- The thermal strain functions for normal thermal strains along the principal material directions are given by the `THERMAL STRAIN AA FUNCTION`, `THERMAL STRAIN BB FUNCTION`, and `THERMAL STRAIN CC FUNCTION` command lines.

See Reference 8 for more information about the elastic three-dimensional orthotropic model.

4.1.12 Orthotropic Crush Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    EX  = <real>modulus_x
    EY  = <real>modulus_y
    EZ  = <real>modulus_z
    GXY = <real>shear_modulus_xy
    GYZ = <real>shear_modulus_yz
    GZX = <real>shear_modulus_zx
    VMIN = <real>min_crush_volume
    CRUSH XX = <string>stress_volume_xx_function_name
    CRUSH YY = <string>stress_volume_yy_function_name
    CRUSH ZZ = <string>stress_volume_zz_function_name
    CRUSH XY = <string>shear_stress_volume_xy_function_name
    CRUSH YZ = <string>shear_stress_volume_yz_function_name
    CRUSH ZX = <string>shear_stress_volume_zx_function_name
  END [PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The orthotropic crush model is an empirically based constitutive relation that is useful for modeling materials like metallic honeycomb and wood. This particular implementation follows the formulation of the metallic honeycomb model in DYNA3D [9]. The orthotropic crush model divides material behavior into three phases:

- orthotropic elastic,

- volumetric crush (partially compacted), and
- elastic–perfectly plastic (fully compacted).

For an orthotropic crush material, an orthotropic crush command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH] .
```

In the above command blocks:

- The uncompacted density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- Only two of the following elastic constants are required:
  - Young's modulus for the fully compacted state is defined with the `YOUNGS MODULUS` command line. This is the elastic–perfectly plastic value of Young's modulus.
  - Poisson's ratio for the fully compacted state is defined with the `POISSONS RATIO` command line. This is the elastic–perfectly plastic value of Poisson's ratio.
  - The bulk modulus is defined with the `BULK MODULUS` command line.
  - The shear modulus is defined with the `SHEAR MODULUS` command line.
  - Lambda is defined with the `LAMBDA` command line.
- The yield stress for the fully compacted state is defined with the `YIELD STRESS` command line. This is the elastic–perfectly plastic value of the yield stress.
- The initial directional modulus E_{xx} is defined with the `EX` command line.
- The initial directional modulus E_{yy} is defined with the `EY` command line.
- The initial directional modulus E_{zz} is defined with the `EZ` command line.
- The initial directional shear modulus G_{xy} is defined with the `GXY` command line.
- The initial directional shear modulus G_{yz} is defined with the `GYZ` command line.

- The initial directional shear modulus G_{zx} is defined with the `GZX` command line.
- The minimum crush volume as a fraction of the original volume is defined with the `VMIN` command line.
- The directional stress σ_{xx} as a function of the volume crush is defined by the function referenced in the `CRUSH XX` command line.
- The directional stress σ_{yy} as a function of the volume crush is defined by the function referenced in the CRUSH `YY` command line.
- The directional stress σ_{zz} as a function of the volume crush is defined by the function referenced in the `CRUSH ZZ` command line.
- The directional stress σ_{xy} as a function of the volume crush is defined by the function referenced in the `CRUSH XY` command line.
- The directional stress σ_{yz} as a function of the volume crush is defined by the function referenced in the `CRUSH YZ` command line.
- The directional stress σ_{zx} as a function of the volume crush is defined by the function referenced in the `CRUSH ZX` command line.

Note that several of the command lines in this command block (those beginning with `CRUSH`) reference functions. These functions must be defined in the domain scope. For information about the orthotropic crush model, consult Reference 9.

4.1.13 Orthotropic Rate Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_RATE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    MODULUS TTTT = <real>modulus_tttt
    MODULUS TTLL = <real>modulus_ttll
    MODULUS TTWW = <real>modulus_ttww
    MODULUS LLLL = <real>modulus_llll
    MODULUS LLWW = <real>modulus_llww
    MODULUS WWWW = <real>modulus_wwww
    MODULUS TLTL = <real>modulus_tltl
    MODULUS LWLW = <real>modulus_lwlw
    MODULUS WTWT = <real>modulus_wtwt
    TX = <real>tx
    TY = <real>ty
    TZ = <real>tz
    LX = <real>lx
    LY = <real>ly
    LZ = <real>lz
    MODULUS FUNCTION = <string>modulus_function_name
    RATE FUNCTION = <string>rate_function_name
    T FUNCTION = <string>t_function_name
    L FUNCTION = <string>l_function_name
    W FUNCTION = <string>w_function_name
    TL FUNCTION = <string>tl_function_name
    LW FUNCTION = <string>lw_function_name
    WT FUNCTION = <string>wt_function_name
  END [PARAMETERS FOR MODEL ORTHOTROPIC_RATE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

Orthotropic rate is a new and improved version of the orthotropic crush constitutive model. This model has been developed to describe the behavior of an aluminum honeycomb subjected to large deformation. The new orthotropic rate model, like the original orthotropic crush model, has six independent yield functions that evolve with volume strain. Unlike the original model, this new model has yield functions that also depend on strain rate. The new model also uses an orthotropic elasticity tensor with nine elastic moduli in place of the orthotropic elasticity tensor with six elastic moduli used in the original orthotropic crush model. A new honeycomb orientation capability has also been added that allows users to prescribe initial honeycomb orientations that are not aligned with the original global coordinate system.

For an orthotropic rate material, an orthotropic rate command block starts with the input line

```
BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_RATE
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ORTHOTROPIC_RATE] .
```

In the above command blocks:

- Density is defined with the `DENSITY` command line.
- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- In the following list of elastic constants, only the elastic modulus (Young's modulus) is required for this model. If two elastic constants are supplied, the elastic constants will be completed. However, only the elastic modulus is used in this model.
  - Young's modulus for the fully compacted honeycomb is defined with the `YOUNGS MODULUS` command line.
  - Poisson's ratio for the fully compacted state is defined with the `POISSONS RATIO` command line.
  - The bulk modulus for the fully compacted state is defined with the `BULK MODULUS` command line.
  - The shear modulus for the fully compacted state is defined with the `SHEAR MODULUS` command line.

  - Lambda for the fully compacted state is defined with the `LAMBDA` command line.

- The yield stress for the fully compacted honeycomb is defined with the `YIELD STRESS` command line.

- The nine elastic moduli for the orthotropic uncompacted honeycomb are defined with the `MODULUS TTT`, `MODULUS TTLL`, `MODULUS TTWW`, `MODULUS LLLL`, `MODULUS LLWW`, `MODULUS WWWW`, `MODULUS TLTL`, `MODULUS LWLW`, and `MODULUS WTWT` command lines. The T-direction is usually associated with the generator axis for the honeycomb. The L-direction is in the ribbon plane (plane defined by flat sheets used in reinforced honeycomb) and orthogonal to the generator axis. The W-direction is perpendicular to the ribbon plane.

- The components of a vector defining the T-direction of the honeycomb are defined by the `TX`, `TY`, and `TZ` command lines. The values `tx`, `ty`, and `tz` are components of a vector in the global coordinate system that define the orientation of the honeycomb's T-direction (generator axis).

- The components of a vector defining the L-direction of the honeycomb are defined by the `LX`, `LY`, and `LZ` command lines. The values lx, ly, and lz are components of a vector in the global coordinate system that define the orientation of the honeycomb's L-direction. *Caution*: The vectors T and L must be orthogonal.

- The function describing the variation in moduli with compaction is given by the `MODULUS FUNCTION` command line. The moduli vary continuously from their initial orthotropic values to isotropic values when full compaction is obtained.

- The function describing the change in strength with strain rate is given by the `RATE FUNCTION` command line. Note that all strengths are scaled with the multiplier obtained from this function.

- The function describing the T-normal strength of the honeycomb as a function of compaction is given by the `T FUNCTION` command line.

- The function describing the L-normal strength of the honeycomb as a function of compaction is given by the `L FUNCTION` command line.

- The function describing the W-normal strength of the honeycomb as a function of compaction is given by the `W FUNCTION` command line.

- The function describing the TL-normal strength of the honeycomb as a function of compaction is given by the `TL FUNCTION` command line.

- The function describing the LW-normal strength of the honeycomb as a function of compaction is given by the `LW FUNCTION` command line.
- The function describing the WT-normal strength of the honeycomb as a function of compaction is given by the `WT FUNCTION` command line.

Note that several of the command lines in this command block reference functions. These functions must be defined in the domain scope. For information about the orthotropic rate model, contact William Scherzinger at Sandia National Laboratories (Sandia) in Albuquerque, NM. His phone number is 505-284-4866, and his email address is wmscher@sandia.gov.

4.1.14 Mie-Gruneisen Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL MIE_GRUNEISEN
    RHO_0 = <real>density
    C_0 = <real>sound_speed
    SHUG = <real>const_shock_velocity
    GAMMA_0 = <real>ambient_gruneisen_param
    POISSR = <real>poissons_ratio
    Y_0 = <real>yield_strength
    PMIN = <real>mean_stress(REAL_MAX)
  END [PARAMETERS FOR MODEL MIE_GRUNEISEN]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Mie-Gruneisen material model describes the nonlinear pressure-volume (or equivalently pressure-density) response of solids or fluids in terms of a reference pressure-volume curve and deviations from the reference curve in energy space. The reference curve is taken to be the experimentally determined principal Hugoniot, which is the locus of end states that can be reached by a shock transition from the ambient state. For details about this model, see Reference 10.

For Mie-Gruneisen energy-dependent materials, the Mie-Gruneisen command block begins with the input line

```
BEGIN PARAMETERS FOR MODEL MIE_GRUNEISEN
```

and is terminated with the input line

```
END [PARAMETERS FOR MODEL MIE_GRUNEISEN] .
```

In the above command blocks:

- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.

- The ambient density, ρ_0, is defined with the `RHO_0` command line. The ambient density is the density at which the mean pressure is zero, not necessarily the initial density.

- The ambient bulk sound speed, c_0, is defined by the `C_0` command line. The ambient bulk sound speed is also the first constant in the shock-velocity-versus-particle-velocity relation $D = c_0 + Su$, where u is the particle velocity. (See the following description of the `SHUG` command line for the definition of S.)

- The second constant in the shock-velocity-versus-particle-velocity equation, S, is defined by the `SHUG` command line. The shock-velocity-versus-particle-velocity relation is $D = c_0 + Su$, where u is the particle velocity. (See the previous description of the `C_0` command line for the definition of c_0.)

- The ambient gruneisen parameter, Γ_0, is defined by the `GAMMA_0` command line.

- Poisson's ratio, ν, is defined by the `POISSR` command line. Poisson's ratio is assumed constant.

- The yield strength, y_0, is defined by the `Y_0` command line. The yield strength is zero for the hydrodynamic case.

- The fracture stress is defined by the `PMIN` command line. The fracture stress is a mean stress or pressure, so it must be negative or zero. This is an optional parameter; if not specified, the parameter defaults to `REAL_MAX` (no fracture).

For information about the Mie-Gruneisen model, consult Reference 10.

4.1.15 Mie-Gruneisen Power-Series Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL MIE_GRUNEISEN_POWER_SERIES
    RHO_0 = <real>density
    C_0 = <real>sound_speed
    K1 = <real>power_series_coeff1
    K2 = <real>power_series_coeff2
    K3 = <real>power_series_coeff3
    K4 = <real>power_series_coeff4
    K5 = <real>power_series_coeff5
    GAMMA_0 = <real>ambient_gruneisen_param
    POISSR = <real>poissons_ratio
    Y_0 = <real>yield strength
    PMIN = <real>mean_stress(REAL_MAX)
  END [PARAMETERS FOR MODEL MIE_GRUNEISEN_POWER_SERIES]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Mie-Gruneisen power-series model describes the nonlinear pressure-volume (or equivalently pressure-density) response of solids or fluids in terms of a reference pressure-volume curve and deviations from the reference curve in energy space. The reference curve is taken to be the experimentally determined principal Hugoniot, which is the locus of end states that can be reached by a shock transition from the ambient state. The Mie-Gruneisen power-series model is very similar to the Mie-Gruneisen model, except that the Mie-Gruneisen model bases the Hugoniot pressure-volume response on the assumption of a linear shock-velocity-versus-particle-velocity relation, while the Mie-Gruneisen power-series model uses a power-series expression. For details about this model, see Reference 10.

For Mie-Gruneisen power-series energy-dependent materials, the Mie-Gruneisen power-series command block begins with the input line

```
BEGIN PARAMETERS FOR MODEL MIE_GRUNEISEN_POWER_SERIES
```

and is terminated with the input line

```
  END [PARAMETERS FOR MODEL MIE_GRUNEISEN_POWER_SERIES] .
```

In the above command blocks:

- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- The ambient density, ρ_0, is defined with the `RHO_0` command line. The ambient density is the density at which the mean pressure is zero, not necessarily the initial density.
- The ambient bulk sound speed, c_0, is defined by the `C_0` command line.
- The power-series coefficients k_1, k_2, k_3, k_4, and k_5 are defined by the command lines `K1`, `K2`, `K3`, `K4`, and `K5`, respectively. Only the nonzero power-series coefficients need be input, since coefficients not specified will default to zero.
- The ambient gruneisen parameter, Γ_0, is defined by the `GAMMA_0` command line.
- Poisson's ratio, ν, is defined by the `POISSR` command line. Poisson's ratio is assumed constant.
- The yield strength, y_0, is defined by the `Y_0` command line. The yield strength is zero for the hydrodynamic case.
- The fracture stress is defined by the `PMIN` command line. The fracture stress is a mean stress or pressure, so it must be negative or zero. This is an optional parameter; if not specified, the parameter defaults to `REAL_MAX` (no fracture).

For information about the Mie-Gruneisen power-series model, consult Reference 10.

4.1.16 JWL (Jones-Wilkins-Lee) Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL JWL
    RHO_0 = <real>initial_density
    D = <real>detonation_velocity
    E_0 = <real>init_chem_energy
    A = <real>jwl_const_pressure1
    B = <real>jwl_const_pressure2
    R1 = <real>jwl_const_nondim1
    R2 = <real>jwl_const_nondim2
    OMEGA = <real>jwl_const_nondim3
    XDET = <real>x_detonation_point
    YDET = <real>y_detonation_point
    ZDET = <real>z_detonation_point
    TDET = <real>time_of_detonation
    B5 = <real>burn_width_const(2.5)
  END [PARAMETERS FOR MODEL JWL]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The JWL model describes the pressure-volume-energy response of the gaseous detonation products of HE (High Explosive). For details about this model, see Reference 10.

For JWL energy-dependent materials, the `JWL` command block begins with the input line

```
BEGIN PARAMETERS FOR MODEL JWL
```

and is terminated with the input line

```
END [PARAMETERS FOR MODEL JWL] .
```

In the above command blocks:

- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.

- The initial density of the unburned explosive, ρ_0, is given by the `RHO_0` command line.
- The detonation velocity, D, is given by the `D` command line.
- The initial chemical energy per unit mass in the explosive, E_0, is given by the `E_0` command line. Most compilations of JWL parameters give E_0 in units of energy per unit volume, rather than energy per unit mass. Thus, the tabulated value must be divided by ρ_0, the initial density of the unburned explosive.
- The JWL constants with units of pressure, A and B, are given by the `A` and `B` command lines, respectively.
- The dimensionless JWL constants, R_1, R_2, and ω, are given by the `R1`, `R2`, and `OMEGA` command lines, respectively.
- The x-coordinate of the detonation point, x_D, is given by the `XDET` command line.
- The y-coordinate of the detonation point, y_D, is given by the `YDET command` line.
- The z-coordinate of the detonation point, z_D, is given by the `ZDET` command line.
- The time of detonation, t_D, is given by the `TDET` command line.
- The burn-width constant, B_5, is given by the `B5` command line. The burn-width constant has a default value of 2.5.

For information about the JWL model, consult Reference 10.

4.1.17 Ideal Gas Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL IDEAL_GAS
    RHO_0 = <real>initial_density
    C_0 = <real>initial_sound_speed
    GAMMA = <real>ratio_specific_heats
  END [PARAMETERS FOR MODEL IDEAL_GAS]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The ideal gas model provides a material description based on the ideal gas law. For details about this model, see Reference 10.

For ideal gas materials, the ideal gas command block begins with the input line

```
BEGIN PARAMETERS FOR MODEL IDEAL_GAS
```

and is terminated with the input line

`END [PARAMETERS FOR MODEL IDEAL_GAS]` .

In the above command blocks:

- The {`thermal strain option`} is used to define thermal strains. See Section 4.1.1 and Section 4.2 for further information on specifying and applying thermal strains and temperatures.
- The initial density, ρ_0, is given by the `RHO_0` command line.
- The initial sound speed, c_0, is given by the `C_0` command line.
- The ratio of specific heats, γ, is given by the `GAMMA` command line.

For information about the ideal gas model, consult Reference 10.

4.1.18 Elastic Laminate Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BEGIN PARAMETERS FOR MODEL ELASTIC_LAMINATE
    A11 = <real>a11_value
    A12 = <real>a12_value
    A16 = <real>a16_value
    A22 = <real>a22_value
    A26 = <real>a26_value
    A66 = <real>a66_value
    A44 = <real>a44_value
    A45 = <real>a45_value
    A55 = <real>a55_value
    B11 = <real>b11_value
    B12 = <real>b12_value
    B16 = <real>b16_value
    B22 = <real>b22_value
    B26 = <real>b26_value
    B66 = <real>b66_value
    D11 = <real>d11_value
    D12 = <real>d12_value
    D16 = <real>d16_value
    D22 = <real>d22_value
    D26 = <real>d26_value
    D66 = <real>d66_value
    COORDINATE SYSTEM = <string>coord_sys_name
    DIRECTION FOR ROTATION = 1|2|3
    ALPHA = <real>alpha_value_in_degrees
    THETA = <real>theta_value_in_degrees
    NTH11 FUNCTION = <string>nth11_function_name
    NTH22 FUNCTION = <string>nth22_function_name
    NTH12 FUNCTION = <string>nth12_function_name
    MTH11 FUNCTION = <string>mth11_function_name
    MTH22 FUNCTION = <string>mth22_function_name
    MTH12 FUNCTION = <string>mth12_function_name
  END [PARAMETERS FOR MODEL ELASTIC_LAMINATE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The elastic laminate model can be used to describe the overall linear elastic response of layered shells. The response of each layer is pre-integrated through the thickness under an assumed variation of strain through the thickness. That is, the user inputs laminate stiffness matrices directly, and the overall response is calculated appropriately. This model allows the user to input laminate stiffness matrices that are consistent with a state of generalized plane stress for each layer. Each layer can be orthotropic with a unique orientation. This model is primarily intended for capturing the response of fiber-reinforced laminated composites. The user inputs the laminate stiffness matrices calculated with respect to a chosen coordinate system and then specifies this coordinate system's definition relative to the global coordinate system. Thermal stresses are handled via the input of thermal-force and thermal-force-couple resultants for the laminate as a whole. At present, the user cannot get layer stresses out from this material model. However, the overall section-force and force-couple resultants can be computed from available output. The details of this model are described in References 11 and 12.

For elastic laminate materials, the elastic laminate command block begins with the input line

```
BEGIN PARAMETERS FOR MODEL ELASTIC_LAMINATE
```

and terminates with the input line

```
END [PARAMETERS FOR MODEL ELASTIC_LAMINATE] .
```

In the above command blocks:

- The density is defined with the `DENSITY` command line.
- The extensional stiffnesses are defined with the $\mathtt{A}_{ij}$ command lines, where the values of ij are 11, 12, 16, 22, 26, 66, 44, 45, and 55.
- The coupling stiffnesses are defined with the $\mathtt{B}_{ij}$ command lines, where the values of ij are 11, 12, 16, 22, 26, and 66.
- The bending stiffnesses are defined with the $\mathtt{D}_{ij}$ command lines, where the values of ij are 11, 12, 16, 22, 26, and 66.
- The initial laminate coordinate system is defined with the `COORDINATE SYSTEM` command line.
- The rotation of the initial laminate coordinate system is defined with the `DIRECTION FOR ROTATION` and `ALPHA` command lines. The axis of initial laminate coordinate system is specified by the `DIRECTION FOR ROTATION` command line, while the angle of rotation is given by the `ALPHA` command line. This produces an intermediate laminate coordinate system that is then projected onto the surface of each shell element.
- The projected intermediate laminate coordinate system is rotated about the element normal by angle theta, which is specified by the `THETA` command line.
- The thermal-force resultants are defined with functions named on the `NTH11 FUNCTION`, `NTH22 FUNCTION`, and `NTH12 FUNCTION` command lines.

- The thermal-force-couple resultants are defined with functions named on the `MTH11 FUNCTION`, `MTH22 FUNCTION`, and `MTH12 FUNCTION` command lines.

4.2 Applying Temperatures and Thermal Strains

Presto has the capability to compute thermal strains on three-dimensional continuum and two-dimensional (shell, membrane) elements. Three things are required to activate thermal strains:

- First, one or more thermal strain functions (strain as a function of temperature) must be defined. Each thermal strain function is defined with a `DEFINITION FOR FUNCTION` command block. (This function is the standard function definition that appears in the domain scope.) The thermal strain function gives the total thermal strain associated with a given temperature. It is the change in thermal strain with the change in temperature that gives rise to thermal stresses in a body.
- Second, each element block with either isotropic or orthotropic thermal strain behavior must reference a material model command block with thermal strain command lines defined in Section 4.1.1. If an element block exhibits isotropic thermal strain behavior, the block must reference a material model with a `THERMAL STRAIN FUNCTION` command line. (This `THERMAL STRAIN FUNCTION` command line must reference some function name defined with a `DEFINITION FOR FUNCTION` command block in the domain scope.) If an element block exhibits orthotropic strain behavior, the block must reference a material model with the `THERMAL STRAIN FUNCTION X`, `THERMAL STRAIN FUNCTION Y`, and `THERMAL STRAIN FUNCTION Z` command lines. (These three command lines must reference function names defined with `DEFINITION FOR FUNCTION` command blocks in the domain scope.)
- Third, a temperature field must be applied to the Presto region. The command block to specify the application of temperatures is `PRESCRIBED TEMPERATURE`, which is implemented as a standard boundary condition. You should consult with Chapter 6 for a description of the `PRESCRIBED TEMPERATURE` command block.

Temperature is applied to the nodes. For elements, the nodal values are averaged (depending on element) connectivity to produce an element temperature. At the element level, two element temperatures and the current time step are used to compute a thermal strain rate. The difference between the thermal strain associated with the temperature at the current time and the thermal strain associated with the temperature at the current time plus the current time step divided by the current time step is the thermal strain rate.

You will not want to use the `THERMAL STRAIN FUNCTION` command line or the `THERMAL STRAIN FUNCTION X`, `THERMAL STRAIN FUNCTION Y`, and `THERMAL`

`STRAIN FUNCTION Z` command lines with the elastic three-dimensional orthotropic model. This model has command lines that define thermal strains in the material direction. See Section 4.1.11 regarding the definition of thermal strains for the elastic three-dimensional orthotropic model. You will need function definitions and a temperature field to apply thermal strains to this model.

The elastic laminate does not use thermal strains. Therefore, you will not want to use the `THERMAL STRAIN FUNCTION` command line or the `THERMAL STRAIN FUNCTION X`, `THERMAL STRAIN FUNCTION Y`, and `THERMAL STRAIN FUNCTION Z` command lines with the elastic laminate model. The elastic laminate model uses thermal-force and thermal-force-couple resultants. The thermal-force and thermal-force-couple resultants are defined with functions. You will need these function definitions and a temperature field to apply the thermal force and thermal force-couple resultants for an elastic laminate model. See Section 4.1.18 for a description of the elastic laminate model.

4.3 Energy Deposition

```
BEGIN PRESCRIBED ENERGY DEPOSITION
  # {block set commands}
  BLOCK = <string_list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK
  #
  # function commands
  T FUNCTION = <string>t_func_name
  X FUNCTION = <string>x_func_name
  Y FUNCTION = <string>y_func_name
  Z FUNCTION = <string>z_func_name
  #
  # input mesh command
  READ VARIABLE = <string>mesh_var_name
  #
  # user subroutine commands
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  # {other user subroutine command lines}
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
END [PRESCRIBED ENERGY DEPOSITION]
```

The `PRESCRIBED ENERGY DEPOSITION` command block applies a set quantity of energy to energy-dependent material models for a given set of element blocks. Energy deposition represents a particular type of boundary condition, and thus this command block follows the general specification of command blocks used to specify boundary conditions in Chapter 6. The `PRESCRIBED ENERGY DEPOSITION` command block must appear in the region scope.

There are three options for defining the energy deposition for a set of elements: with standard SIERRA functions, with a mesh variable in the input mesh file, and by a user subroutine. If the energy deposition is a reasonably simple description and can be defined using the standard SIERRA functions, the function option is recommended. If the energy deposition requires a more complex description, it is necessary to use either the input mesh option or the user subroutine option. Only one of the three options can be specified in the command block.

The `PRESCRIBED ENERGY DEPOSITION` command block contains four groups of commands—block set, function, input mesh, and user subroutine. Each of these command groups, with the exception of the `T FUNCTION` command line, is basically independent of the others. Following are descriptions of the different command groups.

4.3.1 Block Set Commands

The {`block set commands`} portion of the `PRESCRIBED ENERGY DEPOSITION` command block defines a set of element blocks associated with the prescribed energy deposition and can include some combination of the following command lines:

```
BLOCK = <string_list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of blocks. See Section 6.1 for more information about the use of these command lines for creating a set of blocks used in the command block. Either the `BLOCK` command line or the `INCLUDE ALL BLOCKS` command line must be present in the command block.

4.3.2 Function Commands

If the function option is used, all four function-type command lines, each referencing a user-defined function, must be included in the command block.

Following are the command lines related to the function option:

```
T FUNCTION = <string>t_func_name
X FUNCTION = <string>x_func_name
Y FUNCTION = <string>y_func_name
Z FUNCTION = <string>z_func_name
```

Each of the above command lines references a function name (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block). All of the functions referenced in these four command lines must appear in the domain scope.

The `T FUNCTION` command line gives the name of the user-defined T function. The T function describes how the applied input energy dose is integrated over time t. The T function should be 0 at the start time and 1 at the time at which all energy is deposited. The T function must be monotonically increasing over the time it is defined. The T function describes the total percentage of energy that is deposited at a given time.

The `X FUNCTION`, `Y FUNCTION`, and `Z FUNCTION` command lines define three functions, which we will denote as X, Y, and Z, respectively. The X, Y, and Z functions describe the total amount of energy to be deposited in an element as a function of position. Suppose we have element A with centroid (A_x, A_y, and A_x). The total energy that will have been deposited in element A at time t is given by

$$E_A = X\left(A_x\right)Y\left(A_y\right)Z\left(A_z\right)T\left(t\right) \tag{4.7}$$

where E_A is the total energy deposited.

4.3.3 Input Mesh Command

If the input mesh option is used, the quantity of energy deposited for each element will be read from an element variable defined in the mesh file.

Following is the command line related to the input mesh option:

```
READ VARIABLE = <string>mesh_var_name
```

The string `mesh_var_name` must match the name of an element variable in the mesh file that defines the energy deposition. Suppose that the total energy to be deposited for element A is $\nu(A)$. The quantity of energy deposited at time t is then given by

$$E_A = \nu(A)T(t). \tag{4.8}$$

The T function in Equation (4.8) is the same as that described in Section 4.3.2.

4.3.4 User Subroutine Commands

The user subroutine option allows for a very general description of the energy deposition, but this option requires that you write a user subroutine to implement this capability. The subroutine will be called by Presto at the appropriate time to generate the energy deposition.

Energy deposition uses an element subroutine signature. The subroutine returns one value per element for all of the elements selected by use of the block set commands. The returned value is the current energy flux at an element at a given time. The output flags array is ignored. The total energy deposited in an element is found by a time integration of the returned subroutine fluxes. See Chapter 9 for more information about user subroutines.

Following are the command lines related to the user subroutine option:

```
ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user. Following the `ELEMENT BLOCK SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

4.4 References

1. Stone, C. M. *SANTOS – A Two-Dimensional Finite Element Program for the Quasistatic, Large Deformation, Inelastic Response of Solids*, SAND90-0543. Albuquerque, NM: Sandia National Laboratories, 1996.

2. Bammann, D. J., M. L. Chiesa, and G. C. Johnson. "Modelling Large Deformation and Failure in Manufacturing Processes." In *Proceedings of the 19th International Congress of Theoretical and Applied Mechanics*, edited by T. Tatsumi, E. Watanabe, and T. Kambe, 359–376. Amsterdam: Elsevier Science Publishers, 1997.

3. Bammann, D. J., M. L. Chiesa, M. F. Horstemeyer, and L. E. Weingarten. "Failure in Ductile Materials Using Finite Element Methods." In *Structural Crashworthiness and Failure*, edited by N. Jones and T. Wierzbicki, 1–53. London: Elsevier Applied Science, 1993.

4. Bammann, D. J. "Modeling Temperature and Strain Dependent Large Deformations in Metals." *Applied Mechanics Reviews* 43, no. 5 (1990): S312–319.

5. Taylor, L. M., and D. P. Flanagan. *Pronto3D: A Three-Dimensional Transient Solid Dynamics Program*, SAND87-1912. Albuquerque, NM: Sandia National Laboratories, March 1989.

6. Krieg, R. D. *A Simple Constitutive Description for Cellular Concrete*, SAND SC-DR-72-0883. Albuquerque, NM: Sandia National Laboratories, 1978.

7. Swenson, D. V., and L. M. Taylor. "A Finite Element Model for the Analysis of Tailored Pulse Stimulation of Boreholes." *International Journal for Numerical and Analytical Methods in Geomechanics* 7 (1983): 469–484.

8. Green, A. E., and W. Zerna. *Theoretical Elasticity, 2nd Edition.* Oxford: Clarendon Press, 1968.

9. Whirley, R. G., B. E. Engelmann, and J. O. Halquist. *DYNA3D Users Manual.* Livermore, CA: Lawrence Livermore Laboratory, 1991.

10. Swegle, J. W. *SIERRA: PRESTO Theory Documentation: Energy Dependent Materials Version 1.0.* Albuquerque, NM: Sandia National Laboratories, October 2001.

11. Hammerand, D. C. *Laminated Composites Modeling in ADAGIO/PRESTO*, SAND2004-2143. Albuquerque, NM: Sandia National Laboratories, 2004.

12. Hammerand, D. C. *Critical Time Step for a Bilinear Laminated Composite Mindlin Shell Element*, SAND2004-2487. Albuquerque, NM: Sandia National Laboratories, 2004.

Chapter 5

Elements

This chapter explains how we associate material, geometric, and other properties with the various element blocks in a mesh file. A mesh file contains, for the most part, only topological information about elements. For example, there may be a group of elements in the mesh file that consists of four nodes defining a planar facet in three-dimensional space. Whether or not these elements are used as shells or membranes in our actual model of an object is determined by command lines in the Presto input file. The specifics of a material type associated with these four node facets are also set in the Presto input file.

In addition to the regular elements in Presto, there is some specialized functionality that exhibits element-like behavior. We discuss this functionality—torsional springs and rigid bodies—in this chapter. Furthermore, element-related subjects—mass property calculations, element death, mesh rebalancing—are also discussed in this chapter.

Highlights of chapter contents follow. Section 5.1 discusses the `FINITE ELEMENT MODEL` command block, which provides the description of a mesh that will be associated with the elements. Section 5.2 presents the section command blocks that are used to define the different element sections. Next are descriptions of command blocks that exhibit element-like functionality. Section 5.3.1 describes how to implement a torsional spring mechanism in Presto. Section 5.3.2 explains the use of rigid bodies. In Section 5.4, the `MASS PROPERTIES` command block is described, which lets the user compute the total mass of the model or the mass of subparts of the model once the element blocks are completely defined in terms of geometry and material. Section 5.5 details the `ELEMENT DEATH` command block, which lets the user delete (kill) elements based on various criteria during an analysis. Three command blocks for derived quantities that are used, under certain conditions, with a command line in `ELEMENT DEATH` are discussed in Section 5.6. Finally, Section 5.7 presents various options for partitioning a mesh for parallel runs with Presto. The partitioning scheme can greatly influence the run time for a particular analysis. The command block for selecting a partitioning scheme is `REBALANCE`. The `REBALANCE` references a `ZOLTAN PARAMETERS` command block. The `ZOLTAN PARAMETERS` command block sets various parameters that control the partitioning.

Most of the command blocks and command lines described next appear within the domain scope. There are some exceptions, and these exceptions are noted.

5.1 Finite Element Model

```
BEGIN FINITE ELEMENT MODEL <string>mesh_descriptor
  DATABASE NAME = <string>mesh_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  ALIAS <string>mesh_identifier AS <string>user_name
  BEGIN PARAMETERS FOR BLOCK <string list>block_names
    #
    # Command lines that define attributes for
    # a particular element block appear in this
    # command block.
    #
  END [PARAMETERS FOR BLOCK <string list>block_names]
END [FINITE ELEMENT MODEL <string>mesh_descriptor]
```

The Presto input file must point to a mesh file that is to be used for an analysis. The name of the mesh file appears within a `FINITE ELEMENT MODEL` command block, which appears in the domain scope. In this command block, you will identify the particular mesh file that describes your model. Also within this command block, there will be one or more `PARAMETERS FOR BLOCK` command blocks. (All the `PARAMETERS FOR BLOCK` command blocks are embedded in the `FINITE ELEMENT MODEL` command block.) Within the `PARAMETERS FOR BLOCK` command block, you will set a material type and model, a section, and various other parameters for the element block. The concept of "section" is explained in Section 5.1.2.

Currently, the elements supported in Presto are as follows:

- Eight-node, uniform-gradient hexahedron: Both a midpoint-increment formulation [1] and a strongly objective formulation are implemented [2]. These elements can be used with any of the material models described in Chapter 4, including the equation of state (EOS) models.
- Eight-node, selective-deviatoric hexahedron: Only a strongly objective formulation is provided. This element can be used with any of the material models described in Chapter 4 except the EOS models.

- Four-node tetrahedron: There is now the regular element formulation for the four-node tetrahedron and a node-based formulation for the four-node tetrahedron. For the regular element formulation, only a strongly objective formulation is implemented. The concept of a node-based four-node tetrahedron is described in Reference 3. The regular four-node tetrahedron can be used with any of the material models described in Chapter 4 except the EOS models. The node-based tetrahedron can be used with any of the material models described in Chapter 4, including the EOS models.

- Eight-node tetrahedron: This tetrahedral element has nodes at the four vertices and nodes on the four faces. The eight-node tetrahedron has only a strongly objective formulation [4]. The eight-node tetrahedron uses a mean quadrature formulation even though it has the additional nodes. This element can be used with any of the material models described in Chapter 4, including the EOS models.

- Ten-node tetrahedron: Only a strongly objective formulation is implemented. This element can be used with any of the material models described in Chapter 4, including the EOS models.

- Four-node, quadrilateral, uniform-gradient membrane: Both a midpoint-increment formulation and a strongly objective formulation are implemented. This element is derived from the Key-Hoff shell formulation [5]. The strongly objective formulation has not been extensively tested, and it is recommended that the midpoint-increment formulation, which is the default, be used for this element type. These elements can be used with any of the following material models described in Chapter 4:

  – Elastic
  – Elastic-plastic
  – Elastic-plastic power-law hardening
  – Multilinear elastic-plastic power-law hardening (no failure)

- Four-node, quadrilateral shell: This shell uses the Key-Hoff formulation [5]. Both a midpoint-increment formulation and a strongly objective formulation are implemented. The strongly objective formulation has not been extensively tested, and it is recommended that the midpoint-increment formulation, which is the default, be used for this element type. These elements can be used with any of the following material models described in Chapter 4:

  – Elastic
  – Elastic-plastic
  – Elastic-plastic power-law hardening

  – Multilinear elastic-plastic power-law hardening (no failure)

- Four-node, quadrilateral, selective-deviatoric membrane: Only a midpoint-increment formulation is implemented. These elements can be used with any of the following material models described in Chapter 4:
  – Elastic
  – Elastic-plastic
  – Elastic-plastic power-law hardening
  – Multilinear elastic-plastic power-law hardening (no failure)
- Two-node beam: The beam element is a uniform result model. Strains and stresses are computed only at the midpoint of the element. These midpoint values determine the forces and moments for the beam. There are five different sections currently implemented for the beam: rod, tube, bar, box, and I. This element can be used with any of the following material models described in Chapter 4:
  – Elastic
  – Elastic-plastic
- Two-node truss: The two-node truss element carries only a uniform axial stress. Currently, there is a linear-elastic material model for the truss element.
- Two-node damper: The two-node damping element computes a damping force based on the relative velocity of the two nodes along the axis of the element. This element uses only a damping parameter for a material property.
- Point mass: Presto has a point mass element that allows the user to put a specified mass (and a mass only) at a node. This element requires input for density and an elastic material, but does not make use of the elastic material properties.
- Smoothed particle hydrodynamics (SPH) elements: These are one-dimensional elements. These elements can be used with any of the material models described in Chapter 4, including the EOS models.

The command block to describe a mesh file begins with

```
BEGIN FINITE ELEMENT MODEL <string>mesh_descriptor
```

and is terminated with

```
END [FINITE ELEMENT MODEL <string>mesh_descriptor] ,
```

where `mesh_descriptor` is a user-selected name for the mesh. In this section, we will first discuss the command lines within the scope of the `FINITE ELEMENT MODEL` command block but outside the scope of the `PARAMETERS FOR BLOCK` command block. We will then discuss the `PARAMETERS FOR BLOCK` command block and the associated command lines for this particular block.

5.1.1 Identification of Mesh File

Nested within the `FINITE ELEMENT MODEL` command block are two command lines (`DATABASE NAME` and `DATABASE TYPE`) that give the mesh name and define the type for the mesh file, respectively. The command line

```
DATABASE NAME = <string>mesh_file_name
```

gives the name of the mesh file with the string `mesh_file_name`. If the current mesh file is in the default directory and is named `job.g`, then this command line would appear as

```
DATABASE NAME = job.g .
```

If the mesh file is in some other directory, the command line would have to show the path to that directory. For parallel runs, the string `mesh_file_name` is the base name for the spread of parallel mesh files. For example, for a four-processor run, the actual mesh files associated with a base name of `job.g` would be `job.g.4.0`, `job.g.4.1`, `job.g.4.2`, and `job.g.4.3`. The database name on the command line would be `job.g`.

If the mesh file does not use the Exodus II format, you must specify the format for the mesh file using the command line

```
DATABASE TYPE = <string>database_type(exodusII) .
```

Currently, only the Exodus II database is supported by Presto. Other options will be added in the future.

It is possible to associate a user-defined name with some mesh entity. The mesh entity for Exodus II relies on some type of integer identification. You can relate the integer identification to some name that is more descriptive by using the `ALIAS command line`:

```
ALIAS <string>mesh_identifier AS <string>user_name .
```

This alias can then be used in other locations in the input file in place of the Exodus II name.

Examples of this association are as follows:

```
Alias block_1   as Case
Alias block_10  as Fin
Alias block_12  as Nose
Alias surface_1 as Nose_Case_Interface
Alias surface_2 as OuterBoundary
```

The above examples use the Exodus II naming convention described in Section 1.5.

5.1.2 Descriptors of Element Blocks

```
BEGIN PARAMETERS FOR BLOCK <string list>block_names
  MATERIAL <string>material_name
  SOLID MECHANICS USE MODEL <string>model_name
  SECTION = <string>section_id
  LINEAR BULK VISCOSITY =
    <real>linear_bulk_viscosity_value(0.06)
  QUADRATIC BULK VISCOSITY =
    <real>quad_bulk_viscosity_value(1.20)
  HOURGLASS STIFFNESS =
    <real>hour_glass_stiff_value(solid = 0.05,
      shell/membrane = 0.0)
  HOURGLASS VISCOSITY =
    <real>hour_glass_visc_value(solid = 0.0,
      shell/membrane = 0.0)
  EFFECTIVE MODULI MODEL = <string>PRESTO|PRONTO|CURRENT|
      ELASTIC(PRONTO)
  ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD)
  ACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
    <string list>period_names
  RIGID BODY = <string>rb_name
END [PARAMETERS FOR BLOCK <string list>block_names]
```

The finite element model consists of one or more element blocks. Associated with an element block or group of element blocks will be a `PARAMETERS FOR BLOCK` command block, which is also referred to in this document as an *element-block command block*. The basic information about the element blocks (number of elements, topology, connectivity, etc.) is contained in a mesh file. Specific attributes for an element block must be specified in the Presto input file. If for example, a block of eight-node hexahedra is to use the selective-deviatoric versus mean-quadrature formulation, then the selective-deviatoric formulation must be specified in the input file. The elements currently implemented in Presto are listed at the beginning of Section 5.1.

The element-block command block begins with the input line

```
BEGIN PARAMETERS FOR BLOCK <string list>block_names
```

and is terminated with the input line

```
END [PARAMETERS FOR BLOCK <string list>block_names] ,
```

where `block_names` is the list of element blocks assigned to the element-block command block. If the format for the mesh file is Exodus II, the typical form of a `block_name` is `block_integerid`, where `integerid` is the integer identifier for the block. If the element block is 280, the value of `block_name` would be `block_280`. It is also possible to generate an alias identifier for the element block and use this for the `block_name`. If `block_280` is aliased to `AL6061`, then `block_name` becomes `AL6061`.

All the element blocks listed on the `PARAMETERS FOR BLOCK` command line will have the same mechanics properties. The mechanics properties are set by use of the various command lines. One of the key command lines, i.e., `MATERIAL`, will let you associate a material with the elements in the block. Another key command line is the `SECTION` command line. This command line lets you differentiate between elements with the same topology but different formulations. For example, assume that the topology of the elements in a block is a four-node quadrilateral. With the `SECTION` command line you can specify whether the element block will be used as a membrane or a shell. The `SECTION` command line also lets you assign a variety of parameters to an element, depending on the element formulation.

It is important to state here that the `SECTION` command line only specifies an identifier that maps to a section command block that is defined by the user. There are currently several kinds of section command blocks for the different elements: `SOLID SECTION`, `SHELL SECTION`, `MEMBRANE SECTION`, `BEAM SECTION`, `TRUSS SECTION`, `DAMPER SECTION`, `POINT MASS SECTION`, and `SPH SECTION`. It is within a section command block that the formulation-specific entities related to a particular element are specified. If no `SECTION` command line is present in an element-block command block, Presto assumes the element block is a block of eight-node hexahedra using mean quadrature and the midpoint-increment formulation.

(Similar to the `SECTION` command line is a `RIGID BODY` command line. The `RIGID BODY` command line specifies an identifier that maps to a rigid body command block that appears in the region scope. This identifier ties an element block to a specific rigid body.)

In addition to the material- and section-related command lines in an element-block command block, there are a number of other command lines. There are, for example, two command lines, `HOURGLASS STIFFNESS`, and `HOURGLASS VISCOSITY`, that will let you specify hourglass control parameters for the elements in the block (if these elements use hourglass control). These two command lines will overwrite the default hourglass control parameters for all elements in the block.

All the command lines that can be used for the element-block command block are described in Section 5.1.2.1 through Section 5.1.2.8. The various section command blocks are described in Section 5.2. The section command blocks and their related command lines are much easier to understand once the element-block command lines are described.

5.1.2.1 Material Property

```
MATERIAL <string>material_name
SOLID MECHANICS USE MODEL <string>model_name
```

The material property specification for an element block is done by using the above two command lines. The property specification references both a `PROPERTY SPECIFICATION FOR MATERIAL` command block and a material-model command block, which has the general form `PARAMETERS FOR MODEL model_name.` These command blocks are described in Chapter 4. The `PROPERTY SPECIFICATION FOR MATERIAL` command block contains all the parameters needed to define a material, and is associated with an element block (`PARAMETERS FOR BLOCK` command block) by use of the `MATERIAL` command line. Some of the material parameters inside the property specification are grouped on the basis of material models. A material-model command block is associated with an element block by use of the `SOLID MECHANICS USE MODEL` command line.

Consider the following example. Suppose there is a `PROPERTY SPECIFICATION FOR MATERIAL` command block with a `material_name` of steel. Embedded within this command block for steel is a material-model command block for an elastic model of steel and an elastic-plastic model of steel. Suppose that for the current element block we would like to use the material steel with the elastic model. Then the element-block command block would contain the input lines

```
MATERIAL steel
SOLID MECHANICS USE MODEL elastic .
```

If, on the other hand, we would like to use the material steel with the elastic-plastic model, the element-block command block would contain the input lines

```
MATERIAL steel
SOLID MECHANICS USE MODEL elastic_plastic .
```

The user should remember that not all material types can be used with all element types.

5.1.2.2 Section

```
SECTION = <string>section_id
```

The section specification for an element-block command block is done by using the above command line. The `section_id` is a string associated with a section command block. The various section command blocks are described in Section 5.2.

Suppose you wanted the current element-block command block to use the membrane formulation. You would define a `MEMBRANE SECTION` command block with some name, such as `membrane_rubber`. Inside the current element-block command block you would have the command line

```
SECTION = membrane_rubber .
```

The thickness of the membrane would be described in the `MEMBRANE SECTION` command block and then associated with the current element-block command block.

There can be only one `SECTION` command line in an element-block command block. Each element-block command block within the model description can reference a unique section command block, or several element-block command blocks can reference the same section command block. For example, in Figure 5.1, the section named `membrane_rubber` appears in two different `PARAMETERS FOR MODEL` command blocks, but there is only one specification for their associated `MEMBRANE SECTION` command block. When several element-block command blocks reference the same section, the input file is less verbose, and it is easier to maintain the input file.

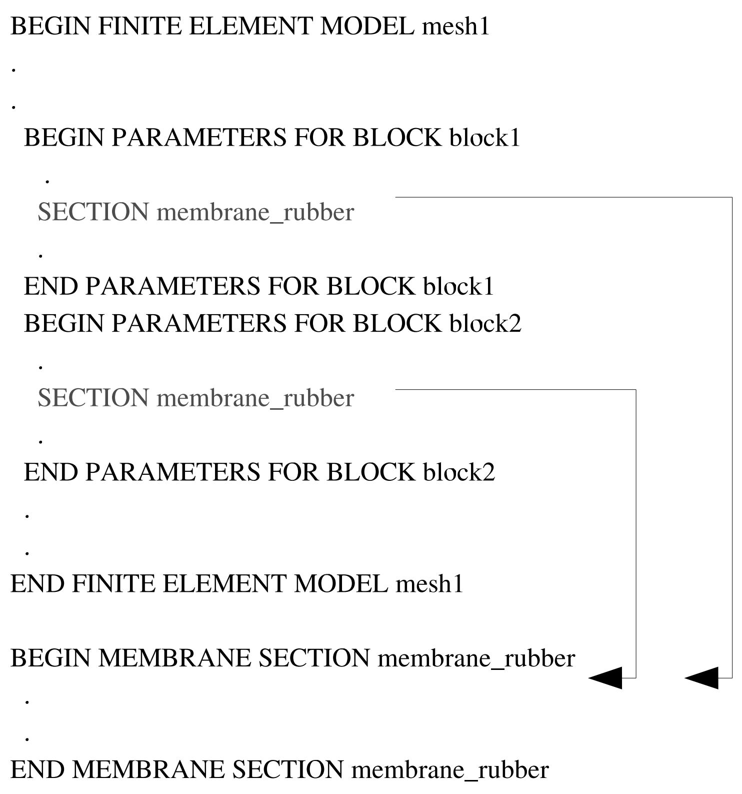
5.1.2.3 Linear and Quadratic Bulk Viscosity

```
LINEAR BULK VISCOSITY =
  <real>linear_bulk_viscosity_value(0.06)
QUADRATIC BULK VISCOSITY =
  <real>quad_bulk_viscosity_value(1.20)
```

The linear and quadratic bulk viscosity are set with these two command lines. Consult the documentation for the elements [6] for a description of the bulk viscosity parameters.

5.1.2.4 Hourglass Control

```
HOURGLASS STIFFNESS = <real>hour_glass_stiff_value(solid
  = 0.05, shell/membrane = 0.0)
HOURGLASS VISCOSITY = <real>hour_glass_visc_value(solid
  = 0.0, shell/membrane = 0.0)
```



```
BEGIN FINITE ELEMENT MODEL mesh1
.
.
 BEGIN PARAMETERS FOR BLOCK block1
   .
  SECTION membrane_rubber
  .
 END PARAMETERS FOR BLOCK block1
 BEGIN PARAMETERS FOR BLOCK block2
  .
  SECTION membrane_rubber
  .
 END PARAMETERS FOR BLOCK block2
 .
 .
END FINITE ELEMENT MODEL mesh1

BEGIN MEMBRANE SECTION membrane_rubber
 .
 .
END MEMBRANE SECTION membrane_rubber
```

Figure 5.1: Association between `SECTION` command lines and a section command block.

These two command lines set the hourglass control parameters for elements that use hourglass control. Currently, the included elements are the eight-node, uniform-gradient hexahedral elements; the eight-node and ten-node tetrahedral elements; and the four-node membrane and shell elements. Consult the element documentation [6] for a description of the hourglass parameters.

The hourglass stiffness parameter defaults to 0.05 for solids using hourglass control; it defaults to 0.0 for shell and membrane elements. The hourglass viscosity parameter defaults to 0.0 for all elements currently using hourglass control.

The hourglass stiffness is the same as the dilatational hourglass parameter, and the hourglass viscosity is the same as the deviatoric hourglass parameter.

The computation of the hourglass parameters can be strongly affected by the method that computes the effective moduli. The command line in Section 5.1.2.5 selects the method for computing the effective moduli.

5.1.2.5 Effective Moduli Model

```
EFFECTIVE MODULI MODEL =
  <string>PRESTO|PRONTO|CURRENT|ELASTIC(PRONTO)
```

The hourglass force computations require a measure of the material moduli to ensure appropriate scaling of the hourglass forces. For elastic, isotropic material models, the moduli are constant throughout the analysis. However, for nonlinear materials, the moduli are typically computed numerically from the stresses. For models with softening regimes or that approach perfect plasticity, the moduli may be difficult to define, and the way in which they are computed may adversely affect the analysis. Through the `EFFECTIVE MODULI MODEL` command line, Presto provides several methods for the computation of these effective moduli:

- `PRESTO`: This method includes a number of techniques for returning reasonable moduli for softening and perfectly plastic materials. The effective moduli that this approach produces are stiffer than those computed by the `PRONTO` approach.
- `PRONTO`: This method is the default and is identical to the method of computing effective moduli present in the Pronto3D code. It is similar to the `PRESTO` approach but generally produces moduli that are softer than the `PRESTO` approach.
- `CURRENT`: This method computes the effective moduli without any extra handling of negative or near-zero moduli cases. It generally provides the softest response but is also less stable.
- `ELASTIC`: This method simply uses the initial elastic moduli for the entire analysis. It is the most robust but also the most stiff, and may produce an overly stiff global response.

The `EFFECTIVE MODULI MODEL` command line should be used with caution because it can strongly affect the analysis results.

5.1.2.6 Element Numerical Formulation

```
ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD)
```

For calculation of the critical time step, it is necessary to determine a characteristic length for each element. In one dimension, the correct characteristic element length is obviously the distance between the two nodes of the element. In higher dimensions, this length is usually taken to be the minimum distance between any of the nodes in the element. However, some finite element codes, primarily those based on Pronto3D [1], use as a characteristic length an eigenvalue estimate based on work by Flanagan and Belytschko [7]. That characteristic length provides a stable time step, but in many cases is far more conservative than the minimum distance between nodes. For a cubic element with side length equal to 1, and thus also surface area of each face and volume equal to 1, the minimum distance between nodes is 1. However, the eigenvalue estimate is $1/\sqrt{3}$, which is only 58% of the minimum distance. As the length of the element is increased in one direction while keeping surfaces in the lateral direction squares of area 1, the eigenvalue estimate asymptotes to $1/\sqrt{2}$ for very long elements. If the length is decreased, the eigenvalue estimate asymptotes to the minimum distance between nodes for very thin elements. In this case, the eigenvalue estimate is always more conservative than the minimum distance between nodes. However, consider an element whose cross section in one direction is not a square but a trapezoid with one side length much greater than the other. Assume the large side length is 1 and the other side length is arbitrarily small, ε. In this case, the minimum distance between nodes becomes ε, creating a very small and inefficient time step. However, the eigenvalue estimate is related to the length across the middle of the trapezoid, which for the conditions stated is 1/2. Since both distances provide stable time steps, and one or the other can be much larger in various circumstances, the most efficient calculation is obtained by using the maximum of the two lengths, either the eigenvalue estimate or the minimum distance between nodes, to determine the time step.

By using the maximum of the lengths, the computed critical time step should be at the edge of instability, and the `TIME STEP SCALE FACTOR` command line should be used to provide a margin of safety. In this case the scale factor for the time step should not be greater than 0.9, and in some cases it may have to be reduced further. Thus, although the maximum of the lengths provides a time step that is closer to the critical value and provides better accuracy and efficiency, you may need to specify a smaller-than-expected scale factor for stability. For this reason, the choice of which approach to use is left to the user and is determined by the command line

```
ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD) .
```

If the input parameter is `OLD`, only the eigenvalue estimate is used; `NEW` means that the maximum of the two lengths is used. The default is `OLD` so that users will have to specifically choose the new approach and be aware of the scale factor for the time step.

The `ELEMENT NUMERICAL FORMULATION` command line is applicable to both the energy-dependent and purely mechanical material models. If this command line is applied to blocks using energy-dependent materials, only the determination of the characteristic length is affected. If this command line is applied to an element block with a purely mechanical model and the `OLD` option is used, the Pronto3D-based artificial viscosity, time step, and eigenvalue estimate will be used in the element calculations. If, however, the `NEW` option is used, the artificial viscosity and time step will be computed from equations associated with the energy-dependent models. You should consult Reference 8 for further details about the critical time-step calculations and the use of this command line.

5.1.2.7 Activation/Deactivation of Element Blocks by Time

```
ACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
  <string list>period_names
```

This command line permits the activation and deactivation of element blocks by time period. The time periods are defined in the `TIME STEPPING BLOCK` command block (Section 3.1.1) within a specific procedure named in a `PRESTO PROCEDURE` command block (Section 2.2.1). In the `ACTIVE FOR PROCEDURE` command line, the element block is active for all periods listed for the named procedure. The element block is deactivated for all time periods that are absent from the list. If this command line is not used, then by default the block is active during all time periods. While this command line controls the activation and deactivation of all elements in a block, individual elements can be deactivated with the `ELEMENT DEATH` command block (see Section 5.5).

5.1.2.8 Rigid Body Declaration

```
RIGID BODY = <string>rb_name
```

Any combination of elements blocks (except a combination involving SPH elements) can be used to create a rigid body. Suppose you wish to create a rigid body named `rigidbody_1`. If you want to include element block 280 as part of this rigid body, then, in the `PARAMETERS FOR BLOCK` command block for element block 280, you will include the command line

```
RIGID BODY = rigidbody_1 .
```

Consult with Section 5.3.2 for a full discussion of how to create rigid bodies in Presto.

5.2 Element Sections

Element sections are defined by section command blocks. There are currently eight different types of section command blocks. The section command blocks appear in the domain scope, at the same level as the `FINITE ELEMENT MODEL` command block. In general, a section command block has the following form:

```
BEGIN section_type SECTION <string>section_name
  command lines dependent on section type
END [section_type SECTION <string>section_name]
```

Currently, `section_type` can be `SOLID`, `SHELL`, `MEMBRANE`, `BEAM`, `TRUSS`, `DAMPER`, `POINT MASS`, or `SPH`. These various section types are identified as individual section command blocks and are described below. The corresponding `section_name` parameter in each of these command blocks, e.g., `truss_section_name` in the `TRUSS SECTION` command block, is selected by the user. Associating these names with individual `SECTION` command lines in `PARAMETERS FOR BLOCK` command blocks is discussed in Section 5.1.2.2.

5.2.1 Solid Section

```
BEGIN SOLID SECTION <string>solid_section_name
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC(MEAN_QUADRATURE)
  DEVIATORIC PARAMETER = <real>deviatoric_param
  STRAIN INCREMENTATION = <string>MIDPOINT_INCREMENT|
    STRONGLY_OBJECTIVE|NODE_BASED(MIDPOINT_INCREMENT)
  NODE BASED ALPHA FACTOR = <real>bulk_stress_weight(0.01)
  NODE BASED BETA FACTOR = <real>shear stress_weight(0.35)
END [SOLID SECTION <string>solid_section_name]
```

The `SOLID SECTION` command block is used to specify the properties for solid elements (hexahedra and tetrahedra). This command block is to be referenced by an element block made up of solid elements. The two types of solid-element topologies currently supported by Presto are hexahedra and tetrahedra. The parameter `solid_section_name` is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The `FORMULATION` command line specifies whether the element will use a single-point integration rule (mean quadrature) or a selective-deviatoric rule. The selective-deviatoric integration rule is a higher-order integration scheme, which is discussed below.

If the user wishes to use the selective-deviatoric rule, the `DEVIATORIC PARAMETER` command line must also appear in the `SOLID SECTION` command block. The selective-deviatoric parameter, `deviatoric_param`, which is valid from 0.0 to 1.0, indicates how much of the deviatoric response should be taken from a uniform-gradient integration and how much should be taken from a full integration of the element. A value of 0.0 will give a pure uniform-gradient response with no hourglass control. Thus, this value is of little practical use. A value of 1.0 will give a fully integrated deviatoric response. Although any value between 0.0 and 1.0 is perfectly valid, lower values are generally preferred.

The selective-deviatoric elements, when used with a value greater than 0.0, provide hourglass control without artificial hourglass parameters.

Some of the solid elements support different strain-incrementation formulations. See the element summary at the beginning of Section 5.1 to determine what strain-incrementation formulations are available for the various elements. The `STRAIN INCREMENTATION` command line, lets you specify a midpoint-increment strain formulation (`MIDPOINT_INCREMENT`), a strongly objective strain formulation (`STRONGLY_OBJECTIVE`), or a node-based formulation (`NODE_BASED`) for some of the elements. Consult the element documentation [2, 6] for a description of these strain formulations.

The node-based formulation can only be used with four-node tetrahedral elements. If your element-block command block (i.e., a `PARAMETERS FOR BLOCK` command block) has a `SECTION` command line that references a `SOLID SECTION` command block that uses

```
STRAIN INCREMENTATION = NODE_BASED ,
```

then the element block must be a block of four-node tetrahedral elements.

The node-based formulation lets you calculate a solution that is some mixture of an element-based formulation (information from the center of an element) and a node-based formulation (information at a node that is based on all elements attached to the node). The node-based tetrahedron allows the user to model with four-node tetrahedral elements and avoid the main problems with regular tetrahedral elements. Regular tetrahedral elements are much too stiff and can produce very inaccurate results.

You can adjust the mixture of node-based versus element-based information incorporated into your solution with the `NODE BASED ALPHA FACTOR` and `NODE BASED BETA FACTOR` command lines. These two lines apply only if you have selected the `NODE BASED` option on the `STRAIN INCREMENTATION` command line. The value for `bulk_stress_weight` on the `NODE BASED ALPHA FACTOR` command line sets the element bulk stress weighting factor; the value for `shear_stress_weight` on the `NODE BASED BETA FACTOR` command line sets the element shear stress weighting factor. You should consult Reference 3 to better understand the use of these weighting factors. If both of these factors are set to 0.0, you will be using a strictly node-based formulation. If both of these factors are set to 1.0, you will be using a strictly element-based formulation.

5.2.2 Shell Section

```
BEGIN SHELL SECTION <string>shell_section_name
  THICKNESS = <real>shell_thickness
  THICKNESS MESH VARIABLE =
    <string>THICKNESS|<string>var_name
  THICKNESS TIME STEP = <real>time_value
  THICKNESS SCALE FACTOR = <real>thick_scale_factor(1.0)
  INTEGRATION RULE = TRAPEZOID|GAUSS|LOBATTO|SIMPSONS|
    USER(TRAPEZOID)
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points(5)
  BEGIN USER INTEGRATION RULE
    <real>location_1 <real>weight_1
    <real>location_2 <real>weight_2
    .
    .
    <real>location_n <real>weight_n
  END [USER INTEGRATION RULE]
  LOFTING FACTOR = <real>lofting_factor(0.5)
  ORIENTATION = <string>orientation_name
END [SHELL SECTION <string>shell_section_name]
```

The `SHELL SECTION` command block is used to specify the properties for a shell element. If this command block is referenced in an element block of three-dimensional, four-node elements, the elements in the block will be treated as shell elements. The parameter, `shell_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

Either a `THICKNESS` command line or a `THICKNESS MESH VARIABLE` command line must appear in the `SHELL SECTION` command block.

If a shell element block references a `SHELL SECTION` command block with the command line

```
THICKNESS = <real>shell_thickness ,
```

then all the membrane elements in the block will have their thickness initialized to the value `shell_thickness`.

Presto can also initialize the thickness using an attribute defined on elements in the mesh file. Meshing programs such as PATRAN and CUBIT typically set the element thickness as an attribute on the elements. If the elements have one and only one attribute defined on the mesh, the `THICKNESS MESH VARIABLE` command line should be specified as

```
THICKNESS MESH VARIABLE = THICKNESS ,
```

which causes the thickness of the element to be initialized to the value of the attribute for that element. If there are zero attributes or more than one attribute, the thickness variable will not be automatically defined, and the command will fail.

The thickness may also be initialized by any other field present on the input mesh. To specify a field other than the single-element attribute, use this form of the `THICKNESS MESH VARIABLE` command line:

```
THICKNESS MESH VARIABLE = <string>var_name ,
```

where the string `var_name` is the name of the initializing field.

The input mesh may have values defined at more than one point in time. To choose the point in time in the mesh file that the variable should be read, use the command line

```
THICKNESS TIME STEP = <real>time_value .
```

The default time point in the mesh file at which the variable is read is 0.0.

Once the thickness of a shell element is initialized by using either the `THICKNESS` command line or the `THICKNESS MESH VARIABLE` command line, this initial thickness value can then be scaled using the scale-factor command line

```
THICKNESS SCALE FACTOR = <real>thick_scale_factor .
```

If the initial thickness of the shell is 0.15 inch, and the value for `thick_scale_factor` is 0.5, then the scaled thickness of the membrane will be 0.075.

The thickness mesh variable specification may be coupled with the `THICKNESS SCALE FACTOR` command line. In this case, the thickness mesh variable is scaled by the specified factor.

For shell elements, the user can select from a number of integration rules, including a user-defined integration option. The integration rule is selected with the command line

```
INTEGRATION RULE = <string>TRAPEZOID|GAUSS|LOBATTO|SIMPSONS|
  USER(TRAPEZOID) .
```

Consult the element documentation [6] for a description of different integration schemes for shell elements.

The default integration scheme is `TRAPEZOID` with five integration points through the thickness. The number of integration points for `TRAPEZOID` can be set to any number greater than one by using the following command line:

```
NUMBER OF INTEGRATION POINTS = <integer>num_int_points(5)
```

The `SIMPSONS`, `GAUSS`, and `LOBATTO` integration schemes in the `INTEGRATION RULE` command line all default to five integration points. The number of integration points for these three schemes can be reset by using the `NUMBER OF INTEGRATION POINTS` command line. There are limitations on the number of integration points for some of these integration rules. The `SIMPSONS` rule can be set to any number greater than one, the `GAUSS` scheme can be set to one through seven integration points, and the `LOBATTO` integration scheme can be set to two through seven integration points.

In addition to these standard integration schemes, you may also define an integration scheme by using the `USER INTEGRATION RULE` command block.

```
BEGIN USER INTEGRATION RULE
  <real>location_1 <real>weight_1
  <real>location_2 <real>weight_2
  .
  .
  <real>location_n <real>weight_n
END [USER INTEGRATION RULE]
```

You may NOT specify both a standard integration scheme and a user scheme. If the `USER` option is specified in the `INTEGRATION RULE` command line, a set of integration locations with associated weight factors must be specified. This is done with tabular input command lines inside the `USER INTEGRATION RULE` command block. The number of command lines inside this command block should match the number of integration points specified in the `NUMBER OF INTEGRATION POINTS` command line. For example, suppose we wish to use a user-defined scheme with three integration points. The `NUMBER OF INTEGRATION POINTS` command line should specify three (3) integration points and the number of command lines inside the `USER INTEGRATION RULE` command block should be three (to give three locations and three weight factors).

For the user-defined rule, the integration point locations should fall between –1 and +1, and the weights should sum to 1.0.

The command line

```
LOFTING FACTOR = <real>lofting_factor(0.5)
```

allows the user to shift the location of the midsurface of a shell element relative to the geometric location of the shell element. By default, the geometric location of a shell element in a mesh represents the midsurface of the shell. If a shell has a thickness of 0.2 inch, the top surface of the shell is 0.1 inch above the geometric surface defined by the shell element, and the bottom surface of the shell is 0.1 inch below the geometric surface defined by the shell element. (The top surface of the shell is the surface with a positive element normal; the bottom surface of the shell is the surface with a negative element normal.)

Figure 5.2 shows an edge-on view of shell elements with a thickness of t and the location of the geometric plane in relation to the shell surfaces for three different values of the lofting factor—0.0, 0.5, and 1.0. If you want to have the geometric surface defined by the shell correspond to the top surface of the shell element, set the lofting factor to 1.0. If you want to have the geometric surface defined by the shell correspond to the bottom surface of the shell element, set the lofting factor to 0.0. The geometric surface is midway between the top and bottom surfaces for a lofting factor of 0.5. Note that the default for this factor is 0.5.

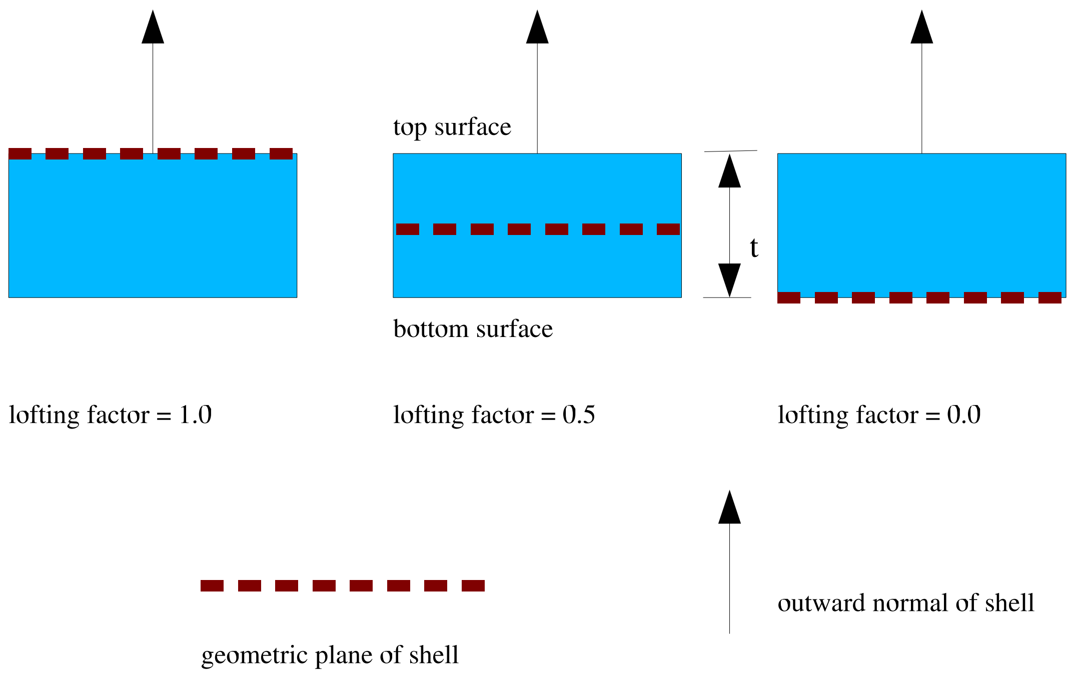


Figure 5.2: Location of geometric plane of shell for various lofting factors.

Suppose that the lofting factor is set to 1.0 and the thickness of the shell is 0.2 inch. Let us measure distances to the shell surfaces (top and bottom) by measuring along the positive element normal. The top surface of the shell will be located at a distance of 0.0 inch from the geometric shell surface, and the bottom surface of the shell will be located at a distance of –0.2 inch from the geometric shell surface.

Both the shell mechanics and contact use shell lofting. See Section 7.2 for a discussion of lofting surfaces for shells and contact.

The `ORIENTATION` command line lets you select a coordinate system for output of stresses. The `ORIENTATION` option makes use of an embedded coordinate system rst associated with each shell element. The rst coordinate system for a shell element is shown in Figure 5.3. The r-axis extends from the center of the shell to the midpoint of the side of the shell defined by nodes 1 and 2. The t-axis is located at the center of the shell and is normal to the surface of the shell at the center point. The s-axis is the cross-product of the t-axis and the r-axis. The rst-axes form a local coordinate system at the center of the shell; this local coordinate system moves with the shell element as the element deforms.

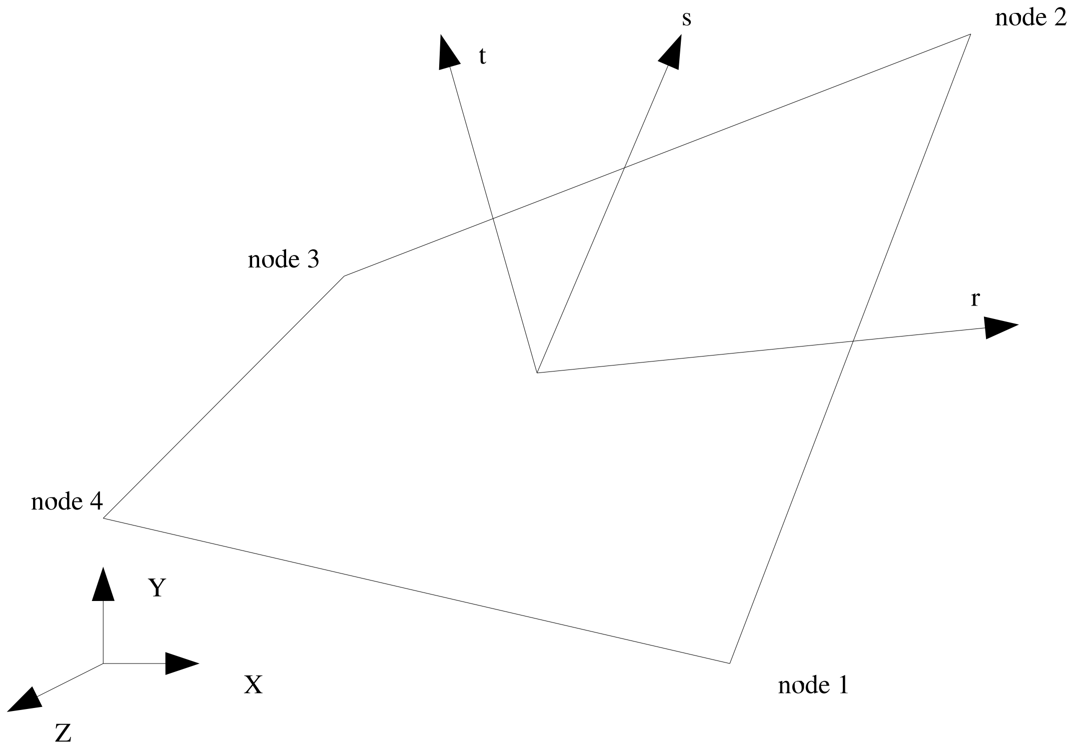


Figure 5.3: Local rst coordinate system for a shell element.

The `ORIENTATION` command line in the `SHELL SECTION` command block references an `ORIENTATION` command block that appears in the domain scope. As described in Chapter 2 of this document, the `ORIENTATION` command block can be used to define a local coordinate system $X''Y''Z''$ at the center of a shell element. In the original shell configuration (time 0), one of the axes—X'', Y'', or Z''—is projected onto the plane of the shell element. The angle between this projected axis of the $X''Y''Z''$ coordinate system and the r-axis is used to establish the transformation for the stresses. We will illustrate this with an example.

Suppose that in our `ORIENTATION` command block we have specified a rotation of 30 degrees about the 1-axis (X'-axis). The command line for this rotation in the `ORIENTATION` command block would be

```
    ROTATION ABOUT 1 = 30 .
```

For this case, we project the Y''-axis onto the plane of the shell (Figure 5.4). The angle between this projection and the r-axis establishes a transformation for the in-plane stresses of the shell (the stresses in the center of the shell lying in the plane of the shell). What will be output as the in-plane stress σ_{xx}^{ip} will be in the Y''-direction; what will be output as the in-plane stress σ_{yy}^{ip} will be in the Z''-direction. The in-plane stress σ_{xy}^{ip} is a shear stress in the $Y''Z''$-plane. The $X''Y''Z''$ coordinate system maintains the same relative position in regard to the rst coordinate system. This means that the $X''Y''Z''$ coordinate system is a local coordinate system that moves with the shell element as the element deforms.

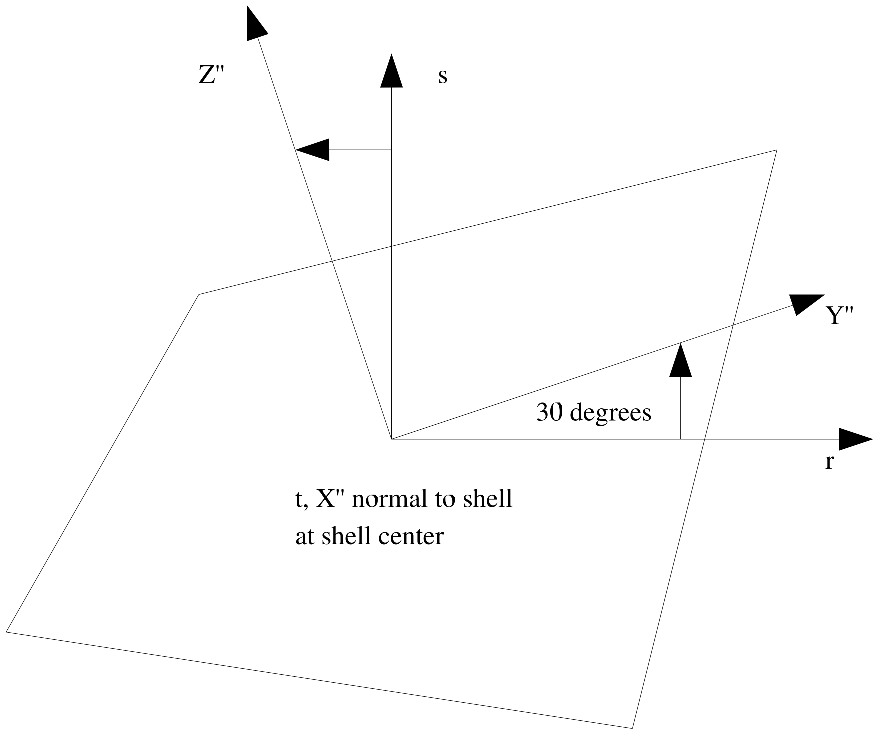


Figure 5.4: Rotation of 30 degrees about the 1-axis (X'-axis).

The following permutations for output of the in-plane stresses occur depending on the axis (1, 2, or 3) specified in the `ROTATION ABOUT` command line:

- Rotation about the 1-axis (X'-axis): The in-plane stress σ_{xx}^{ip} will be in the Y''-direction; the in-plane stress σ_{yy}^{ip} will be in the Z''-direction. The in-plane stress

σ^{ip}_{xy} is a shear stress in the $Y''Z''$-plane.

- Rotation about the 2-axis (Y'-axis): The in-plane stress σ^{ip}_{xx} will be in the Z''-direction; the in-plane stress σ^{ip}_{yy} will be in the X''-direction. The in-plane stress σ^{ip}_{xy} is a shear stress in the $Z''X''$-plane.
- Rotation about the 3-axis (Z'-axis): The in-plane stress σ^{ip}_{xx} will be in the X''-direction; the in-plane stress σ^{ip}_{yy} will be in the Y''-direction. The in-plane stress σ^{ip}_{xy} is a shear stress in the $X''Y''$-plane.

5.2.3 Membrane Section

```
BEGIN MEMBRANE SECTION <string>membrane_section_name
  THICKNESS = <real>mem_thickness
  THICKNESS MESH VARIABLE =
    <string>THICKNESS|<string>var_name
  THICKNESS TIME STEP = <real>time_value
  THICKNESS SCALE FACTOR = <real>thick_scale_factor(1.0)
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC(MEAN_QUADRATURE)
  DEVIATORIC PARAMETER = <real>deviatoric_param
  LOFTING FACTOR = <real>lofting_factor(0.5)
END [MEMBRANE SECTION <string>membrane_section_name]
```

The `MEMBRANE SECTION` command block is used to specify the properties for a membrane element. If this command block is referenced in an element block of three-dimensional, four-node elements, the elements in the block will be treated as a membrane element. The parameter, `membrane_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

Either a `THICKNESS` command line or a `THICKNESS MESH VARIABLE` command line must appear in the `MEMBRANE SECTION` command block.

If a membrane element block references a `MEMBRANE SECTION` command block with the command line

```
THICKNESS = <real>mem_thickness ,
```

then all the membrane elements in the block will have their thickness initialized to the value `mem_thickness`.

Presto can also initialize the thickness using an attribute defined on elements in the mesh file. Meshing programs such as PATRAN and CUBIT typically set the element thickness as an attribute on the elements. If the elements have one and only one attribute defined on the mesh, the `THICKNESS MESH VARIABLE` command line should be specified as

```
THICKNESS MESH VARIABLE = THICKNESS ,
```

which causes the thickness of the element to be initialized to the value of the attribute for that element. If there are zero attributes or more than one attribute, the thickness variable will not be automatically defined, and the command will fail.

The thickness may also be initialized by any other field present on the input mesh. To specify a field other than the single-element attribute, use this form of the `THICKNESS MESH VARIABLE` command line:

```
THICKNESS MESH VARIABLE = <string>var_name ,
```

where the string `var_name` is the name of the initializing field.

The input mesh may have values defined at more than one point in time. To choose the point in time in the mesh file that the variable should be read, use the command line

```
THICKNESS TIME STEP = <real>time_value .
```

The default time point in the mesh file at which the variable is read is 0.0.

Once the thickness of a membrane element is initialized by using either the `THICKNESS` command line or the `THICKNESS MESH VARIABLE` command line, this initial thickness value can then be scaled by using the scale-factor command line

```
THICKNESS SCALE FACTOR = <real>thick_scale_factor .
```

If the initial thickness of the membrane is 0.15 inch, and the value for `thick_scale_factor` is 0.5, then the scaled thickness of the membrane will be 0.075.

The `FORMULATION` command line specifies whether the element will use a single-point integration rule (mean quadrature) or a selective-deviatoric integration rule:

```
FORMULATION = <string>MEAN_QUADRATURE|SELECTIVE_DEVIATORIC
  (MEAN_QUADRATURE)
```

The selective-deviatoric rule is a higher-order integration scheme, which is discussed below.

If the user wishes to use the selective-deviatoric rule, the `DEVIATORIC PARAMETER` command line must also appear in the `MEMBRANE SECTION` command block:

```
DEVIATORIC PARAMETER = <real>deviatoric_param
```

The selective-deviatoric parameter, `deviatoric_param`, which is valid from 0.0 to 1.0, indicates how much of the deviatoric response should be taken from a uniform-gradient integration and how much should be taken from a full integration of the element. A value of 0.0 will give a pure uniform-gradient response with no hourglass control. Thus, this value is of little practical use. A value of 1.0 will give a fully integrated deviatoric response. Although any value between 0.0 and 1.0 is perfectly valid, lower values are generally preferred.

The selective-deviatoric elements, when used with a parameter greater than 0.0, provide hourglass control without artificial hourglass parameters.

The command line

```
LOFTING FACTOR = <real>lofting_factor(0.5)
```

allows the user to shift the location of the midsurface of a membrane element relative to the geometric location of the membrane element. By default, the geometric location of a membrane element in a mesh represents the midsurface of the membrane. If a membrane has a thickness of 0.2 inch, the top surface of the membrane is 0.1 inch above the geometric surface defined by the membrane element, and the bottom surface of the membrane is 0.1 inch below the geometric surface defined by the membrane element. (The top surface of the membrane is the surface with a positive element normal; the bottom surface of the membrane is the surface with a negative element normal.)

Figure 5.2, which shows lofting for shells, is also applicable to membranes. For membranes, Figure 5.2 represents an edge-on view of membrane elements with a thickness of t and the location of the geometric plane in relation to the membrane surfaces for three different values of the lofting factor—0.0, 0.5, and 1.0. If you want to have the geometric surface defined by the membrane correspond to the top surface of the membrane element, set the lofting factor to 1.0. If you want to have the geometric surface defined by the membrane correspond to the bottom surface of the membrane element, set the lofting factor to 0.0. The geometric surface is midway between the top and bottom surfaces for a lofting factor of 0.5. Note that the default for this factor is 0.5.

Suppose that the lofting factor is set to 1.0 and the thickness of the membrane is 0.2 inch. Let us measure distances to the membrane surfaces (top and bottom) by measuring along the positive element normal. The top surface of the membrane will be located at a distance of 0.0 inch from the geometric membrane surface, and the bottom surface of the membrane will be located at a distance of –0.2 inch from the geometric membrane surface.

Both the membrane mechanics and contact use membrane lofting. See Section 7.2 for a discussion of lofting surfaces for membranes and contact.

5.2.4 Beam Section

```
BEGIN BEAM SECTION <string>beam_section_name
  SECTION = <string>ROD|TUBE|BAR|BOX|I
  WIDTH = <real>section_width
  HEIGHT = <real>section_width
  WALL THICKNESS = <real>wall_thickness
  FLANGE THICKNESS = <real>flange_thickness
  T AXIS = <real>tx <real>ty <real>tz (0 0 1)
  REFERENCE AXIS = <string>CENTER|RIGHT|
    TOP|LEFT|BOTTOM(CENTER)
  AXIS OFFSET = <real>s_offset <real>t_offset
END [BEAM SECTION <string>beam_section_name]
```

The `BEAM SECTION` command block is used to specify the properties for a beam element. If this command block is referenced in an element block of three-dimensional, two-node elements, the elements in the block will be treated as beam elements. The parameter, `beam_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

Five different cross sections can be specified for the beam—`ROD`, `TUBE`, `BAR`, `BOX`, and `I`—via use of the `SECTION` command line. Each section requires a specific set of command lines for a complete geometric description. The command lines related to section geometry are `WIDTH`, `HEIGHT`, `WALL THICKNESS`, and `FLANGE THICKNESS`. We present a summary of the geometric parameter command lines required for each section as a quick reference.

- If the section is `ROD`, the following geometry command lines are required:

  ```
  WIDTH = <real>section_width
  HEIGHT = <real>section_width
  ```

- If the section is `TUBE`, the following geometry command lines are required:

  ```
  WIDTH = <real>section_width
  HEIGHT = <real>section_width
  WALL THICKNESS = <real>wall_thickness
  ```

- If the section is `BAR`, the following geometry command lines are required:

  ```
  WIDTH = <real>section_width
  HEIGHT = <real>section_width
  ```

- If the section is `BOX`, the following geometry command lines are required:

```
            WIDTH = <real>section_width
            HEIGHT = <real>section_width
            WALL THICKNESS = <real>wall_thickness
```

- If the section is `I`, the following geometry command lines are required:

```
            WIDTH = <real>section_width
            HEIGHT = <real>section_width
            WALL THICKNESS = <real>wall_thickness
            FLANGE THICKNESS = <real>flange_thickness
```

All the sections require the `T AXIS` command line. The `REFERENCE AXIS` and `AXIS OFFSET` command lines are optional.

Before presenting details about the various sections, we will discuss the local coordinate system for the beam. (The geometric properties are related to this local coordinate system.) For the beam, it is necessary to specify a local Cartesian coordinate system, which will be designated as r, s, and t. The r-axis lies along the length of the beam and passes through the centroid of the beam. The t-axis is specified by the user. The initial direction of the t-axis is specified by a vector in the global coordinate system. If we want the initial position of the t-axis to be parallel to the global Z-axis, then we would use the command line

```
    T AXIS = 0 0 1 .
```

If we wanted the initial position of the t-axis to be parallel to a vector (0.5, 0.8660, 0) in the global coordinate system, then we would use the command line

```
    T AXIS = 0.5 0.8660 0.0 .
```

The t-axis will change position as the beam deforms (rotates about the r-axis). This change in position is tracked internally by the computations for the beam element. The s-axis is computed from the cross-product of the t-axis and the r-axis. The `HEIGHT` for the beam cross section is in the direction of the t-axis, and the `WIDTH` of the beam cross section is in the direction of the s-axis.

Now that the local coordinate system for the beam has been defined, we can describe the definition of each section.

- The `ROD` section is a solid elliptical section. The diameter along the height is specified by the `HEIGHT` command line, and the diameter along the width is specified by the `WIDTH` command line.

- The `TUBE` section is a hollow elliptical section. The diameter along the height is specified by the `HEIGHT` command line, and the diameter along the width is specified by the `WIDTH` command line. The wall thickness for the tube is specified by the `WALL THICKNESS` command line.

- The `BAR` section is a solid rectangular section. The height is specified by the `HEIGHT` command line, and the width is specified by the `WIDTH` command line.
- The `BOX` section is a hollow rectangular section. The height is specified by the `HEIGHT` command line, and the width is specified by the `WIDTH` command line. The wall thickness for the box is specified by the `WALL THICKNESS` command line.
- The `I` section is the standard I-section associated with a beam. The height of the I-section is given by the `HEIGHT` command line, and the width of the flanges is given by the `WIDTH` command line. The thickness of the vertical member is given by the `WALL THICKNESS` command line, and the thickness of the flanges is given by the `FLANGE THICKNESS` command line.

By default, the r-axis coincides with the geometric centerline of the beam. The geometric centerline of the beam is defined by the location of the two nodes defining the beam connectivity. It is possible to offset the local r-axis, s-axis, and t-axis from the geometric centerline of the beam. To do this, one can use either the `REFERENCE AXIS` command line or the `AXIS OFFSET` command line, but not both.

The `REFERENCE AXIS` command line has the options `CENTER`, `TOP`, `RIGHT`, `BOTTOM`, and `LEFT`. The `CENTER` option is the default, which means that the r-axis coincides with the geometric centerline of the beam. If the `TOP` option is used, the r-axis is moved in the direction of the original t-axis by a positive distance `HEIGHT/2` from the centroid so that it passes through the top of the beam section (top being defined in the direction of the positive t-axis). If the `RIGHT` option is used, the r-axis is moved in the direction of the original s-axis by a positive distance `WIDTH/2` so that it passes through the right side of the beam section (the section being viewed in the direction of the negative r-axis). If the `BOTTOM` option is used, the r-axis is moved in the direction of the original t-axis by a distance `HEIGHT/2` so that it passes through the bottom of the beam section (bottom being defined in the direction of the negative t-axis). If the `LEFT` option is used, the r-axis is moved in the direction of the original s-axis by a negative distance `WIDTH/2` so that it passes through the left side of the beam section (the section being viewed in the direction of the negative r-axis). For all options, the s-axis and the t-axis remain parallel to their original positions before the translation of the r-axis.

The `AXIS OFFSET` command line allows the user to offset the local coordinate system from the geometric centerline by an arbitrary distance. The first parameter on the command line moves the r-axis a distance `s_offset` from the centroid of the section along the original s-axis. The second parameter on the command line moves the r-axis a distance `t_offset` from the centroid of the section along the original t-axis. The s-axis and t-axis remain parallel to their original positions before the translation of the r-axis.

Strains and stresses are computed at the midpoint of the beam. The integration of the stresses over the cross section at the midpoint is used to compute the internal forces in the beam. Each beam section has its own integration scheme. The integration scheme for each of the sections is shown in Figure 5.5 through Figure 5.7. The numbers in these figures show the relative location of the integration points in regard to the centroid of the section and the s-axis and the t-axis.

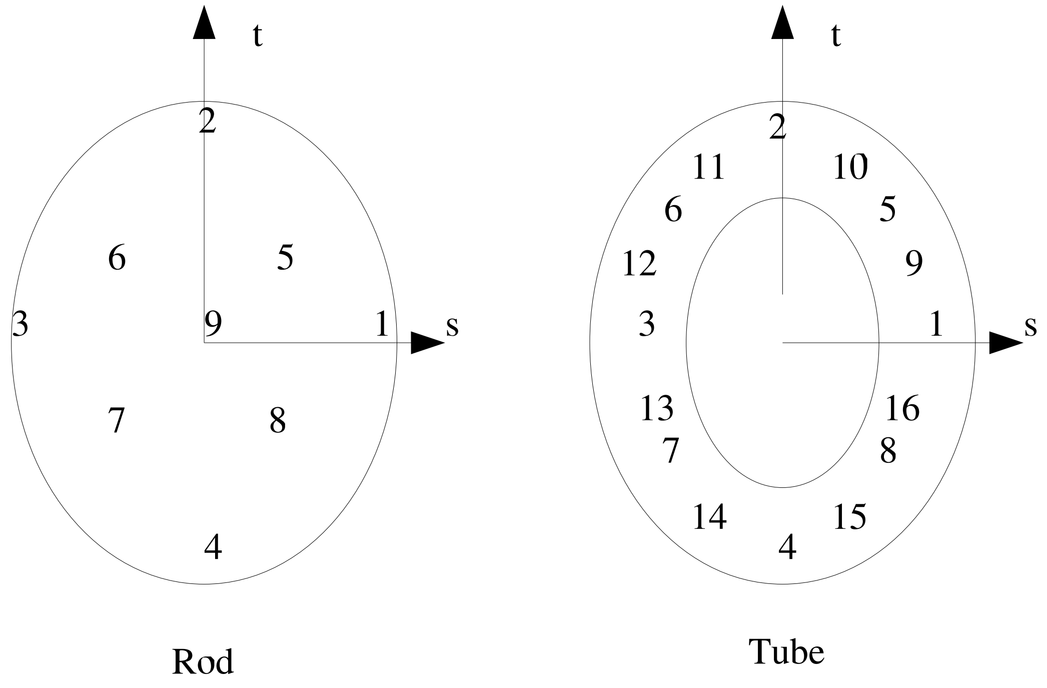


Figure 5.5: Integration points for rod and tube.

At each integration point, there is an axial strain (with a corresponding axial stress) and an in-plane (in the plane of the cross section) shear strain (with a corresponding shear stress). The user can output this stress and strain information by using the `RESULTS OUTPUT` commands described in Chapter 8. The registered variable that will let users access the strain at the beam integration points is `beam_strain_inc`, and the registered variable that will let users access the stress at the beam integration points is `beam_stress`. If the user requests output for the beam strain, 32 values are given for the strain. The first value (designated in the output as 01) is the axial strain at the first integration point, the second value (designated in the output as 02) is the shear strain at the first integration point, etc. The odd values for the strain output (01, 03, 05, etc.) are the axial strains at the integration points. The even values of the strain output (02, 04, 06, etc.) are the shear strains at the integration points. For the case where there are only nine integration points (the rod), only the

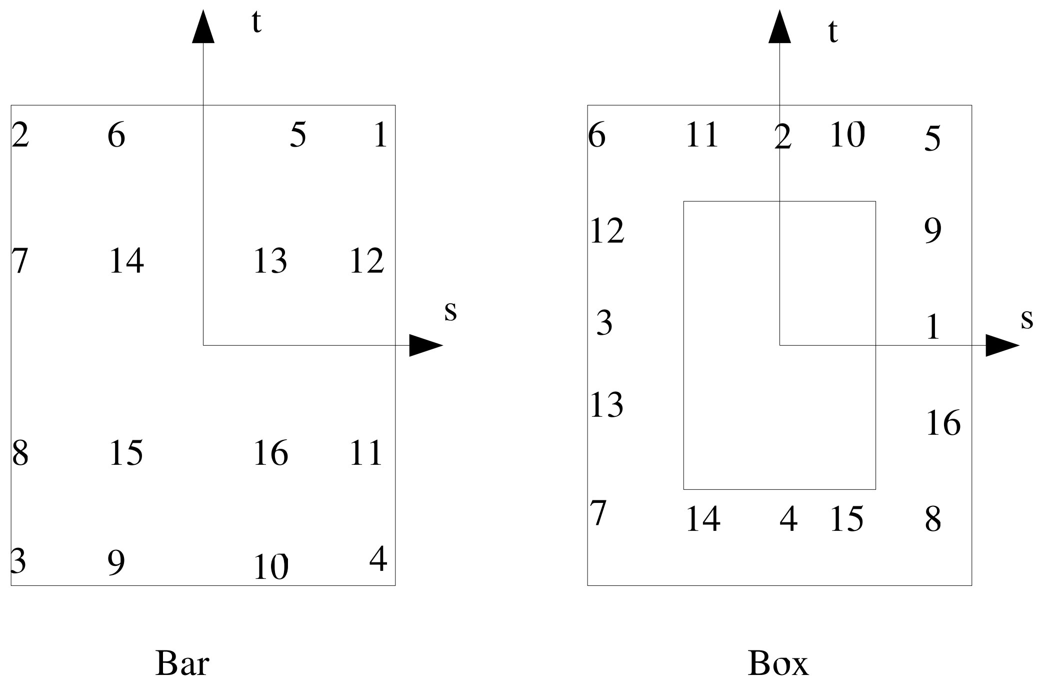


Figure 5.6: Integration points for bar and box.

first 18 values for strain have any meaning for the section (the values 19 through 32 are zero). For the I-section, only the first 30 of the strain values have meaning since this section only has 15 integration points. For all other sections, all 32 values have meaning. A pattern similar to that for the strains holds for stresses.

As as alternative for the stress output, you may use the registered variables `beam_stress_axial` and `beam_stress_shear`. The variable `beam_stress_axial` contains only the axial stresses. The first value associated with `beam_stress_axial` (designated as 01) corresponds to the axial stress at integration point 1, the second value associated with `beam_stress_axial` (designated as 02) corresponds to the axial stress at integration point 2, and so on. The variable `beam_stress_shear` contains only shear stresses. The correlation between numbering the values for `beam_stress_shear` (01, 02, . . .) and the integration points is the same as for `beam_stress_axial`.

It is possible to access mean values for the internal forces at the midpoint of the beam. The axial force at the midpoint of the beam is obtained by referencing the registered variable `beam_axial_force`. The transverse forces at the midpoint of the beam in the s-direction and the t-direction are obtained by referencing `beam_transverse_force_s` and `beam_transverse_force_t`, respectively. The

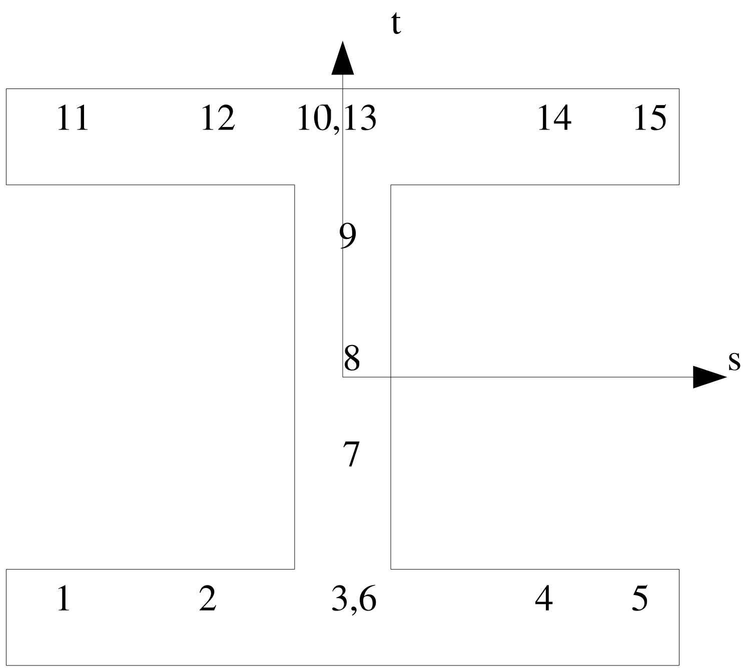


Figure 5.7: Integration points for I-section.

torsion at the midpoint of the beam (the moment about the r-axis), is obtained by referencing `beam_moment_r`. The moments about the s-axis and the t-axis are obtained by referencing `beam_moment_s` and `beam_moment_t`, respectively.

5.2.5 Truss Section

```
BEGIN TRUSS SECTION <string>truss_section_name
  AREA = <real>cross_sectional_area
  INITIAL LOAD = <real>initial_load
  PERIOD = <real>period
END [TRUSS SECTION <string>truss_section_name]
```

The `TRUSS SECTION` command block is used to specify the properties for a truss element. If this command block is referenced in an element block of three-dimensional, two-node elements, the elements in the block will be treated as truss elements. The parameter, `truss_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The cross-sectional area for truss elements is specified by the `AREA` command line. The value `cross_sectional_area` is the cross-sectional area of the truss members in the element block.

The truss can be given some initial load over some given time period. The magnitude of the load is specified by the `INITIAL LOAD` command line. If the load is compressive, the sign on the value `initial_load` should be negative; if the load is tensile, the sign on the value `initial_value` should be positive. The period is specified by the `PERIOD` command line.

The initial load is applied over some period by specifying the axial strain rate in the truss, $\dot{\varepsilon}$, over some period p. At some given time t, the strain rate is

$$\dot{\varepsilon} = \frac{ap}{2}\left[1 - \cos\left(\pi t/p\right)\right], \tag{5.1}$$

where

$$a = \frac{2F_i}{EAp}. \tag{5.2}$$

In Equation (5.2), F_i is the initial load, E is the modulus of elasticity for the truss, and A is the area of the truss. Over the period p, the total strain increment generates the desired initial load in the truss.

During the initial load period, the time increments should be reasonably small so that the integration of $\dot{\varepsilon}$ over the period is accurate. The period should be set long enough so that if the model was held in a steady state after time p, there would only be a small amount of oscillation in the load in the truss.

When doing an analysis, you may not want to activate certain boundary conditions until after the prestressing is done. During the prestressing, time-independent boundary conditions such as fixed displacement will most likely be turned on. Time-dependent boundary conditions such as prescribed acceleration or prescribed force will most likely be activated after the prestressing is complete.

5.2.6 Damper Section

```
BEGIN DAMPER SECTION <string>damper_section_name
  AREA = <real>damper_cross_sectional_area
END [DAMPER SECTION <string>damper_section_name]
```

The `DAMPER SECTION` command block is used to specify the properties for a damper element. If this command block is referenced in an element block of three-dimensional, two-node elements, the elements in the block will be treated as damper elements. The parameter, `damper_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The cross-sectional area for damper elements is specified by the `DAMPER AREA` command line. The value `damper_cross_sectional_area` is the cross-sectional area of the dampers in the element block.

The damper area is used only to generate mass associated with the damper element. The mass is the density for the damper element multiplied by the original volume of the element (original length multiplied by the damper area).

The force generated by the damper element depends on the relative velocity along the current direction vector for the damper element. If $\boldsymbol{n}$ is a unit normal pointing in the direction from node 1 to node 2, if $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are the velocity vectors at nodes 1 and 2, respectively, then the force generated by the damper element is

$$F_d = \eta \mathbf{n} \cdot (\boldsymbol{v}_2 - \boldsymbol{v}_1) , \tag{5.3}$$

where η is the damping parameter. Currently, the damping parameter must be specified by using an elastic material model for the damper element. The value for Young's modulus in the elastic material model is used for the damping parameter η.

5.2.7 Point Mass Section

```
BEGIN POINT MASS SECTION <string>pointmass_section_name
  VOLUME = <real>volume
END [POINT MASS SECTION <string>pointmass_section_name]
```

A point mass element is simply a mass at a node, which can be a convenient modeling tool in certain instances. The user can create an element block with one or more point masses. Each point mass must be associated with an existing node. A point mass will have its mass added to the mass at the node. (Other mass at the node will be derived from mass due to elements attached to the node.) The mass at a node due to a point mass is treated like any other mass at a node derived from an element. The mass due to point mass will be included in body force calculations and kinetic energy calculations, for example.

Point masses are a convenient modeling tool to be used in conjunction with rigid bodies. An element block including one or more point masses can be included like any other element block in a collection of element blocks used to define a rigid body. The element block of point masses can be used to adjust the total mass and inertia properties for the rigid body. (The point mass element does not have to be used only in conjunction with rigid bodies. One can place a point mass at a node associated with solid or structural elements.)

If you have an element block in which the connectivity for each element is only one node, then you may use this element block as a collection of point masses. This command block would have the following form:

```
BEGIN PARAMETERS FOR BLOCK <string>block_id
  MATERIAL = <string>material_name
  SOLID MECHANICS USE MODEL <string> material_model_name
  SECTION = <string>point_mass_section_name
END PARAMETERS FOR BLOCK <string>block_id
```

The element block associated with our point mass must reference a material command block just like any other element block. The product of the density specified in the material block and the volume specified in the section block (for the point mass) will be taken as the mass of each point mass in the element block. Suppose, for example, you have the following `PARAMETERS FOR BLOCK` command block for an element block with point masses:

```
BEGIN PARAMETERS FOR BLOCK block_105
  MATERIAL = mass_for_pointmass
  SOLID MECHANICS USE MODEL elastic
  SECTION = pmass_1
END PARAMETERS FOR BLOCK block_105
```

The above element block for the point mass elements references the material `mass_for_pointmass`. Let the density associated with this material be 0.0153. Let the value of the volume parameter associated with the section block `pmass_1` be 4.0. Each point mass in the element block will have a mass of $4.0 \times 0.0153 = 0.0612$.

If you have access to the SEACAS codes from Sandia National Laboratories, you may use the codes in this library to generate element blocks with point mass elements. See Reference 10 for an overview of the SEACAS codes. By using various SEACAS codes, you can easily generate an element block with one or more point masses. For each point mass in the element block, you should create an eight-node hexahedral element that is centered at the point where you want the point mass located. The hexahedron can have arbitrary dimensions, but it is best to work with a unit cube. Suppose you wanted a point mass at (13.5, 27.0, 3.1415). You could create a unit hexahedron (1 by 1 by 1) centered on (13.5, 27.0, 3.1415). The SEACAS program SPHGEN3D will convert the hexahedron to a zero-dimensional element in three-dimensional space located at the center of the hexahedron. For our specific example, the program SPHGEN3D would create a zero-dimensional element, basically an element consisting of a single node, at (13.5, 27.0, 3.1415).

5.2.8 SPH Section

```
BEGIN SPH SECTION <string>sph_section_name
  RADIUS MESH VARIABLE = <string>var_name|<string>attribute|
    SPHERE INITIAL RADIUS = <real>rad
  RADIUS MESH VARIABLE TIME STEP = <string>time
  PROBLEM DIMENSION = <integer>1|2|3(3)
  CONSTANT SPHERE RADIUS
END [SPH SECTION <string>sph_section_name]
```

SPH (smoothed particle hydrodynamics) is useful for modeling fluids or for modeling materials that undergo extremely large distortions. One must be careful when using SPH for modeling. SPH tends to exhibit both accuracy and stability problems, particularly in tension. An SPH particle interacts with other nearest-neighbor SPH particles based on radius properties of all the elements involved; SPH particles react with other elements, such as tetrahedra, hexahedra, and shells, through contact. You should consult Reference 9 regarding the theoretical background for SPH.

All the particles contained in an SPH element block must be given some initial radius. There are two options for setting the initial radius for each particle. First, each particle can be given the same radius. To set the radius for each particle in an element block to the same value, use the `SPHERE INITIAL RADIUS` command line. The parameter `rad` on this command line sets the radius for all the SPH particles in the element block. Second, the radius for each particle can be read from a mesh file. The radii can be read from a variable on the mesh file as the attributes associated with the element block. If you want to read some variable from the mesh file for the radii, then you would use

```
RADIUS MESH VARIABLE = sph_radius ,
```

where `sph_radius` is the variable name on the mesh file. If you want to use the variable associated with a specific time on the mesh file, you should use the `RADIUS MESH VARIABLE TIME STEP` command line to select the specific time. If you want to read the attributes associated with the particles, then you should insert the command line

```
RADIUS MESH VARIABLE = attribute
```

(as shown) into the `SPH SECTION` command block. Pronto3d [1] only offers the attribute option. To compare Presto and Pronto3d results, you should use the attribute option.

Once SPH determines the initial radius (through either the `SPHERE INITIAL RADIUS` command line or the `RADIUS MESH VARIABLE` command line), it will recalculate the optimal radius for the particle. The initial radii must be such that each sphere will overlap at least a few other SPH elements. If the initial radii are too small, the optimal radius calculation will fail, and the particles will not interact. If the initial radii are too large, many interactions may need to be checked, and the initialization calculation for the optimal radius step may take a long time.

After the radii are initialized, you may determine whether the radii are to remain constant or are to change throughout the analysis. The `CONSTANT SPHERE RADIUS` command line is an optional command line that prevents the sphere radius from changing over the course of the calculation. By default, the sphere radii will expand or contact based on the changing density in the elements to satisfy the relation that element mass (a constant) equals element volume times element density. If the `CONSTANT SPHERE RADIUS` command line appears, then the radii for all particles will remain constant.

Your analysis problem using SPH may be inherently one-, two-, or three-dimensional. You may indicate whether or not there is some inherent dimensionality in the problem by using the `PROBLEM DIMENSION` command line. The possible value for this command line is 1 (one-dimensional), 2 (two-dimensional), or 3 (three-dimensional). The default value is 3 for three-dimensional. The internal SPH calculations are modified depending on the value set on the `PROBLEM DIMENSION` command line. If, for example, your problem is inherently two-dimensional in nature, you may get more accurate results by specifying the dimension for your problem as 2 (as opposed to 1 or 3).

Utility Commands. In addition to the SPH-related command lines just described (which appear in the `SPH SECTION` command block) there are two other SPH-related command lines:

```
SPH SYMMETRY PLANE <string>+X|+Y|+Z|-X|-Y|-Z
  <real>position_on_axis(0.0)
SPH DECOUPLE STRAINS: <string>material1 <string>material2
```

If either one or both of these command lines are used, they should be placed directly into the domain scope. (All other SPH-related command lines must be nested within the `SPH SECTION` command block; the `SPH SECTION` command block, like all other section command blocks, is in the domain scope.) The symmetry conditions are applied to all SPH element blocks.

The `SPH SYMMETRY PLANE` command line may be used to reduce model sizes by specifying symmetry planes and modeling only a portion of the model. Due to the nonlocal nature of SPH element integration, symmetry planes cannot be defined with boundary conditions alone; these planes must be explicitly defined. A plane is defined by an outward normal vector aligned with one of the axes (`+X`, `+Y`, `+Z`, `-X`, `-Y`, `-Z`) and some point on the axis, which represents a point in the plane. Suppose for example, the outward normal to the plane of symmetry is in the negative Y-direction (`-Y`) and the plane of symmetry passes through the y-axis at $y = +2.56$. Then the definition for the symmetry plane would be

```
SPH SYMMETRY PLANE -Y +2.56 .
```

The `SPH DECOUPLE STRAINS` command line prevents two dissimilar materials from directly interacting. Generally, the material properties at a particle are the average of the material properties from nearby particles. If particles with very dissimilar material properties are interacting, this interaction can create problems. The `SPH DECOUPLE STRAINS` command line ensures that particles with very dissimilar material properties do not directly interact by material-property averaging, but instead just interact with a contact-like interaction. The two material types that are not to interact are specified by the parameters `material1` and `material2`. These parameters will appear as material names on a `PROPERTY SPECIFICATION FOR MATERIAL` command block.

Display. For purposes of visualizing the element stresses, it may be necessary to copy these element variables into nodal variables. This can easily be done by defining a `USER VARIABLE` command block (Section 9.2.4) in conjunction with a `USER OUTPUT` command block (Section 8.1.2). Once the nodal variable is defined, it can be output in a `RESULTS OUTPUT` command block (Section 8.1.1). An example is provided below. The `SPH` element blocks for the problem are element blocks 20, 21, and 22. All other element blocks are non-SPH elements.

- In the domain scope:

```
BEGIN USER VARIABLE nodal_stress
  TYPE = NODE SYM_TENSOR LENGTH = 1
END
```

- In the region scope:

```
BEGIN USER OUTPUT
  BLOCK = block_20 block_21 block_22
  COPY ELEMENT VARIABLE rotated_stress TO NODAL VARIABLE
    nodal_stress
  END

  BEGIN RESULTS OUTPUT output_presto
    DATABASE NAME = sph.e
    DATABASE TYPE = exodusII
    AT TIME 0.0 INCREMENT = 1.0e-04
    NODAL VARIABLES = nodal_stress
  END RESULTS OUTPUT output_presto
```

5.3 Element-like Functionality

This section describes functionality in Presto that resembles the previously described elements to some extent. This functionality is not really implemented using the element structure in Presto, however. The functionality described in this section—the torsional spring mechanism and rigid bodies—is specified through command blocks that appear in a Presto region.

5.3.1 Torsional Spring Mechanism

```
BEGIN TORSIONAL SPRING MECHANISM <string>spring_name
  NODE SETS = <string>nodelist_int1 <string>nodelist_int2
    <string>nodelist_int3 <string>nodelist_int4
  TORSIONAL STIFFNESS = <real>stiffness
  INITIAL TORQUE = <real>init_load
  PERIOD = <real>time_period
  ACTIVE PERIODS = <string list>period_names
END [TORSIONAL SPRING MECHANISM <string>spring_name]
```

This feature was originally implemented to model a torsional spring wrapped around a fixed pin. One end of the pin is fixed to a base, and one end of the spring is attached to this base. There is an arm on the other end of the pin, and this arm can rotate around the pin. The second end of the spring is attached to this arm. The spring resists motion of the arm. Any similar mechanism can be modeled with the torsional spring. Although the torsional spring is element-like in its overall behavior, its implementation within the code structure is different from the other elements described in Chapter 5. The torsional spring does not make use of a section, and its command block (`TORSIONAL SPRING MECHANISM`) should appear in the region scope. A schematic for the torsional spring mechanism is shown in Figure 5.8.

The mechanism consists of two nodes that represent the axis of a torsional spring. Node 0 is at the base of the torsional spring, and node 1 is at the top of the torsional spring. A third node, reference node 0, defines an arm extending from the axis of the torsional spring to some attachment point near the base of the spring. A fourth node, reference node 1, defines an arm extending from the axis of the torsional spring to some attachment point near the top of the spring. The rotation of the two arms relative to each other as measured along the axis of the torsional spring represents the angular deformation of the spring and determines the moment in the spring. The moment in the spring is translated into external forces at the two attachment points, reference node 0 and reference node 1.

In the `TORSIONAL SPRING MECHANISM` command block, the string `spring_name` is defined by the user. Via the `NODE SETS` command line, the mechanism is de-

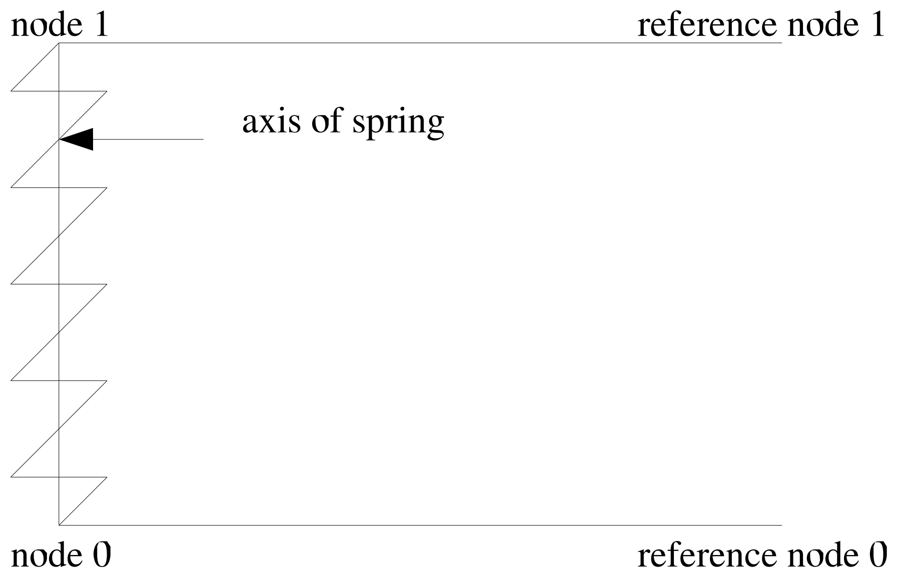


Figure 5.8: Schematic for torsional spring.

fined with four node sets, and each node set has a single node. The first set (`nodelist_int1`) defines node 0 in Figure 5.8; node 0 is the origin of a local coordinate system for the torsional spring mechanism. The second node set (`nodelist_int2`) defines node 1 in Figure 5.8; node 0 and node 1 define the axis of the torsional spring mechanism. The third node set (`nodelist_int3`) defines reference node 0; reference node 0 is an attachment point for the spring associated with node 0. The fourth node set (`node_int4`) defines reference node 1; reference node 1 is an attachment point for the spring associated with node 1.

The nodes defining the spring mechanism are used to set up a local coordinate system (x', y', z'). The (z'-axis runs along the axis of the spring from node 0 to node 1. The x'-axis extends from the axis of the spring and passes through reference node 0. If we are looking down the axis of the spring in the negative z'-direction, a positive rotation of the arm defined by node 1 and reference node 1 is in the counterclockwise direction. This is shown in Figure 5.9.

The torque, T, in the spring is simply

$$T = K\theta, \tag{5.4}$$

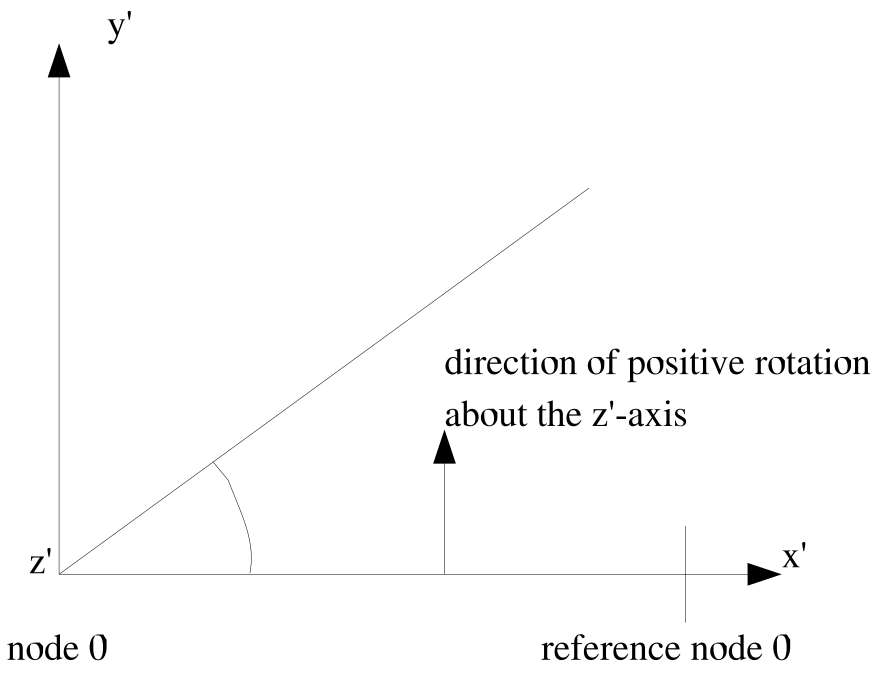


Figure 5.9: Positive direction of rotation for torsional spring.

where K is the torsional stiffness and θ is the rotation of the top arm relative to the bottom arm as measured along the axis of the spring. In the `TORSIONAL STIFFNESS` command line, K is specified by the real value `stiffness`.

The torque in the spring is converted to external forces with components in the global coordinate system XYZ. These external forces depend on the torque and the length of the spring arms. The length of the spring arms is automatically calculated.

You can apply an initial torque with the real value `init_load` in the `INITIAL TORQUE` command line. The maximum value of the torque is reached in the time specified by the real value `time_period` in the `PERIOD` command line.

The initial torque is applied over some period by specifying the angular rate of deformation in the torsional spring, $\dot{\theta}$, over some period p. At some given time t, the angular rate of deformation is

$$\dot{\theta} = \frac{ap}{2}\left[1 - \cos\left(\pi t/p\right)\right], \tag{5.5}$$

where

$$a = \frac{2T_i}{Kp}. \tag{5.6}$$

In Equation (5.6), T_i is the initial torque. Over the period p, the total strain increment generates the desired initial load in the truss.

During the initial load period, the time increments should be reasonably small so that the integration of $\dot{\theta}$ over the period is accurate. The period should be set long enough so that if the model was held in a steady state after time p, there would be only a small amount of oscillation in the load in the torsional spring.

When doing an analysis, you may not want to activate certain boundary conditions until after the prestressing is done. During the prestressing, time-independent boundary conditions such as fixed displacement will most likely be turned on. Time-dependent boundary conditions such as prescribed acceleration or prescribed force will most likely be activated after the prestressing is complete.

You can output the torque in the spring, the total rotation, and the last angle between the arms. The name specified on the command block is used to construct parameters for the mechanism. Suppose the input line is

```
begin torsional spring mechanism lower_spring .
```

where `lower_spring` is a user-specified name. The code will automatically generate the parameters `TS_lower_spring_MOMENT`, `TS_lower_spring_ROTATION`, and `TS_lower_spring_LAST_ANGLE`. These variables can then be output in a results file. For example, one could use

```
global variables = TS_lower_spring_MOMENT as ts_lspring_m
global variables = TS_lower_spring_ROTATION as ts_lspring_r
global variables = TS_lower_spring_LAST_ANGLE as
  ts_lspring_la
```

in the `RESULTS OUTPUT` command block. If several torsional spring mechanisms appear in one model, you can generate unique names to keep track of the parameters associated with each spring. See Section 8.1 for further information about results output.

The `ACTIVE PERIODS` command line determines when the torsional spring is active. See Section 2.5 for more information about this command line. Although the active periods option is available in the `TORSIONAL SPRING` command block, use of this option to turn the torsional spring off and on repeatedly is not recommended. Turning the torsional spring off and on repeatedly may lead to erroneous behavior in the spring model.

5.3.2 Rigid Body

```
BEGIN RIGID BODY <string>rb_name
  POINT INERTIA = <real>Ixx <real>Iyy <real>Izz <real>Ixy
    <real>Iyz <real>Izx
  MAGNITUDE = <real>magnitude_of_velocity
  DIRECTION = <string>direction_definition
  ANGULAR VELOCITY = <real>omega
  CYLINDRICAL AXIS = <string>axis_definition
END [RIGID BODY <string>rb_name]
```

A rigid body can consist of any combination of elements—solid elements, structural elements, and point masses—except SPH elements. All nodes associated with a rigid body maintain their same relative position to each other as determined at time 0 when there is no deformation of the body. This means that the elements associated with the rigid body do not deform over time. These elements are free to move (rotate and translate) through space, but they cannot deform. Element blocks that are part of a rigid body can adjoin deformable element blocks. For any rigid body consisting of several element blocks, the element blocks defining the rigid body do not have to be contiguous. You may have more than one rigid body in a model.

If you construct a model where all the element blocks compose a rigid body, you will need to set an initial time in the `TIME CONTROL` command block (Section 3.1.1). Include the line

```
INITIAL TIME STEP = 1.0e-6 .
```

If an element block is declared to be a part of a rigid body, the internal force calculations are not called for the elements in that block. Part of the internal force calculation for an element is an element time-step estimate. If all elements are in a rigid body, the element time-step computations are not performed, and there is no estimate for a global time step. You must supply an initial time step only if all the elements are part of a rigid body. If some elements are in a rigid body, but others are not, then you will automatically obtain a valid time step estimate for the problem. If you must set an initial time step for your problem because all elements are in a rigid body, then you should not override the default value of 1.0 for the time step scale factor (see Section 3.1.1). The time step you set for this particular case (all elements in a rigid body) must remain constant. The value of 1.0×10^{-6} should work well for most problems. Do not use an initial time step larger than 1.0×10^{-6} as this could cause loss of accuracy in the solution of the problem.

To construct the rigid body, you will need to use the above command block, which appears in the region scope, plus the `RIGID BODY` command line that appears in the `PARAMETERS FOR BLOCK` command block for an element block. Suppose, for example, `rigidbody_1` consists of element blocks 100, 110, and 280. The `PARAMETERS`

`FOR BLOCK` command blocks for element blocks 100, 110, and 280 must all contain the command line

```
    RIGID BODY = rigidbody_1 .
```

Once you have declared an element block or some collection of element blocks to be a rigid body and created a rigid body name, that rigid body name must appear as the name in a `RIGID BODY` command block. In our example, we must have a `RIGID BODY` command block with the value for `rb_name` set to `rigidbody_1`. Therefore, at a minimum, you must have a command block in the region with the form

```
    BEGIN RIGID BODY rigidbody_1
    END RIGID BODY rigidbody_1
```

for our example.

The `RIGID BODY` command block has several different command lines, composing essentially three groups of commands. One group consists of the `POINT INERTIA` command line, a second group consists of the paired `MAGNITUDE` and `DIRECTION` command lines, and a third group consists of the paired `ANGULAR VELOCITY` and `CYLINDRICAL AXIS` command lines. Each of the three groups is optional. You can combine any of these groups in the command block, or you could have a command block that contains none of the command lines, whereupon you would only supply the value for `rb_name` in the block.

Input to the `POINT INERTIA` command line consists of six real numbers that define the moments (`Ixx`, `Iyy`, `Izz`) and products (`Ixy`, `Iyx`, `Izx`) of the inertia referenced to the center of mass of the rigid body. For the rigid body, the center of mass is first calculated from the element masses for all elements that define the rigid body. Moments and products of inertia are then computed for the rigid body based on the location of the center of mass of the rigid body and the element masses for all the elements associated with the rigid body. The moments and products of inertia specified in the `POINT INERTIA` command line are then added to the products and moments computed from the elements masses. This modified inertia tensor (rather than the inertia tensor based solely on element mass) is then used to calculate the motion of the rigid body.

If the rigid body has an initial velocity in some direction, this should be specified with the `MAGNITUDE` and `DIRECTION` command lines. The `MAGNITUDE` command line gives the magnitude of the initial velocity applied to the center of mass of the rigid body, and the `DIRECTION` command line gives a defined direction for the initial velocity for the center of mass. All blocks associated with the rigid body should be given the same initial velocity by using an `INITIAL VELOCITY` command block. (The information in the `RIGID BODY` command block is only applied to the center of mass of the rigid body.)

If the rigid body has an initial rotation about some axis, this should be specified with the `ANGULAR VELOCITY` and `CYLINDRICAL AXIS` command lines. The `ANGULAR VELOCITY` command line gives the initial velocity of the center of mass of the rigid body due to an angular velocity about some defined axis given on the `CYLINDRICAL AXIS` command line. All blocks associated with the rigid body should be given the same initial angular velocity by using an `INITIAL VELOCITY` command block. (The information in the `RIGID BODY` command block is only applied to the center of mass of the rigid body.)

You can output the acceleration, velocity, and displacement for the center of mass of the rigid body. The name assigned to a rigid body will be used to construct registered variable names that give the above quantities. This lets you identify the output associated with a rigid body based on the name you assigned for the rigid body.

Immediately before the results file is written, the accelerations for nodes associated with a rigid body are updated to reflect the accelerations due to the rigid-body constraints. This ensures that the accelerations sent to the results output are correct for a given time.

In summary, if you use a rigid body in an analysis, you will do one or more of the following steps:

- Create a rigid body using one or more element blocks (except SPH element blocks). A `RIGID BODY` command line must appear in the `PARAMETERS FOR BLOCK` command block for any element block associated with a rigid body.
- Include point mass element blocks with the rigid body if appropriate. A `RIGID BODY` command line must appear in the `PARAMETERS FOR BLOCK` command block for any point mass element block associated with a rigid body.
- Include a `RIGID BODY` command block in the region.
- Associate an initial velocity or initial rotation about an axis with the rigid body. If any of the blocks associated with a rigid body have been given an initial velocity or initial rotation, the rigid body must have the same specification for the initial velocity or initial rotation.
- Output center-of-mass information for the rigid body.

The above steps involve a number of different command blocks. To demonstrate how to fully implement a rigid body, we will provide a specific example that exercises the various options available to a user.

Let us assume that we want to create a rigid body named `part_a` consisting of three element blocks. Two of the element blocks, element block 100 and element block

535, are eight-node hexahedra; one of the element blocks, element block 600, consists of only point masses. The `RIGID BODY` command block and the element blocks we want to associate with the rigid body will be as follows:

```
begin parameters for block block_100
  material steel
  solid mechanics model use elastic
  rigid body = part_a
end
begin parameters for block block_535
  material = aluminum
  solid mechanics model use elastic
  rigid body = part_a
end
begin parameters for block block_600
  material = mass_for_pointmass
  solid mechanics model use elastic
  rigid body = part_a
end
```

To adjust the moments and products of inertia computed by Presto for the rigid body `part_a`, we have to included the `POINT INERTIA` command line in the above section command block for the rigid body.

Now that we have defined the rigid body, we will examine how to specify an initial angular velocity about an axis for the rigid body and how to output information at the center of mass for the rigid body. The center of mass of the rigid body is some computed point associated with the body. It may or may not be at or near any node in the body.

Suppose we want to have the rigid body spin at 600 radians/sec about an axis parallel to the x-axis and passing through a point at (0, 10, 20). We would define this axis using the following set of `DEFINE` command lines:

```
define direction parallel_to_x with vector 2.0 0.0 0.0
define point off_axis with coordinates 0.0 10.0 20.0
define axis body_axis with point off_axis
direction parallel to x
```

The blocks in the rigid body will be given an initial angular velocity of 600 radians/sec about the above axis if we use the following command block for initial angular velocity:

```
begin initial velocity
```

```
    block = block_100 block_535 block_600
    cylindrical axis = body_axis
    angular velocity = 600
  end initial velocity
```

The RIGID BODY command block must be given the same specification for an initial angular velocity. The angular velocity specification in the RIGID BODY command block is applied to the center of mass of the rigid body to make sure its initial motion is consistent with the initial motion of all the nodes in the rigid body as defined by the INITIAL VELOCITY command block. Our RIGID BODY command block (in the region) will appear as follows:

```
  begin rigid body part_a
    cylindrical axis = body_axis
    angular velocity = 600
  end rigid body part_a
```

Now suppose that we want the acceleration, velocity, and displacement output at the center of mass for the rigid body that we have named part_a. Presto automatically generates registered variables giving this information. The registered variables for acceleration, velocity, and displacement are RB_section_name_ACCL, RB_section_name_VEL, and RB_section_name_DISPL, respectively. In the preceding sentence, the value of "section_name" will be the name you supply for the rigid body in the RIGID BODY SECTION command block. The registered variables can be listed as output in a RESULTS OUTPUT command block. For our specific example, we would have the following:

```
 begin results output out_presto
   database name = model_with_rb.e
   at time 0.0 increment = 5.0e-5
   .
   .
   global variables = RB_part_a_ACCL as part_a_accl
   global variables = RB_part_a_VEL as part_a_vel
   global variables = RB_part_a_DISPL as part_a_displ
 end results output out_presto
```

If you have more than one rigid body in your model, you will be able to keep track of the center-of-mass information based on the section name you give to each rigid body.

5.4 Mass Property Calculations

```
BEGIN MASS PROPERTIES
  # {block set commands}
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  # structure command
  STRUCTURE NAME = <string>structure_name
END [MASS PROPERTIES]
```

Presto automatically gives mass property information for the total model, which consists of all the element blocks. (The mass for the total model, for example, is the total mass of all the element blocks.) Presto also automatically gives mass property information for each element block.

In addition to the mass property information that is generated, Presto gives you the option of defining a structure that represents some combination of element blocks and then of calculating the mass properties for this particular structure. If you wish to define a structure that is a combination of some group of element blocks, you must use the `MASS PROPERTIES` command block. This command block appears in the region scope.

For the total model, each element block, and any user-defined structure, Presto reports the total mass, the center of mass in the global coordinate system XYZ, the moments of inertia (Ixx, Iyy, and Izz), and the products of inertia (Ixy, Iyz, Izx). The moments and products of inertia are in the global coordinate system.

The `MASS PROPERTIES` command block contains two groups of commands—block set and structure. Each of these groups is basically independent of the other. Following are descriptions of the two command groups.

5.4.1 Block Set Commands

The {`block set commands`} portion of the `MASS PROPERTIES` command block defines a set of blocks for which mass properties are being requested, and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of blocks. See Section 6.1 for more information about the use of these command lines for creating a set of blocks used by the command block. There must be at least one `BLOCK` or `INCLUDE ALL BLOCKS` command line in the command block.

The `REMOVE BLOCK` command line allows you to delete blocks from the set specified in the `BLOCK` and/or `INCLUDE ALL BLOCKS` command line(s) through the string list `block_names`. Typically, you will use the `REMOVE BLOCK` command line with the `INCLUDE ALL BLOCKS` command line. If you want to include all but a few of the element blocks, a combination of the `REMOVE BLOCK` command line and `INCLUDE ALL BLOCKS` should minimize input information.

Suppose that only one element block, `block_300`, is specified on the `BLOCK` command line. Then only the mass properties for that block will be calculated. If several element blocks are specified on the `BLOCK` command line, then that collection of blocks will be treated as one entity, and the mass properties for that single entity will be calculated. Thus, for example, if two element blocks, say, `block_150` and `block_210`, are specified on the `BLOCK` command line, the total mass for the two element blocks will be reported as the total mass property.

5.4.2 Structure Command

The output for the mass properties will be identified by the command line

```
STRUCTURE NAME = <string>structure_name ,
```

where the string `structure_name` is a user-defined name for the structure.

5.5 Element Death

```
BEGIN ELEMENT DEATH <string>death_name
  # {block set commands}
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  # criteria commands
  CRITERION IS AVG|MAX|MIN NODAL VALUE OF
    <string>var_name[(<integer>component_num)]
    <|<=|=|>=|> <real>tolerance
  CRITERION IS ELEMENT VALUE OF
    <string>var_name[(<integer>component_num)]
    <|<=|=|>=|> <real>tolerance |
    <string>derived_quantity[(<integer>component_num)]
    <|<=|=|>=|> <real>tolerance
  CRITERION IS GLOBAL VALUE OF
    <string>var_name[(<integer>component_num)]
    <|<=|=|>=|> <real>tolerance
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  MATERIAL CRITERION = <string list>material_model_names
  #
  # evaluation commands
  CHECK STEP INTERVAL = <integer>num_steps
  CHECK TIME INTERVAL = <real>delta_t
  DEATH START TIME = <real>time
  #
  # miscellaneous option commands
  DEATH STEPS = <integer>death_steps(1)
  FORCE VALID ACME CONNECTIVITY
  #
  # additional command
  ACTIVE PERIODS = <string list>period_names
END [ELEMENT DEATH <string>death_name]
```

The `ELEMENT DEATH` command block is used to remove elements from an analysis. For example, the command block can be used to remove elements that have fractured, that are no longer important to the analysis results, or that are nearing inversion. The name of the command block, `death_name`, is user-defined and can be referenced in other commands to update boundary or contact conditions based on the death of elements creating new exposed surfaces.

Any element in an element block or element blocks selected in the `ELEMENT DEATH` command block is removed (killed) when one of the criteria specified in the `ELEMENT DEATH` command block is satisfied by that element. When an element dies, it is removed permanently. Any number of `ELEMENT DEATH` command blocks may exist within a region. However, a particular block of elements may be associated with at most one element death instance. If all of the elements in a region are killed, the analysis will terminate. If all of the elements attached to a node are killed, the mass for the node and all associated nodal quantities will be set to zero. If an element is killed, but an attached element is not, the mass of the killed element will remain at the nodes shared with the "living" element.

Elements may be killed using a derived stress, a derived strain, or a derived log strain quantity. To kill elements using a derived quantity, the quantity must be specifically listed in one of the command blocks described in Section 5.6.

The `ELEMENT DEATH` command block contains four groups of commands—block set, criteria, evaluation, and miscellaneous. In addition to the command lines in the four groups, there is one additional command line: `ACTIVE PERIODS`. The command block must contain commands from the block set and criteria groups. Command lines from the evaluation and miscellaneous groups are optional, as is the additional command. Following are descriptions of the different command groups and the `ACTIVE PERIODS` command line, an example of using the `ELEMENT DEATH` command block, and some concluding remarks related to element death visualization.

5.5.1 Block Set Commands

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

The {`block set commands`} portion of the `ELEMENT DEATH` command block defines a set of blocks for selecting the elements to be referenced. These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of blocks, as described in Section 6.1.

Element death must apply to a group of elements. There are two commands for selecting the elements to be referenced: `BLOCK` and `INCLUDE ALL BLOCKS`. In the `BLOCK` command line, you can list a series of blocks through the string list `block_names`. This command line may also be repeated multiple times. The `IN-`

CLUDE ALL BLOCKS command line adds all the element blocks present in the region to the current element death instance. There must be at least one BLOCK or INCLUDE ALL BLOCKS command line in the ELEMENT DEATH command block.

The REMOVE BLOCK command line allows you to delete blocks from the set specified in the BLOCK and/or INCLUDE ALL BLOCKS command line(s) through the string list block_names. Typically, you will use the REMOVE BLOCK command line with the INCLUDE ALL BLOCKS command line. If you want to include all but a few of the element blocks, a combination of the REMOVE BLOCK command line and INCLUDE ALL BLOCKS command line should minimize input information.

5.5.2 Criteria Commands

Any combination of criteria (CRITERION IS NODAL, CRITERION IS ELEMENT, CRITERION IS GLOBAL, ELEMENT BLOCK SUBROUTINE, MATERIAL CRITERION) can be specified within a single ELEMENT DEATH command block. However, only one user subroutine criterion may appear in a set of criteria command lines. As long as any one criterion is met, element death will occur. You do not have to meet all criteria that appear in an ELEMENT DEATH command block for element death to occur. Element death with multiple criteria is an OR condition rather than an AND condition.

A problem with a material that has both a tension cutoff and a compression cutoff would be an example for which you would want to use element death with multiple criteria. You would use one criterion to set failure in tension and another criterion to set failure in compression. If either the tension cutoff value, as set by the tension criterion, or the compression cutoff value, as set by the compression criterion, was exceeded for an element, the element would be killed.

As another example, you might have a problem that uses a nodal criterion and a user subroutine criterion. For this second example, you are precluded from using another subroutine criterion because you already have one. You could add some combination of nodal, element, global, or material criteria to the existing nodal and subroutine criteria, but you could not add another subroutine criterion.

5.5.2.1 Nodal Variable Death Criterion

```
CRITERION IS AVG|MAX|MIN NODAL VALUE OF
  <string>var_name[(<integer>component_num)]
  <|<=|=|>=|> <real>tolerance
```

Any registered Presto nodal variable may be used by an element death criterion, as specified in this nodal criterion command line.

The input parameters are described as follows:

- Nodal variables are present on the nodes of an element, and these nodal values must be reduced to a single element value for use by the criterion. The available types of reduction are `AVG`, which takes the average of the nodal values; `MAX`, which takes the maximum of the nodal values; and `MIN`, which takes the minimum of the nodal values.

- The string `var_name` gives the name of the registered variable. See Section 8.5 for a listing of the registered variables.

- For variables with multiple components, a component number can be specified. For example, a value of `(2)` for `component_num` refers to the y component of displacement. See Table 8.4 in Chapter 8 for how to determine the component number.

- The specified variable, with an optional component specification if the variable has components, may be compared to a given tolerance with one of five operators. The operator is specified with the appropriate symbol for less than (<), less than or equal to (<=), equal to (=), greater than or equal to (>=), or greater than (>). The given tolerance is specified with the real value `tolerance`.

Any user-defined variable is also treated as a registered variable (see Section 9.2.4). Therefore, you could use a user-defined variable as the `var_name` in the above death criterion command line, and the syntax will actually accept this usage of a user-defined variable. However, the use of a user-defined variable in a death criterion command line is not recommended. There are timing (when user-defined variables are calculated) and parallel issues regarding the use of a user-defined variable with element death. User subroutines should cover situations for element death where you might want to reference a user variable.

5.5.2.2 Element Variable Death Criterion

```
CRITERION IS ELEMENT VALUE OF
  <string>var_name[(<integer>component_num)]
  <|<=|=|>=|> <real>tolerance |
  <string>derived_quantity[(<integer>int_num)]
  <|<=|=|>=|> <real>tolerance
```

Any registered Presto element variable, derived stress quantity, derived strain quantity, or derived log strain quantity may be used by an element death criterion, as specified in this element criterion command line. An element variable or a derived quantity is present on the element itself, so no reduction is required, which is why the first line in the format of the above command line differs from that of the nodal criterion command line in Section 5.5.2.1. As shown in the command line format above, the input parameters for an element variable differ somewhat from those of a derived quantity and are thus described separately.

For a criterion using an element variable, the variable name, component number, and tolerance can be specified in the same manner as defined for the nodal criterion command line.

The input parameters for the case of an element variable name are described as follows:

- The string `var_name` gives the name of the registered variable. See Section 8.5 for a listing of the registered variables.
- For variables with multiple components, a component number can be specified. For example, a value of `(2)` for `component_num` refers to the yy component of stress. See Table 8.4 in Chapter 8 for how to determine the component number.
- The specified variable, with an optional component specification if the variable has components, may be compared to a given tolerance with one of five operators. The operator is specified with the appropriate symbol for less than ($<$), less than or equal to ($<=$), equal to ($=$), greater than or equal to ($>=$), or greater than ($>$). The given tolerance is specified with the real value `toler-ance`.

Any user-defined variable is also treated as a registered variable (see Section 9.2.4). Therefore, you could use a user-defined variable as the `var_name` in the above death criterion command line, and the syntax will actually accept this usage of a user-defined variable. However, the use of a user-defined variable in a death criterion command line is not recommended. There are timing (when user-defined variables are calculated) and parallel issues regarding the use of a user-defined variable with element death. User subroutines should cover situations for element death where you might want to reference a user variable.

For a criterion using a derived quantity, the derived quantity name, integration point number, and tolerance can be specified. The complete set of derived stress, strain, and log strain quantities is described in Section 8.1.1.4, which deals with the output of derived quantities. Table 8.1 lists all derived stress quantities, Table 8.3 lists all derived strain quantities, and Table 8.2 lists all derived log strain quantities.

The input parameters for the case of a derived quantity are described as follows:

- The string `derived_quantity` gives the name of the derived quantity. See Section 8.1.1.4 for a listing of the derived quantities.

- For a derived quantity defined at multiple integration points, an integration point number can be specified. The parameter `int_num` specifies the integration point for the derived quantity. For example, if you wanted to use the von Mises stress at integration point 3 for an element criterion, you would use `von_mises` for `derived_quantity` and `(3)` for `int_num`, which would appear as follows:

```
CRITERION IS ELEMENT VALUE OF von_mises(3)
```

- The specified variable, with an optional component specification if the variable has components, may be compared to a given tolerance with one of five operators. The operator is specified with the appropriate symbol for less than (<), less than or equal to (<=), equal to (=), greater than or equal to (>=), or greater than (>). The given tolerance is specified with the real value `tolerance`.

Importantly, when you specify a derived quantity via the `CRITERION IS ELEMENT VALUE OF` command line, you may also need to include an associated command block for the specific quantity. The associated command block for a derived stress is `DERIVED OUTPUT`; for a derived strain, it is `DERIVED STRAIN OUTPUT`; and for a derived log strain, it is `DERIVED LOG STRAIN OUTPUT`. These three command blocks are described in Section 5.6. The conditions determining their usage are partly contingent upon certain output considerations and are discussed next.

As explained in Section 8.1.1.4, derived quantities can be output in the results file using the `ELEMENT VARIABLES` command line. If you specify the derived quantity that you want to use as a criterion for element death in the `ELEMENT VARIABLES` command line in the `RESULTS OUTPUT` command block, you do not need to include the command block in Section 5.6 associated with that derived quantity because Presto will calculate the derived quantity for output and make it available for use as an element death criterion. However, if you only want to use the derived quantity as a criterion for element death and you do not want to have it calculated for output, you will need to include the associated command block (`DERIVED OUTPUT`, `DERIVED STRAIN OUTPUT`, or `DERIVED LOG STRAIN OUTPUT`) for the particular derived quantity in the input file.

5.5.2.3 Global Death Criterion

```
CRITERION IS GLOBAL VALUE OF
<string>var_name[(<integer>component_num)]
<|<=|=|>=|> <real>tolerance
```

Any registered Presto global variable may be used as an element death criterion. Once the global criterion is reached, all elements specified in the `ELEMENT DEATH` command block are killed. The variable name, component number, and tolerance can be specified in the same manner as defined for the nodal criterion command line.

The input parameters are described as follows:

- The string `var_name` gives the name of the registered variable. See Section 8.5 for a listing of the registered variables.

- For variables with multiple components, a component number can be attached. Global variables are typically scalar quantities. If you encounter a global variable with multiple components, consult with Table 8.4 in Chapter 8 for how to determine the component number. Section 5.5.2.1 and Section 5.5.2.2 provide examples of using the component option.

- The specified variable, with an optional component specification if the variable has components, may be compared to a given tolerance with one of five operators. The operator is specified with the appropriate symbol for less than (<), less than or equal to (<=), equal to (=), greater than or equal to (>=), or greater than (>). The given tolerance is specified with the real value `tolerance`.

5.5.2.4 Subroutine Death Criterion

```
ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

A death criterion can be specified via a user-defined subroutine (see Chapter 9), which is invoked by using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user. The user-defined subroutine for element death must be an element subroutine signature (see Section 9.2.2). The element subroutine will return an output values array and a flag array of one flag per element (see Table 9.2 in Chapter 9). The output values array is ignored. Death is determined by the flag return value. For user-defined subroutines, a flag return value of –1 indicates that the element should die. A flag return value greater than or equal to 0 indicates that the element should remain alive.

Following the `ELEMENT BLOCK SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Section 9.

5.5.2.5 Material Death Criterion

```
MATERIAL CRITERION = <string list>material_model_names
```

Some material models have a failure criterion. When this failure criterion is satisfied within an element, the element has fractured or disintegrated. The material models reduce the stress in these fractured or disintegrated elements to zero. The `MATERIAL CRITERION` command line can be used to remove these fractured or disintegrated elements from an analysis. Removal of the fractured elements will speed computations, enhance visualization, and prevent spurious inversion of these elements that may stop the analysis.

The current supported material models for use by the `MATERIAL CRITERION` command line are `ELASTIC_FRACTURE` (see Section 4.1.3), `DUCTILE_FRACTURE` (see Section 4.1.6), and `ML_EP_FAIL` (see Section 4.1.7).

Suppose you have an element block named `part1_ss304` that references a material named `SS304`. This material, `SS304`, uses the `DUCTILE_FRACTURE` material model (see Section 4.1.6). You also have an element block named `ring5_al6061` that references a material named `al6061`. This material, `al6061`, uses the `ML_EP_FAIL` material model (see Section 4.1.7). If you have an `ELEMENT DEATH` command block with the command line

```
BLOCK = part1_ss304 ring5_al6061
```

and the command line

```
MATERIAL CRITERION = DUCTILE_FRACTURE ML_EP_FAIL ,
```

then any element in `part1_ss304` that fails according to the material model `DUCTILE_FRACTURE` (in material `SS304`) and any element in `ring5_al6061` that fails according to the material model `ML_EP_FAIL` (in material `al6061`) will be killed by element death.

5.5.3 Evaluation Commands

```
CHECK STEP INTERVAL = <integer>num_steps
CHECK TIME INTERVAL = <real>delta_t
```

```
        DEATH START TIME = <real>time
```

Evaluation of element death criteria may be time consuming. Additionally, reconstruction of contact or other boundary conditions after element death can be very time consuming. For these reasons, three command lines are available for determining the frequency at which element death is evaluated. The default is to evaluate element death at every time step. You can limit the number of times at which the element death evaluation is done by using the following commands.

- The `CHECK STEP INTERVAL` command line instructs element death to evaluate the element death criteria only every `num_steps` time steps.
- The `CHECK TIME INTERVAL` command line instructs element death to evaluate the element death criteria only every `delta_t` time units.
- The `DEATH START TIME` command line instructs element death not to evaluate death criteria before a user-specified time, as given by the real value `time`.

You may use both the `CHECK STEP INTERVAL` and `CHECK TIME INTERVAL` command lines in a command block. Evaluations for element death will be made at both the time and step intervals if both of these command lines are included.

All three of the above command lines—`CHECK STEP INTERVAL`, `CHECK TIME INTERVAL`, and `DEATH START TIME`—are optional command lines.

5.5.4 Miscellaneous Option Commands

The command lines listed below need not be present in the `ELEMENT DEATH` command block unless the conditions addressed by each call for their inclusion.

5.5.4.1 Death Steps

```
      DEATH STEPS = <integer>death_steps(1)
```

If the `DEATH STEPS` command line is used and the value for `death_steps` is set to some value greater than 1, the stress in a killed element will not be set to 0 until the prescribed number of steps has occurred. The stress in the killed element will decrease (if it is positive) to 0 in a linear fashion over the prescribed number of steps; the stress in the killed element will increase (if it is negative) to 0 over the prescribed number of steps. If the stress in a killed element is set to 0 over a single time step, the resulting change in stress can sometimes cause instabilities due to the sudden release of energy. However, elimination of the stress over an excessive number of load steps can make it appear as if the element is present long after it has been killed. The default number of steps, as provided in the integer value `death_steps`, is 1.

The value you select for `death_steps` will depend on your analysis. A small number such as 3 or 5 may be sufficient to prevent instabilities for most cases. However, in some cases it may be necessary to use a value for `death_steps` of 10 or larger. The loads, material models, and model complexity in your analysis will impact the value of `death_steps`.

5.5.4.2 Degenerate Mesh Repair

```
FORCE VALID ACME CONNECTIVITY
```

If the `FORCE VALID ACME CONNECTIVITY` command line is present, degenerate mesh occurrences will be repaired. Element death has the possibility of creating degenerate mesh occurrences that will not be accepted by the ACME (see Reference 11) contact algorithms used by Presto. For example, if two continuum elements are connected only by an edge, ACME will not accept the mesh as a valid mesh. For this degenerate mesh occurrence (continuum elements connected only at an edge), the degeneracy is repaired by deleting all elements attached to the offending edge if we have turned on this repair option.

The option to repair degenerate mesh occurrences is on by default if there is a `CONTACT DEFINITION` command block that includes the command line

```
UPDATE ALL SURFACES FOR ELEMENT DEATH = ON .
```

See Section 7.4 for a description of the `UPDATE ALL SURFACES FOR ELEMENT DEATH` command line.

If you do not have a `CONTACT DEFINITION` command block and want to repair degenerate mesh occurrences for whatever purposes, you should include the `FORCE VALID ACME CONNECTIVITY` command line.

5.5.4.3 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `ELEMENT DEATH` command block:

```
ACTIVE PERIODS = <string list>period_names
```

This command line determines when element death is active. See Section 2.5 for more information about this optional command line.

5.5.5 Example

The following example provides instructions to kill elements in `block_1` when they leave a bounding box. This type of element death can be useful in an analysis where some peripheral parts, because of fracture, separate and fly away from a central body, this central body being our part of interest. In this case, these peripheral parts no longer impact the solution. The instructions in this `ELEMENT DEATH` command block will cause the parts to be killed, thus speeding up the computation.

```
begin element death out_of_bounds
  block = block_1
  # check x coordinates
  criterion is avg nodal value of coordinates(1) >=  10
  criterion is avg nodal value of coordinates(1) <= -10
  # check y coordinates
  criterion is avg nodal value of coordinates(2) >=  10
  criterion is avg nodal value of coordinates(2) <= -10
  # check z coordinates
  criterion is avg nodal value of coordinates(3) >=  10
  criterion is avg nodal value of coordinates(3) <= -10
end element death out_of_bounds
```

5.5.6 Element Death Visualization

When an element dies, information about this element will still be sent, along with information for all other elements, to the Exodus II results file. (Chapter 8 describes the output of element variables to the results file.) The death status of the elements may be output to the results file by requesting element variable output for the element variable `DEATH_DUMMY_VAR`. Including the command line

```
ELEMENT VARIABLES = DEATH_DUMMY_VAR as death_var
```

in a `RESULTS OUTPUT` command block (Chapter 8) will output this element variable with the name `death_var` in the results file.

The convention for `DEATH_DUMMY_VAR` is as follows: An element with a value of 1.0 for `DEATH_DUMMY_VAR` is a living element. An element with a value of 0.0 for `DEATH_DUMMY_VAR` is a dead element. A value between 1.0 and 0.0 indicates an element in the process of dying. A dying element has its material stress scaled down over a number of time steps. The current scaling factor for an element is given by `DEATH_DUMMY_VAR`. Whether or not an element can have a value for `DEATH_DUMMY_VAR` other than 0.0 or 1.0 will depend on whether or not you have used the `DEATH STEPS` option in the `ELEMENT DEATH` command block. If the number of steps over which death occurs is greater than 1, then `DEATH_DUMMY_VAR` can be some value between 0.0 and 1.0.

If `DEATH_DUMMY_VAR` is written to a results file, and the results file is used in some visualization program to examine the mesh for the model, it is possible to use `DEATH_DUMMY_VAR` to exclude killed elements in any view of the model. A subset of the mesh showing just the living elements can be created by selecting only those elements with `DEATH_DUMMY_VAR` $= 1.0$ values.

5.6 Derived Quantities for Element Death

```
BEGIN DERIVED OUTPUT
  COMPUTE AND STORE STRESS VARIABLE =
    <string>derived_quantity_name
END DERIVED OUTPUT
#
BEGIN DERIVED STRAIN OUTPUT
  COMPUTE AND STORE STRAIN VARIABLE =
    <string>derived_quantity_name
END DERIVED OUTPUT
#
BEGIN DERIVED LOG STRAIN OUTPUT
  COMPUTE AND STORE LOG STRAIN VARIABLE =
    <string>derived_quantity_name
END DERIVED OUTPUT
```

The above command blocks are used in conjunction with the `CRITERION IS ELEMENT VALUE OF` command line in the `ELEMENT DEATH` command block (see Section 5.5.2.2). Each of the blocks is associated with a type of derived quantity: `DERIVED OUTPUT` is associated with derived stress, `DERIVED STRAIN OUTPUT` with derived strain, and `DERIVED LOG STRAIN OUTPUT` with derived log strain. Structured similarly, each of the three command blocks can contain one or more command lines, with each command line requiring a single value, the string `derived_quantity_name`. The derived quantity specified by `derived_quantity_name` will also appear on a `CRITERION IS ELEMENT VALUE OF` command line. If a derived quantity is referenced on a `CRITERION IS ELEMENT VALUE OF` command line, it must appear in one of the above three command blocks unless the condition described in the following paragraph is satisfied.

As discussed in Section 5.5.2.2, the inclusion of one of the three command blocks above is not mandatory when you have specified the desired quantity in a `CRITERION IS ELEMENT VALUE OF` command line. You only need to specify the applicable command block for the derived quantity if you have not specified that same derived quantity for output in the `RESULTS OUTPUT` file via the `ELEMENT VARIABLES` command line.

Following are some examples of using the three command blocks discussed in this section.

- If you want to use a derived stress quantity for an element death option (and not for output), you must include a `DERIVED OUTPUT` command block. Suppose you want an element death option that uses the von Mises stress, then you would include the following command block:

```
BEGIN DERIVED OUTPUT
  COMPUTE AND STORE STRESS VARIABLE = von_mises
END DERIVED OUTPUT
```

Consult Table 8.1 for a complete listing of derived stress quantities.

- If you want to use a derived strain quantity for an element death option (and not for output), you must include a `DERIVED STRAIN OUTPUT` command block. Suppose you want an element death option using the octrahedral shear strain (see Section 8.1.1.4). Then you would include the following command block:

```
BEGIN DERIVED STRAIN OUTPUT
  COMPUTE AND STORE STRAIN VARIABLE =
    octrahedral_shear_strain
END DERIVED STRAIN OUTPUT
```

  Consult Table 8.3 for a complete listing of derived strain quantities.

- If you want to use a derived log strain quantity for an element death option (and not for output), you must include a `DERIVED LOG STRAIN OUTPUT` command block. Suppose you want element death options using the octrahedral shear log strain and the first invariant of the log strain (see Section 8.1.1.4). Then you would include the following command block:

```
BEGIN DERIVED LOG STRAIN OUTPUT
  COMPUTE AND STORE LOG STRAIN VARIABLE =
    octrahedral_shear_log_strain
  COMPUTE AND STORE LOG STRAIN VARIABLE =
    log_stain_invariant_1
END DERIVED LOG STRAIN OUTPUT
```

  The above example would require that two `CRITERION FOR ELEMENT VALUE OF` command lines be included in the `ELEMENT DEATH` command block, one providing the criterion for `octrahedral_shear_log_strain` and the other providing the criterion for `octrahedral_shear_log_strain`. Consult Table 8.2 for a complete listing of derived log strain quantities.

5.7 Mesh Rebalancing

Mesh rebalancing is a feature in Presto that may improve the efficiency of an analysis. Two command blocks can be used for mesh rebalancing: `REBALANCE` and `ZOLTAN PARAMETERS`. The `REBALANCE` command block is required; the `ZOLTAN PARAMETERS` command block is optional. Sections 5.7.1 and 5.7.2 describe these command blocks.

5.7.1 Rebalance

```
BEGIN REBALANCE
  INITIAL REBALANCE = ON|OFF(OFF)
  PERIODIC REBALANCE = ON|OFF|AUTO(OFF)
  STEP INTERVAL = <integer>step_interval
  COMMUNICATION RATIO THRESHOLD = <real>ratio
  ZOLTAN PARAMETERS = <string>parameter_name
END REBALANCE
```

Initial decomposition of a mesh for parallel runs with Presto is done by a utility called *loadbal*. The initial decomposition provided by *loadbal* may not provide a decomposition for good-to-optimal parallel performance of Presto under certain circumstances. Therefore, Presto supports a simple mesh-rebalancing capability that can be used to improve the parallel performance of some problems. When mesh rebalancing is invoked, the parallel decomposition is changed, and elements are moved among the processors to balance the computational load and minimize the processor-to-processor communication. Mesh rebalancing may be useful in the following circumstances:

- The mesh decomposition produced by *loadbal* for SPH meshes is nearly always poor. It is recommended that an initial mesh rebalance be done for all SPH problems.

- If a problem experiences very large deformations, periodic rebalancing may be helpful. In contact or SPH problems, communication is performed between physically nearby contact surfaces or SPH particles. To maintain optimum performance, it is helpful to have neighboring particles located on the same processors. Periodic mesh rebalancing can ensure that neighboring entities tend to remain on the same processor during large mesh deformations.

The `REBALANCE` command block is placed in the Presto region scope. The mesh rebalancing in Presto uses a mesh balancing library called Zoltan (Reference 12). Zoltan performs the actual rebalancing. By default, Presto creates a Zoltan object with a default set of parameters. However, a Zoltan object with a customized set of parameters can be created and referenced from the `REBALANCE` command block.

5.7.1.1 Rebalance Command Lines

```
INITIAL REBALANCE = on|off (off)
PERIODIC REBALANCE = on|off|auto (off)
STEP INTERVAL = <integer>step_interval
COMMUNICATION RATIO THRESHOLD = <real>ratio
```

The above command lines control the frequency at which the rebalancing is done.

The `INITIAL REBALANCE` command line is used to rebalance the mesh at time zero before any calculations occur. This option should be used if the initial mesh decomposition passed to Presto is poor.

If the `PERIODIC REBALANCE COMMAND` option is `on`, the mesh will be rebalanced every `step_interval` steps, where `step_interval` is the parameter specified by the `STEP INTERVAL` command line. If the option is `auto`, the mesh will be rebalanced every `step_interval` steps or when the communication ratio reaches a critical value. The communication ratio is currently defined only for SPH problems. The communication ratio is a measure of how much communication is required in the current mesh decomposition versus an estimate of what the communication could be with an optimal decomposition. Mesh rebalancing is expensive, so rebalancing should be done rarely. The communication ratio is set by the parameter `ratio` on the `COMMUNICATION RATIO THRESHOLD` command line. A communication ratio parameter in the range of 1.25 to 1.5 is usually ideal.

5.7.1.2 Zoltan Command Line

The command line

```
ZOLTAN PARAMETERS = <string>parameter_name
```

references a `ZOLTAN PARAMETERS` command block named `parameter_name`. Various parameters for Zoltan can be set in the `ZOLTAN PARAMETERS` command block. If you do not use the `ZOLTAN PARAMETERS` command line, a default set of parameters is used. The default parameter command block is shown as follows:

```
BEGIN ZOLTAN PARAMETERS default
  LOAD BALANCING METHOD = recursive coordinate bisection
    # string parameter
  OVER ALLOCATE MEMORY  = 1.5 # real parameter
  REUSE CUTS = true # string parameter
  ALGORITHM DEBUG LEVEL = 0 # integer parameter
  CHECK GEOMETRY = true # string parameter
  ZOLTAN DEBUG LEVEL = 0 # integer parameter
END ZOLTAN PARAMETERS default
```

See Section 5.7.2 for a discussion of the `ZOLTAN PARAMETERS` command block. Section 5.7.2 lists the command lines that can be used to set Zoltan parameters.

5.7.2 Zoltan Parameters

```
BEGIN ZOLTAN PARAMETERS <string>parameter_name
  LOAD BALANCING METHOD  = <string>recursive coordinate bisection|
    recursive inertial bisection|hilbert space filling curve|
    octree
  DETERMINISTIC DECOMPOSITION = <string>false|true
  IMBALANCE TOLERANCE = <real>imb_tol
  OVER ALLOCATE MEMORY = <real>over_all_mem
  REUSE CUTS = <string>false|true
  ALGORITHM DEBUG LEVEL = <integer>alg_level
    #  0<=(alg_level)<=3
  CHECK GEOMETRY = <string>false|true
  KEEP CUTS = <string>false|true
  LOCK RCB DIRECTIONS = <string>false|true
  SET RCB DIRECTIONS = <string>do not order cuts|xyz|xzy|
    yzx|yxz|zxy|zyx
  RECTILINEAR RCB BLOCKS = <string>false|true
  RENUMBER PARTITIONS = <string>false|true
  OCTREE DIMENSION = <integer>oct_dimension
  OCTREE METHOD = <string>morton indexing|grey code|hilbert
  OCTREE MIN OBJECTS = <integer>min_obj  #  1<=(min_obj)
  OCTREE MAX OBJECTS = <integer>max_obj  #  1<=(max_obj)
  ZOLTAN DEBUG LEVEL = <integer>zoltan_level
    #  0<=(zoltan_level)<=10
  DEBUG PROCESSOR NUMBER = <integer>proc  #  1<=proc
  TIMER = <string> wall|cpu
  DEBUG MEMORY = <integer>dbg_mem  #  0<=(dbg_mem)<=3
END [ZOLTAN PARAMETERS <string>parameter_name]
```

The `ZOLTAN PARAMETERS` command block is used to set parameters for Zoltan (see Reference 12), a program that can be used for mesh rebalancing in Presto. The `ZOLTAN PARAMETERS` command block is used in association with the `REBALANCE` command block. A `REBALANCE` command block may reference a `ZOLTAN PARAMETERS` command block via the name, `parameter_name`, for the parameter command block. Reference Section 5.7.1 regarding the use of the `ZOLTAN PARAMETERS` command block for mesh rebalancing in Presto.

Setting the parameters for Zoltan involves some understanding of how Zoltan works. We will not present a discussion of the various parameters that you may set in the `ZOLTAN PARAMETERS` command block. You should consult with Reference 12 for a discussion of the parameters that can be set by the various command lines in the `ZOLTAN PARAMETERS` command block. Note that some of the command lines in this command block have comments that provide additional information about the parameters. The "#" symbol precedes a comment.

In the above command block, = and `IS` are the allowed delimiters, which is different from the usual Presto convention of =, `IS`, and `ARE`. Note that the `ZOLTAN PARAMETERS` command block should be specified in the domain scope when it is referenced from the `ZOLTAN PARAMETERS` command line in the `REBALANCE` command block. When the default set of parameters is used for a Zoltan object, the `ZOLTAN PARAMETERS` command block need not be included in the input file.

5.8 References

1. Taylor, L. M., and D. P. Flanagan. *Pronto3D: A Three-Dimensional Transient Solid Dynamics Program*, SAND87-1912. Albuquerque, NM: Sandia National Laboratories, March 1989.

2. Rashid, M. M. "Incremental Kinematics for Finite Element Applications." *International Journal for Numerical Methods in Engineering* 36 (1993): 3937–3956.

3. Dohrman, C. R., M. W. Heinstein, J. Jung, S. W. Key, and W. R. Witkowski. "Node-Based Uniform Strain Elements for Three-Node Triangular and Four-Node Tetrahedral Meshes." *International Journal for Numerical Methods in Engineering* 47 (2000): 1549–1568.

4. Key, S. W., M. W. Heinstein, C. M. Stone, F. J. Mello, M. L. Blanford, and K. G. Budge. "A Suitable Low-Order, Tetrahedral Finite Element for Solids." *International Journal for Numerical Methods in Engineering* 44 (1999) 1785–1805.

5. Key, S. W., and C. C. Hoff. "An Improved Constant Membrane and Bending Stress Shell Element for Explicit Transient Dynamics." *Computer Methods in Applied Mechanics and Engineering* 124, no. 1–2 (1995): 33–47.

6. Laursen, T. A., S. W. Attaway, and R. I. Zadoks. *SEACAS Theory Manuals: Part III. Finite Element Analysis in Nonlinear Solid Mechanics*, SAND98-1760/3. Albuquerque, NM: Sandia National Laboratories, 1999.

7. Flanagan, D. P., and T. Belytschko. "A Uniform Strain Hexahedron and Quadrilateral with Orthogonal Hourglass Control." *International Journal for Numerical Methods in Engineering* 17 (1981): 679–706.

8. Swegle, J. W. *SIERRA: PRESTO Theory Documentation: Energy Dependent Materials Version 1.0.* Albuquerque, NM: Sandia National Laboratories, October 2001.

9. Swegle, J. W., S. W. Attaway, M. W. Heinstein, F. J. Mello, and D. L. Hicks. *An Analysis of Smoothed Particle Hydrodynamics*, SAND93-2513. Albuquerque, NM: Sandia National Laboratories, March 1994.

10. Sjaardema, G. D. *Overview of the Sandia National Laboratories Engineering Analysis Code Access System*, SAND92-2292. Albuquerque, NM: Sandia National Laboratories, January 1993.

11. Brown, K. H., R. M. Summers, M. W. Glass, A. S. Gullerud, M. W. Heinstein, and R. E. Jones. *ACME: Algorithms for Contact in a Multiphysics Environment*, API Version, 2.2, SAND2004-5486. Albuquerque, NM: Sandia National Laboratories, October 2001.

12. Sandia National Laboratories. "Zoltan: Data-Management Services for Parallel Applications." http://www.cs.sandia.gov/zoltan/ (accessed May 31, 2006).

Intentionally Left Blank

Chapter 6

Boundary Conditions and Initial Conditions

Presto offers a variety of options for defining boundary and initial conditions. Typically, boundary and initial conditions are defined on some subset of mesh entities (node, element face, element) defining a model. Presto offers a flexible means to define subsets of mesh entities. Section 6.1 describes commands that will let you define some subset of a mesh entity using a collection of commands that constitute a set of Boolean operators.

The remaining parts of this chapter discuss the following functionality:

- Section 6.2 presents methods for setting the initial values of registered variables in Presto. Presto has the flexibility to set a complex initial state for some variable such as nodal velocity or element stress.
- Kinematic boundary conditions typical of those you would expect in an explicit, transient dynamics code (fixed displacement, prescribed acceleration, etc.) are options in Presto and described in Section 6.3. Most of these boundary conditions let you specify a time history using some SIERRA function or a more complex time-varying and spatially varying history with a user subroutine.
- Section 6.4 documents a number of initial velocity options available in Presto.
- Force boundary conditions typical of those you would expect in an explicit, transient dynamics code (prescribed force, traction, etc.) are options in Presto and described in Section 6.5. Most of these force boundary conditions let you specify a time history using some SIERRA function or a more complex time-varying and spatially varying history with a user subroutine.
- Section 6.6 details a number of options available for describing a temperature field in Presto.

- Presto has a number of specialized boundary conditions—gravity, cavity expansion, silent boundary, spot-weld, and line weld. These specialized boundary conditions are described in Section 6.7.

6.1 General Mesh-Entity Assignment Commands

A number of standard command lines exist to define a set of mesh entities (node, element face, element) associated with some type of mechanics. (Mechanics can be a boundary condition, an initial condition, or a gravity load.) All these command lines exist within the command blocks for the various mechanics, which in turn exist within the region scope. These command lines, taken collectively, constitute a set of boolean operators for constructing sets of mesh entities.

The first set of command lines we will consider is as follows:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
```

In the above command lines, the string list `nodelist_names` is used to represent one or more node sets as discussed in Section 1.5. A node set is referenced as `nodelist_id`, where id is some integer. For example, suppose you have three node lists in your model: 10, 23, and 105. If you want to combine all these node lists so that they form one set of nodes for, say, your boundary condition or initial condition, then you would use the command line

`NODE SET = nodelist_10 nodelist_23 nodelist_105` .

This convention applies as well to any surface-related command line that uses the string list `surface_names` or any block-related command line that uses the string list `block_names`.

The `NODE SET` command line associates a set of nodes with a mechanics. A mechanics may be applied to multiple node sets by putting multiple node set names on the command line or by repeating the command line multiple times.

The `SURFACE` command line associates a set of element faces with a mechanics. A mechanics may be applied to multiple surfaces by putting multiple surface names on the command line or by repeating the command line multiple times. The `SURFACE` command line can also be used to associate a set of nodes with a mechanics. For example, suppose we wish to use the fixed displacement kinematic boundary condition. Although this is a nodal boundary condition (the condition is applied to individual nodes), a `SURFACE` command line can be used to establish the set of nodes. If the command line

```
SURFACE = surface_101
```

appears in a fixed displacement boundary condition, then all the nodes associated with surface 101 will be associated with the boundary condition.

The `BLOCK` command line associates a set of elements with a mechanics. A mechanics may be applied to multiple blocks by putting multiple block names on the command line or by repeating the command line multiple times. The `BLOCK` command line can also be used to associate a set of nodes with a mechanics. For example, suppose we wish to use the fixed displacement kinematic boundary condition as in the previous example. If the command line

```
BLOCK = block_50
```

appears in a fixed displacement kinematic boundary condition, then all the nodes associated with block 50 will be associated with the boundary condition.

The `INCLUDE ALL BLOCKS` command line associates all blocks with a mechanics. This will apply the mechanics to all nodes and elements in the model.

The block-related command lines associated with contact will generate surfaces. The block command lines associated with boundary conditions, initial conditions, and gravity will NOT generate surfaces.

Any combination of the above command lines can be used to create a union of mesh entities. Suppose, for example, that the command lines

```
NODE SET = nodelist_2
SURFACE = surface_3
```

appear in a `FIXED DISPLACEMENT` command block for a kinematic boundary condition. The set of nodes associated with the boundary condition will be the union of the set of nodes associated with surface 3 and the set of nodes associated with node set 2.

When a union of mesh entities is created by using two or more of the above command lines, a mesh entity may appear in more than one topological entity—node set, surface, block. However, the mechanics is applied to the mesh entity only once. For example, node 67 may be a part of nodelist 2 and surface 3. Including both nodelist 2 and surface 3 into a mechanics will only apply the mechanics to node 67 once.

The set of mesh entities associated with a mechanics can be edited (mesh entities can be deleted from the set) by using the following command lines:

```
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
```

```
    REMOVE BLOCK = <string list>block_names
```

The `REMOVE NODE SET` command line deletes a set of nodes from the node set associated with a mechanics.

The `REMOVE SURFACE` command line deletes a set of element faces from the set of element faces associated with a mechanics. It will remove a set of nodes associated with the surface from the set of nodes associated with the mechanics.

The `REMOVE BLOCK` command line deletes a set of elements from the set of elements associated with a mechanics. It will remove a set of nodes associated with the block from the node set associated with the mechanics.

6.2 Initial Variable Assignment

```
BEGIN INITIAL CONDITION
  # {mesh-entity set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # variable identification commands
  INITIALIZE VARIABLE NAME = <string>var_name
  VARIABLE TYPE = [NODE|EDGE|FACE|ELEMENT|GLOBAL]
  #
  # constant magnitude command
  MAGNITUDE = <real list>initial_values
  #
  # input mesh commands
  READ VARIABLE = <string>mesh_var_name
  TIME = <real>time
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name |
    SURFACE SUBROUTINE = <string>subroutine_name |
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional command
  SCALE FACTOR = <real>scale_factor(1.0)
END [INITIAL CONDITION]
```

Presto supports a general initialization procedure for setting the value of any variable. This procedure can be used to set material state variables, shell thickness, initial stress, etc. There is minimal checking in Presto, however, to ensure that the changes made yield a consistent system. There is also no guarantee that the changes will not be overwritten or misinterpreted by some other internal routine depending on what variable is being changed. Thus, caution is advised when using this capability.

The `INITIAL CONDITION` command block, which appears in the region scope, is used to select a method and set values for initializing a variable. The command block specifies the initial value of a global variable or a variable associated with a set of mesh entities, i.e., nodes, edges, faces, or elements. The user has three options for setting initial values: with a constant magnitude, with an input mesh variable, or by a user subroutine. Only one of these three options can be specified in the command block.

The command block contains five groups of commands—mesh-entity set, variable identification, magnitude, input mesh variable, and user subroutine. In addition to the command lines in the five groups, there is one additional command line: `SCALE FACTOR`. Following are descriptions of the different command groups and the `SCALE FACTOR` command line.

6.2.1 Mesh-Entity Set Commands

The {`mesh-entity set commands`} portion of the `INITIAL CONDITION` command block specifies the nodes, element faces, or elements associated with the variable to be initialized. This portion of the command block can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes, element faces, or elements. See Section 6.1 for more information about the use of these command lines for mesh entities. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.2.2 Variable Identification Commands

Any variable used in the `INITIAL CONDITION` command block must exist in Presto. The variable can be any currently registered variable in Presto or any user-defined variable created with the `USER VARIABLE` command block (see Section 9.2.4).

There are two command lines that identify the variable:

```
INITIALIZE VARIABLE NAME = <string>var_name
VARIABLE TYPE = [NODE|EDGE|FACE|ELEMENT|GLOBAL]
```

The `INITIALIZE VARIABLE NAME` command line gives the name of the variable for which initial values are being assigned. As mentioned, the string `var_name` must be some variable known to Presto; it cannot be an arbitrary user-selected name.

The `VARIABLE TYPE` command line provides additional information about the variable being initialized. The options `NODE`, `EDGE`, `FACE`, `ELEMENT`, and `GLOBAL` on the command line indicate whether the variable is, respectively, a nodal, edge, face, element, or global quantity. One of these options must appear in the `VARIABLE TYPE` command line.

Both of these command lines are required regardless of the option selected to set values for the variable.

6.2.3 Constant Magnitude Command

If the constant magnitude option is used, one or more initial values are specified directly in the command block. Following is the command line used for the constant magnitude option:

```
MAGNITUDE = <real list>initial_values
```

The `initial_values` specified on the `MAGNITUDE` command line will set the values for the variable given by `var_name` in the `INITIALIZE VARIABLE NAME` command line. The number of values is dependent on the type of the variable specified in the `INITIALIZE VARIABLE NAME` command line. For example, if the user wanted to initialize the velocity at a set of nodes, three quantities would have to be specified since the velocity at a node is a vector quantity. If the user wanted to initialize the stress tensor for a set of uniform-gradient, eight-node hexahedral elements, six quantities would have to be specified since the stress tensor for this element type is described with six values.

6.2.4 Input Mesh Command

If the input mesh option is used, the initial values will be read from a variable that exists in a mesh file. As an example, suppose the mesh file contains a set of element temperatures. These temperature values (which can vary for each element) can be used to initialize a temperature value associated with each element.

Following are the command lines related to the input mesh option:

```
READ VARIABLE = <string>mesh_var_name
```

```
TIME = <real>temp_time
```

The string `mesh_var_name` must match the name of the variable in the mesh file. The number of values associated with the variable in the mesh file must be the same number associated with the variable name specified in the `INITIALIZE VARIABLE NAME` command line. For example, if the variable specified by the `INITIALIZE VARIABLE NAME` has a single value, then the variable specified in the mesh file must also have a single value. You may select the variable at a specific time by using the `TIME` command line. If the specified time on the `TIME` command line does not correspond exactly to a time on the mesh file, the data on the mesh file will be interpolated as needed to obtain the initial values.

The variable name used on the mesh file can be arbitrary. The name can be identical to or different from the registered variable name specified on the `INITIALIZE VARIABLE NAME` command line.

6.2.5 User Subroutine Commands

If the user subroutine option is used, the initial values will be calculated by a subroutine that is written by the user explicitly for this purpose. The subroutine will be called by Presto at the appropriate time to perform the calculations.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

Only one of the first three command lines listed above can be specified in the command block. The particular command line selected depends on the mesh-entity type of the variable being initialized. For example, variables associated with nodes would be initialized if you are using the `NODE SET SUBROUTINE` command line, variables associated with faces if you are using the `SURFACE SUBROUTINE` command line, and variables associated with elements if you are using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the selected subroutine command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

The application of user subroutines for variable initialization is essentially the same as the application of user subroutines in general. See Section 6.3.7 and Chapter 9 for more details on implementing the user subroutine option.

When the user subroutine option is used for variable initialization, the user subroutine is called only once. Also, when a user subroutine is being used, the returned value is the new (initial) variable value at each mesh entity, and the flags array is ignored.

6.2.6 Additional Command

This command line provides an additional option for the `INITIAL CONDITION` command block:

```
    SCALE FACTOR = <real>scale_factor(1.0)
```

Any initial value can be scaled by use of the `SCALE FACTOR` command line. An initial value generated by any one of the three initial-value-setting options in this command block (i.e., constant magnitude, input mesh, or user subroutine) will be scaled by the real value `scale_factor`.

6.3 Kinematic Boundary Conditions

The various kinematic boundary conditions available in Presto are described in this section. The kinematic boundary conditions are nested inside the region scope.

6.3.1 Fixed Displacement Components

```
BEGIN FIXED DISPLACEMENT
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # component commands
  COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z
  #
  # additional command
  ACTIVE PERIODS = <string list>period_names
END [FIXED DISPLACEMENT]
```

The `FIXED DISPLACEMENT` command block fixes displacement components (X, Y, Z, or some combination thereof) for a set of nodes. This command block contains two groups of commands—node set and component. Each of these command groups is basically independent of the other. In addition to the command lines in the two command groups, there is an additional command line: `ACTIVE PERIODS`. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `ACTIVE PERIODS` command line.

6.3.1.1 Node Set Commands

The {`node set commands`} portion of the `FIXED DISPLACEMENT` command block specifies the nodes associated with the boundary condition. This portion of the command block can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
```

```
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.3.1.2 Component Commands

There are two component command lines in the `FIXED DISPLACEMENT` command block:

```
COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z
```

The displacement components that are to be fixed can be specified with either the `COMPONENT` command line or the `COMPONENTS` command line. There can be only one `COMPONENT` command line or one `COMPONENTS` command line in the command block. The user can specify any combination of the components to be fixed, as in `X`, `Z`, `X Z`, `Y X`, etc.

6.3.1.3 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `FIXED DISPLACEMENT` command block:

```
ACTIVE PERIODS = <string list>period_names
```

This command line determines when the boundary condition is active. See Section 2.5 for more information about this optional command line.

6.3.2 Prescribed Displacement

```
BEGIN PRESCRIBED DISPLACEMENT
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z |
    CYLINDRICAL AXIS = <string>defined_axis |
    RADIAL AXIS = <string>defined_axis
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESCRIBED DISPLACEMENT]
```

The `PRESCRIBED DISPLACEMENT` command block prescribes a displacement field for a given set of nodes. The displacement field associates a vector giving the magnitude and direction of the displacement with each node in the node set. The displacement field may vary over time and space. If the displacement field has only a time-varying magnitude and uses one of four methods for setting direction, the function option in the above command block can be used to specify the displacement field. If the displacement field is more complex, a user subroutine option is used to specify the displacement field. You cannot use both the function option and the user subroutine option in the same command block.

The `PRESCRIBED DISPLACEMENT` command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the function option or the user subroutine option. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.3.2.1 Node Set Commands

The {`node set commands`} portion of the `PRESCRIBED DISPLACEMENT` command block defines a set of nodes associated with the prescribed displacement field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.3.2.2 Function Commands

If the function option is used, the displacement vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED DISPLACEMENT` command block.

Following are the command lines related to the function option:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z |
  CYLINDRICAL AXIS = <string>defined_axis |
  RADIAL AXIS = <string>defined_axis
FUNCTION = <string>function_name
```

The magnitude of the displacement can be specified in some arbitrary direction, along a component direction (X, Y, or Z), along a cylindrical direction (defined in reference to some axis), or along a radial direction (defined in reference to some axis). Only one of these options (i.e., command lines) is allowed.

- The `DIRECTION` command line is used to specify that the prescribed displacement vector lies along an arbitrary direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope via a `DEFINE DIRECTION` command line.
- `The COMPONENT` command line is used to specify that the prescribed displacement vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component X corresponds to using direction vector (1, 0, 0).
- The `CYLINDRICAL AXIS` command line requires an axis definition that appears in the domain scope. The string `defined_axis` uses an `axis_name` that is defined in the domain scope (via a `DEFINE AXIS` command line). For this option, a radial line will be drawn from a node to the cylindrical axis; the radial line is normal to the cylindrical axis. If we project the motion of any node using the prescribed cylindrical displacement option onto a plane normal to the cylindrical axis, the motion of the node will be a circular path (with the radius equal to the original distance of the node from the axis) around the cylindrical axis.
- The `RADIAL AXIS` command line requires an axis definition that appears in the domain scope. The string `defined_axis` uses an `axis_name` that is defined in the domain scope (via a `DEFINE AXIS` command line). For this option, a radial line is drawn from a node to the radial axis. The prescribed displacement vector lies along this radial line from the node to the radial axis.

The magnitude of the displacement is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the displacement vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.3.2.4.

The function option specifies the displacement only in the prescribed direction. It does not influence the displacement normal to the prescribed direction.

6.3.2.3 User Subroutine Commands

If the user subroutine option is used, the displacement vector may vary spatially at any given time for each of the nodes in the node set associated with the particular

PRESCRIBED DISPLACEMENT command block. The user subroutine option allows for a more complex description of the displacement field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a displacement direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the displacement field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 6.3.2.4.

See Section 6.3.7 and Chapter 9 for more details on implementing the user subroutine option.

6.3.2.4 Additional Commands

These command lines in the PRESCRIBED DISPLACEMENT command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the displacement in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the displacement from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.3.3 Prescribed Velocity

```
BEGIN PRESCRIBED VELOCITY
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z |
    CYLINDRICAL AXIS = <string>defined_axis |
    RADIAL AXIS = <string>defined_axis
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESCRIBED VELOCITY]
```

The `PRESCRIBED VELOCITY` command block prescribes a velocity field for a given set of nodes. The velocity field associates a vector giving the magnitude and direction of the velocity with each node in the node set. The velocity field may vary over time and space. If the velocity field has only a time-varying magnitude and uses one of four methods for setting direction, the function option in the above command block can be used to specify the velocity field. If the velocity field is more complex, a user subroutine option is used to specify the velocity field. You cannot use both the function option and the user subroutine option in the same command block.

The `PRESCRIBED VELOCITY` command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the function option or the user subroutine option. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.3.3.1 Node Set Commands

The {`node set commands`} portion of the `PRESCRIBED VELOCITY` command block defines a set of nodes associated with the prescribed velocity field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.3.3.2 Function Commands

If the function option is used, the velocity vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED VELOCITY` command block.

Following are the command lines related to the function option:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z |
  CYLINDRICAL AXIS = <string>defined_axis |
  RADIAL AXIS = <string>defined_axis
FUNCTION = <string>function_name |
```

The magnitude of the velocity can be specified in some arbitrary direction, along a component direction (X, Y, or Z), along a cylindrical direction (defined in reference to some axis), or along a radial direction (defined in reference to some axis). Only one of these options (i.e., command lines) is allowed.

- The `DIRECTION` command line is used to specify that the velocity vector lies along an arbitrary direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope via a `DEFINE DIRECTION` command line.
- `The COMPONENT` command line is used to specify that the velocity vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component X corresponds to using direction vector (1, 0, 0).
- The `CYLINDRICAL AXIS` command line requires an axis definition that appears in the domain scope. The string `defined_axis` uses an `axis_name` that is defined in the domain scope (via a `DEFINE AXIS` command line). For this option, a radial line will be drawn from a node to the cylindrical axis; the radial line is normal to the cylindrical axis. The velocity vector will lie along a path that is tangent to a circle that lies in a plane normal to the cylindrical axis and has a radius defined by the magnitude of the radial line from the node to the cylindrical axis.
- The `RADIAL AXIS` command line requires an axis definition that appears in the domain scope. The string `defined_axis` uses an `axis_name` that is defined in the domain scope (via a `DEFINE AXIS` command line). For this option, a radial line is drawn from a node to the radial axis. The velocity vector lies along this radial line from the node to the radial axis.

The magnitude of the velocity is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the velocity vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.3.3.4.

The function option specifies the velocity only in the prescribed direction. It does not influence the velocity normal to the prescribed direction.

6.3.3.3 User Subroutine Commands

If the user subroutine option is used, the velocity vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED VELOCITY` command block. The user subroutine option allows for a more complex description of the velocity field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a velocity direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the velocity field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER` Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.3.3.4.

6.3.3.4 Additional Commands

These command lines in the `PRESCRIBED VELOCITY` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the velocity in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the velocity from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.3.4 Prescribed Acceleration

```
BEGIN PRESCRIBED ACCELERATION
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ACCELERATION]
```

The `PRESCRIBED ACCELERATION` command block prescribes an acceleration field for a given set of nodes. The acceleration field associates a vector giving the magnitude and direction of the acceleration with each node in the node set. The acceleration field may vary over time and space. If the acceleration field has only a time-varying component, the function option in the above command block can be used to specify the acceleration field. If the acceleration field has both time-varying and spatially varying components, a user subroutine option is used to specify the acceleration field. You cannot use both the function option and the user subroutine option in the same command block.

The `PRESCRIBED ACCELERATION` command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the function option or the user subroutine option. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.3.4.1 Node Set Commands

The {`node set commands`} portion of the `PRESCRIBED ACCELERATION` command block defines a set of nodes associated with the prescribed acceleration field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.3.4.2 Function Commands

If the function option is used, the acceleration vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED ACCELERATION` command block. The direction of the acceleration vector is constant for all time; the magnitude of the acceleration vector may vary with time, however.

Following are the command lines related to the function option:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The magnitude of the acceleration can be specified either in some arbitrary direction or along a component direction (X, Y, or Z), but not both. The `DIRECTION` command line is used to specify that the acceleration vector lies along an arbitrary direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope (via a `DEFINE DIRECTION` command line). The `COMPONENT` command line is used to specify that the acceleration vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component X corresponds to using direction vector (1, 0, 0).

The magnitude of the acceleration is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the acceleration vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.3.4.4.

The function option specifies the acceleration only in the prescribed direction. It does not influence the acceleration normal to the prescribed direction.

6.3.4.3 User Subroutine Commands

If the user subroutine option is used, the acceleration vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED ACCELERATION` command block. The user subroutine option allows for a more complex description of the acceleration field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define an acceleration direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the acceleration field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines

are described in Section 9.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 6.3.4.4.

See Section 6.3.7 and Chapter 9 for more details on implementing the user subroutine option.

6.3.4.4 Additional Commands

These command lines in the PRESCRIBED ACCELERATION command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the acceleration in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the acceleration from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.3.5 Fixed Rotation

```
BEGIN FIXED ROTATION
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # component commands
  COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z
  #
  # additional command
  ACTIVE PERIODS = <string list>periods_names
END [FIXED ROTATION]
```

The `FIXED ROTATION` command block fixes rotation about direction components (X, Y, Z, or some combination thereof) for a set of nodes. This command block contains two groups of commands—node set and component. Each of these command groups is basically independent of the other. In addition to the command lines in the two command groups, there is an additional command line: `ACTIVE PERIODS`. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `ACTIVE PERIODS` command line.

6.3.5.1 Node Set Commands

The {`node set commands`} portion of the command block specifies the nodes associated with the boundary condition. This portion of the command block can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.3.5.2 Component Commands

There are two component command lines in the `FIXED ROTATION` command block:

```
COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z
```

The rotation components that are to be fixed can be specified with either the `COMPONENT` command line or the `COMPONENTS` command line. There can be only one `COMPONENT` command line or one `COMPONENTS` command line in the command block. The user can specify any combination of the components to be fixed, as in `X`, `Z`, `X Z`, `Y X`, etc.

6.3.5.3 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `FIXED ROTATION` command block:

```
ACTIVE PERIODS = <string list>period_names
```

This command line determines when the boundary condition is active. See Section 2.5 for more information about this optional command line.

6.3.6 Prescribed Rotation

```
BEGIN PRESCRIBED ROTATION
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
   COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ROTATION]
```

The `PRESCRIBED ROTATION` command block prescribes the rotation about an axis for a given set of nodes. The rotation field associates a vector giving the magnitude and direction of the rotation with each node in the node set. The rotation field may vary over time and space. If the rotation field has only a time-varying component, the function option in the above command block can be used to specify the rotation field. If the rotation field has both time-varying and spatially varying components, a user subroutine option is used to specify the rotation field. You cannot use both the function option and the user subroutine option in the same command block.

The `PRESCRIBED ROTATION` command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are two additional command lines: `SCALE FACTOR` and `AC-`

`TIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the function option or the user subroutine option. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.3.6.1 Node Set Commands

The {`node set commands`} portion of the `PRESCRIBED ROTATION` command block defines a set of nodes associated with the prescribed rotation field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.3.6.2 Function Commands

If the function option is used, the rotation vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED ROTATION` command block. The direction of the rotation vector is constant for all time; the magnitude of the rotation vector may vary with time, however.

Following are the command lines related to the function option:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The magnitude of the rotation can be specified either in some arbitrary direction or along a component direction (X, Y, or Z), but not both. The `DIRECTION` command line is used to specify that the rotation vector lies along an arbitrary direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope (via a DEFINE DIRECTION command line). The COMPONENT command line is used to specify that the rotation vector lies along one of the component directions. The COMPONENT command line is a shortcut to an internally defined direction vector; for example, component X corresponds to using direction vector (1, 0, 0).

The magnitude of the rotation is specified by the FUNCTION command line. This references a function_name (defined in the domain scope in a DEFINITION FOR FUNCTION command block) that specifies the magnitude of the rotation vector as a function of time. The magnitude can be scaled by use of the SCALE FACTOR command line described in Section 6.3.6.4.

The magnitude of the rotation, as specified by the product of the function and the scale factor, has units of radians per second.

The function option specifies the rotation only in the prescribed direction. It does not influence the rotation normal to the prescribed direction.

6.3.6.3 User Subroutine Commands

If the user subroutine option is used, the rotation vector may vary spatially at any given time for each of the nodes in the node set associated with the particular PRESCRIBED ROTATION command block. The user subroutine option allows for a more complex description of the rotation field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a rotation direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the rotation field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines

are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.3.6.4.

See Section 6.3.7 and Chapter 9 for more details on implementing the user subroutine option.

6.3.6.4 Additional Commands

These command lines in the `PRESCRIBED ROTATION` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the rotation in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the rotation from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.3.7 Subroutine Usage for Kinematic Boundary Conditions

The prescribed kinematic boundary conditions may be defined by a user subroutine. All these conditions use a node set subroutine. See Chapter 9 for an in-depth discussion of user subroutines. The kinematic boundary conditions will be applied to nodes. The subroutine that you write will have to return six output values per node and one output flag per node. The usage of the output values depends on the returned flag value for a node, as follows:

- If the flag value is negative, no constraint will be applied to the node.
- If the flag value is equal to zero, the constraint will be absolute. All components of the boundary condition will be specified. For example, suppose you have written a user subroutine to be called from a prescribed displacement subroutine. The prescribed displacements are to be passed through an array `output_values`. For a given node `inode`, the `output_values` array would have the following values:

```
output_values(1,inode) = displacement in x at inode
output_values(2,inode) = displacement in y at inode
output_values(3,inode) = displacement in z at inode
output_values(4,inode) = not used
output_values(5,inode) = not used
output_values(6,inode) = not used
```

- If the flag value is equal to one, the constraint will be a specified amount in a given direction. For example, suppose you have written a user subroutine to be called from a prescribed displacement subroutine. The prescribed displacements are to be passed through an array `output_values`. For a given node `inode`, the `output_values` array would have the following values:

```
output_values(1,inode) = magnitude of displacement
output(values(2,inode) = not used
output_values(3,inode) = not used
output_values(4,inode) = x component of direction vector
output_values(5,inode) = y component of direction vector
output_values(6,inode) = z component of direction vector
```

The direction in which the constraint will act is given by `output_values` 4 through 6 for `inode`. The magnitude of the displacement in the specified direction is given by `output_values` 1 at `inode`. To compute the constraint, Presto first normalizes the direction vector. Next, Presto multiplies the normalized direction vector by the magnitude of the displacement and applies the resultant constraint vector.

Displacements or velocities orthogonal to the prescribed direction will not be constrained. (This is true regardless of whether or not one uses a user subroutine for the prescribed kinematic boundary conditions.) Take the case of a prescribed displacement condition. The displacement orthogonal to a prescribed direction of motion depends on the internal and external forces orthogonal to the prescribed direction. Displacement orthogonal to the prescribed direction may or may not be zero.

6.4 Initial Velocity Conditions

```
BEGIN INITIAL VELOCITY
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # direction commands
  COMPONENT = <string>X|Y|Z |
    DIRECTION = <string>defined_direction
  MAGNITUDE = <real>magnitude_of_velocity
  #
  # angular velocity commands
  CYLINDRICAL AXIS = <string>defined_axis
  ANGULAR VELOCITY = <real>angular_velocity
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
END [INITIAL VELOCITY]
```

The `INITIAL VELOCITY` command block specifies an initial velocity field for a set of nodes. There are two simple options for specifying the initial velocity field: by direction and by angular velocity. The user subroutine option available is also available to specify an initial velocity. You may use only one of the available options—direction, angular velocity, or user subroutine.

The `INITIAL VELOCITY` command block contains four groups of commands—node set, direction, angular velocity, and user subroutine. Command lines associated with the node set commands must appear. As mentioned, command lines associated with one of the options must also appear. Following are descriptions of the different command groups.

6.4.1 Node Set Commands

The {`node set commands`} portion of the `INITIAL VELOCITY` command block defines a set of nodes associated with the initial velocity field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.4.2 Direction Commands

If the direction option is used, the initial velocity is applied along a defined direction with a specific magnitude. Following are the command lines for the direction option:

```
COMPONENT = <string>X|Y|Z |
  DIRECTION = <string>defined_direction
MAGNITUDE = <real>magnitude_of_velocity
```

The direction of the velocity vector can be specified by either the `COMPONENT` command line or the `DIRECTION` command line. If the `COMPONENT` command line is used, the velocity is specified in one of the directions (X, Y, or Z) in the global coordinate system. If the `DIRECTION` command line is used, the initial velocity is specified along a user-defined direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope (via a `DEFINE DIRECTION` command line).

The magnitude of the initial velocity is given by the `MAGNITUDE` command line with the real value `magnitude_of_velocity`.

Either the `COMPONENT` command line or the `DIRECTION` command line must be specified with the `MAGNITUDE` command line if you use the direction option.

6.4.3 Angular Velocity Commands

If the angular velocity option is used, the initial velocity is applied as an initial angular velocity about some axis. Following are the command lines for the angular velocity option:

```
CYLINDRICAL AXIS = <string>defined_axis
ANGULAR VELOCITY = <real>angular_velocity
```

The axis about which the body is initially rotating is given by the `CYLINDRICAL AXIS` command line. The string `defined_axis` uses an `axis_name` that is defined in the domain scope (via a `DEFINE AXIS` command line).

The magnitude of the angular velocity about this axis is given by the `ANGULAR VELOCITY` command line with the real value `angular_velocity`. This value is specified in units of radians per unit of time. Typically, the value for the angular velocity will be radians per second.

Both the `CYLINDRICAL AXIS` command line and the `ANGULAR VELOCITY` command line are required if you use the angular velocity option.

6.4.4 User Subroutine Commands

If the user subroutine option is used, the initial velocity vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `INITIAL CONDITION` command block. The user subroutine option allows for a more complex description of the initial velocity field than do the direction and angular-velocity options, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a velocity direction and a magnitude for every node to which the initial velocity field will be applied. The subroutine will be called by Presto at the appropriate time to generate the initial velocity field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

See Section 6.3.7 and Chapter 9 for more details on implementing the user subroutine option.

6.5 Force Boundary Conditions

A variety of force boundary conditions are available in Presto. This section describes these boundary conditions.

6.5.1 Pressure

```
BEGIN PRESSURE
  # {surface set commands}
  SURFACE = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  #
  # function command
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  SURFACE SUBROUTINE = <string>subroutine_name |
    NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external pressure sources
  READ VARIABLE = <string>variable_name
  OBJECT TYPE = <string>NODE|FACE(NODE)
  TIME = <real>time
  FIELD VARIABLE = <string>field_variable
  #
  # output external forces from pressure
  EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
    <string>variable_name
  #
  # additional commands
  USE DEATH = <string>death_name
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESSURE]
```

The `PRESSURE` command block applies a pressure to each face in the associated surfaces. The pressure field can either be constant over the faces and vary in time, or it can be determined by a user subroutine. If the pressure field is constant over the faces and has only a time-varying component, the function option in the above command block can be used to specify the pressure field. If the pressure field has both time-varying and spatially varying components, a user subroutine option is used to specify the pressure field. The pressure field may also be obtained from a mesh file or from another SIERRA code through a transfer operator. You can use only one of these four options—function, user subroutine, mesh file, transfer from another code—to specify the pressure field.

Currently, the `PRESSURE` command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedrons, four-node tetrahedrons, eight-node tetrahedrons, etc.), membranes, and shells.

A pressure boundary condition generates nodal forces that are summed into the external force vector that is used to calculate the motion of a body. The external force vector contains the contribution from all forces acting on the body. There is an option in the `PRESSURE` command block to save information about the contribution to the external force vector due only to pressure loads. This option does not change the magnitude or time history of the pressure load (regardless of how they are defined), but merely stores information in a user-accessible variable.

The `PRESSURE` command block contains five groups of commands—surface set, function, user subroutine, external pressure, and output external forces. Each of these command groups is basically independent of the others. In addition to the command lines in the five command groups, there are three additional command lines: `USE DEATH`, `SCALE FACTOR` and `ACTIVE PERIODS`. The `USE DEATH` command line links the pressure boundary condition to an element death instance. The `SCALE FACTOR` command line can be used in conjunction with either the function option or the user subroutine option. The `ACTIVE PERIODS` command line is used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups and the `USE DEATH`, `SCALE FACTOR`, and `ACTIVE PERIODS` command lines.

6.5.1.1 Surface Set Commands

The {`surface set commands`} portion of the `PRESSURE` command block defines a set of surfaces associated with the pressure field and can include some combination of the following command lines:

```
SURFACE = <string list>surface_names
REMOVE SURFACE = <string list>surface_names
```

In the `SURFACE` command line, you can list a series of surfaces through the string list `surface_names`. There must be at least one `SURFACE` command line in the command block. The `REMOVE SURFACE` command line allows you to delete surfaces from the set specified in the `SURFACE` command line(s) through the string list `surface_names`. See Section 6.1 for more information about the use of these command lines for creating a set of surfaces used by the boundary condition.

6.5.1.2 Function Commands

If the function option is used, the pressure vector at any given time is the same for all surfaces associated with the particular `PRESSURE` command block. The direction of the pressure vector is constant for all time; the magnitude of the pressure vector may vary with time, however.

Following is the command line related to the function option:

```
FUNCTION = <string>function_name
```

The magnitude of the pressure is in the opposite direction to the outward normals of the faces that define the surfaces. The magnitude of the pressure is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the pressure vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.5.1.6.

6.5.1.3 User Subroutine Commands

If the user subroutine option is used, the pressure may vary spatially at any given time for each of the surfaces associated with the particular `PRESSURE` command block. The user subroutine option allows for a more complex description of the pressure field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a pressure for every face to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the pressure field.

Following are the command lines related to the user subroutine option:

```
SURFACE SUBROUTINE = <string>subroutine_name |
   NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `SURFACE SUBROUTINE` command line or the `NODE SET` subroutine command line. The string `subroutine_name` in both command lines is the name of a FORTRAN subroutine that is written by the user. The particular command line selected depends on the mesh-entity type for which the pressure field is being calculated. Associating pressure values with faces would require the use of a `SURFACE SUBROUTINE` command line. Associating pressure values with nodes would require the use of a `NODE SET SUBROUTINE` command line.

Following the selected subroutine command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.5.1.6.

Usage requirements. Following are the usage requirements for the two types of subroutines:

- The surface subroutine operates on a group of faces. The subroutine that you write will return one output value per face. Suppose you write a user subroutine that returns the pressure information through an array `output_value`. The value `output_value(1,iface)` corresponds to the average pressure on face `iface`. The values of the flags array are not used.

- The node set subroutine that you write will return one value per node. Suppose you write a user subroutine that returns the pressure information through an array `output_value`. The return value `output_value(1,inode)` is the pressure at the node `inode`. The total pressure on the each face is found by integrating the pressures at the nodes. The values of the flags array are not used.

See Chapter 9 for more details on implementing the user subroutine option.

6.5.1.4 External Pressure Sources

Pressure may be obtained from two different external sources. The first option for obtaining pressure from an external source uses a mesh file. The commands for obtaining pressure information from a mesh file are as follows:

```
READ VARIABLE = <string>variable_name
OBJECT TYPE = <string>NODE|FACE(NODE)
TIME = <real>time
```

The `READ VARIABLE` command line specifies the name of the variable on the mesh file, `variable_name`, that is used to prescribe the pressure field. The `OBJECT TYPE` command line specifies whether the pressure field on the mesh file is specified for nodes (the mesh object type is `NODE`) or for faces (the mesh object type is `FACE`). If the `OBJECT TYPE` command line is not present, it is assumed that the variable is for nodes. If the `TIME` command line is present, only the pressure field information at a given time, as set by the `time` parameter, is read from the mesh file. If the `TIME` command line is not present, the pressure field information for all times is read. Pressure field information will then be interpolated as necessary during an analysis.

The second option for obtaining pressure from and external sources relies on the transfer of information from another SIERRA code. The command for obtaining pressure information by transfer from another code is

```
FIELD VARIABLE = <string>variable_name ,
```

where `variable_name` is the name of the registered variable where pressure information is to be stored. The pressure information will be transferred into this registered variable from another SIERRA code via a transfer operator.

6.5.1.5 Output Command

This command line lets the user create a variable that stores information about the contribution to the external force vector at a node arising solely from a pressure:

```
EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
  <string>variable_name
```

If the above command line appears in a `PRESSURE` command block, then there will be a variable created with whatever name the user specifies for `variable_name`. The variable defines a three-dimensional vector at each node associated with this particular command block. The three-dimensional vector at each node represents the external force due solely to the pressure on the elements attached to that node. For example, if one of the nodes associated with this particular command block has four elements attached to it and each element has a pressure load, then the external force contribution at the node would be summed from the pressure load for all four elements.

Once this variable for the external force contribution from a pressure load is specified, it may be used like any other registered nodal variable. The user can, for example, specify the variable as a nodal variable to be output in a `RESULTS OUTPUT` command block. Or the user can reference the variable in a user subroutine.

6.5.1.6 Additional Commands

These command lines in the PRESSURE command block provide additional options for the boundary condition:

```
USE DEATH = <string>death_name
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The USE DEATH command line links the pressure boundary condition to an element death instance. The string death_name must match a name used in an ELEMENT DEATH command block. When elements are killed by the named element death instance, the pressure boundary condition will be applied to the newly exposed faces.

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the pressure in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the pressure from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.5.2 Traction

```
BEGIN TRACTION
  # {surface set commands}
  SURFACE = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  #
  # function commands
  DIRECTION = <string>direction_name
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # SIMOD commands
  USE SIMOD MODEL = <string>Simod_model_name
  REFERENCE PLANE AXIS = <string>axis_direction
  REFERENCE PLANE T1 DIRECTION = <string>t1_direction
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  #
END [TRACTION]
```

The `TRACTION` command block applies a traction to each face in the associated surfaces. The traction has units of force per unit area. (A traction, unlike a pressure, may not necessarily be in the direction of the normal to the face to which it is applied.) The given traction is integrated over the surface area of a face.

The traction field can be determined by a SIERRA function, it can be determined by a user subroutine, or it can be determined through the use of Shared Interface MODels (SIMOD). If the traction field is constant over the faces and has only a time-varying component, the function option in the above command block can be used to specify the traction field. If the traction field has both time-varying and spatially varying components, a user subroutine option is used to specify the traction field. The SIMOD option lets you work with surface-physics models to define the traction field on a surface. You should consult with Section 10.2 for a detailed explanation of the use of a SIMOD model and which surface-physics models are currently available in Presto. We will not discuss the details of the use of the SIMOD option in this chapter.

You must use only one of the three options—function, user subroutine, or SIMOD—in a `TRACTION` command block.

Currently, the `TRACTION` command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedrons, four-node tetrahedrons, eight-node tetrahedrons, etc.), membranes, and shells.

The `TRACTION` command block contains four groups of commands—surface set, function, user subroutine, and SIMOD. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with the function option, the user subroutine option, or the SIMOD option. The `ACTIVE PERIODS` command line is used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.5.2.1 Surface Set Commands

The {`surface set commands`} portion of the `TRACTION` command block defines a set of surfaces associated with the traction field and can include some combination of the following command lines:

```
SURFACE = <string list>surface_names
REMOVE SURFACE = <string list>surface_names
```

In the `SURFACE` command line, you can list a series of surfaces through the string list `surface_names`. There must be at least one `SURFACE` command line in the command block. The `REMOVE SURFACE` command line allows you to delete surfaces from the set specified in the `SURFACE` command line(s) through the string list `surface_names`. See Section 6.1 for more information about the use of these command lines for creating a set of surfaces used by the boundary condition.

6.5.2.2 Function Commands

If the function option is used, the traction vector at any given time is the same for all surfaces associated with the particular `TRACTION` command block. The direction of the traction vector is constant for all time; the magnitude of the traction vector may vary with time, however.

Following are the command lines related to the function option:

```
DIRECTION = <string>direction_name
FUNCTION = <string>function_name
```

The magnitude of the traction can be specified along a component direction (X, Y, or Z). The `DIRECTION` command line is used to specify that the traction vector lies along an arbitrary direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope (via a `DEFINE DIRECTION` command line).

The magnitude of the traction is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the traction vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.5.2.4.

6.5.2.3 User Subroutine Commands

If the user subroutine option is used, the traction vector may vary spatially at any given time for each of the surfaces associated with the particular `TRACTION` command block. The user subroutine option allows for a more complex description of the traction field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a traction for every face to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the traction field.

Following is the command line related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET` subroutine command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user. Associating traction values with nodes requires the use of a `NODE SET SUBROUTINE` command line.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines

are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.5.2.4.

Usage requirements for the node set subroutine. The node set subroutine that you write will return six values per node. Suppose you have written a user subroutine that passes the output values through an array `output_values`. For a given node inode, the `output_values` array would have the following values:

```
output_values(1,inode) = magnitude of traction
output(values(2,inode) = not used
output_values(3,inode) = not used
output_values(4,inode) = x component of direction vector
output_values(5,inode) = y component of direction vector
output_values(6,inode) = z component of direction vector
```

The direction in which the traction will act is given by `output_values` 4 through 6 for `inode`. The magnitude of the traction in the specified direction is given by `output_values` 1 at `inode`. The total force on each node is found by integrating the local nodal tractions using the associated directions, which are normalized by Presto, over the face areas. The values of the flags array are not used.

See Chapter 9 for more details on implementing the user subroutine option.

6.5.2.4 Additional Commands

These command lines in the `TRACTION` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the traction in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the traction from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.5.3 Prescribed Force

```
BEGIN PRESCRIBED FORCE
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESCRIBED FORCE]
```

The `PRESCRIBED FORCE` command block prescribes a force field for a given set of nodes. The force field associates a vector giving the magnitude and direction of the force with each node in the node set. The force field may vary over time and space. If the force field has only a time-varying component, the function option in the above command block can be used to specify the force field. If the force field has both time-varying and spatially varying components, a user subroutine option is used to specify the force field. You cannot use both the function option and the user subroutine option in the same command block.

The `PRESCRIBED FORCE` command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PE-`

RIODS. The SCALE FACTOR command line can be used in conjunction with either the function option or the user subroutine option. The ACTIVE PERIODS command line is used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups and the SCALE FACTOR and ACTIVE PERIODS command lines.

6.5.3.1 Node Set Commands

The {node set commands} portion of the PRESCRIBED FORCE command block defines a set of nodes associated with the prescribed force field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one NODE SET, SURFACE, BLOCK, or INCLUDE ALL BLOCKS command line in the command block.

6.5.3.2 Function Commands

If the function option is used, the force vector at any given time is the same for all nodes in the node set associated with the particular PRESCRIBED FORCE command block. The direction of the force vector is constant for all time; the magnitude of the force vector may vary with time, however.

Following are the command lines related to the function option:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The magnitude of the force can be specified either in some arbitrary direction or along a component direction (X, Y, or Z), but not both. The DIRECTION command line is used to specify that the force vector lies along an arbitrary direction. The string defined_direction uses a direction_name that has been defined in the domain scope (via a `DEFINE DIRECTION` command line). The `COMPONENT` command line is used to specify that the acceleration vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component X corresponds to using direction vector (1, 0, 0).

The magnitude of the force is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the force vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.5.3.4.

The function option specifies the force only in the prescribed direction. It does not set the force in any direction normal to the prescribed direction.

6.5.3.3 User Subroutine Commands

If the user subroutine option is used, the force vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED FORCE` command block. The user subroutine option allows for a more complex description of the force field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a force direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the force field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PA-`

RAMETER, and SUBROUTINE STRING PARAMETER Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 6.5.3.4.

Usage requirements for the node set subroutine. The subroutine that you write will return three output values per node. Suppose you write a user subroutine that passes the output values through an array output_values. For a given node inode, the output_values array would have the following values:

```
output_values(1,inode) = x component of force at inode
output_values(2,inode) = y component of force at inode
output_values(3,inode) = z component of force at inode
```

The three components of the force vector are given in output_values 1 through 3. The values of the flags array are ignored.

See Chapter 9 for more details on implementing the user subroutine option.

6.5.3.4 Additional Commands

These command lines in the PRESCRIBED FORCE command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the force in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the force from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.5.4 Prescribed Moment

```
BEGIN PRESCRIBED MOMENT
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [PRESCRIBED MOMENT]
```

The `PRESCRIBED MOMENT` command block prescribes a moment field for a given set of nodes. Moments can only be defined for nodes attached to beam or shell elements. The moment field associates a vector giving the magnitude and direction of the moment with each node in the node set. If the moment field has only a time-varying component, the function option in the above command block can be used to specify the moment field. If the moment field has both time-varying and spatially varying components, a user subroutine option is used to specify the moment field. You cannot use both the function option and the user subroutine option in the same command block.

The `PRESCRIBED MOMENT` command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the function option or the user subroutine option. The `ACTIVE PERIODS` command line is used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.5.4.1 Node Set Commands

The {node set commands} portion of the `PRESCRIBED MOMENT` command block defines a set of nodes associated with the prescribed moment field and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

6.5.4.2 Function Commands

If the function option is used, the moment vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED MOMENT` command block. The direction of the moment vector is constant for all time; the magnitude of the moment vector may vary with time, however.

Following are the command lines related to the function option:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The magnitude of the moment can be specified either about some arbitrary direction or about a component direction (X, Y, or Z), but not both. The `DIRECTION` command line is used to specify that the moment vector lies along an arbitrary direction. The string `defined_direction` uses a `direction_name` that has been defined in the domain scope (via a `DEFINE DIRECTION` command line). The `COMPONENT` command line is used to specify that the moment vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component X corresponds to using direction vector (1, 0, 0).

The magnitude of the moment is specified by the `FUNCTION` command line. This references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the moment vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.5.4.4.

The function option specifies the moment only about the prescribed direction. It does not influence the moment about any direction normal to the prescribed direction.

6.5.4.3 User Subroutine Commands

If the user subroutine option is used, the moment vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED MOMENT` command block. The user subroutine option allows for a more complex description of the moment field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a moment direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the moment field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines

are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.5.4.4.

Usage requirements for the node set subroutine. The subroutine that you write will return three output values per node. Suppose you write a user subroutine that passes the output values through an array `output_values`. For a given node `inode`, the `output_values` array would have the following values:

```
output_values(1,inode) = moment about x-direction at inode
output_values(2,inode) = moment about y-direction at inode
output_values(3,inode) = moment about z-direction at inode
```

The three components of the moment vector are given in `output_values` 1 through 3. The values of the flags array are ignored.

See Chapter 9 for more details on implementing the user subroutine option.

6.5.4.4 Additional Commands

These command lines in the `PRESCRIBED MOMENT` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function option or the user subroutine option. For example, if the magnitude of the moment in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the moment from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.6 Prescribed Temperature

```
     BEGIN PRESCRIBED TEMPERATURE
       # {block set commands}
       BLOCK = <string list>block_names
       INCLUDE ALL BLOCKS
       REMOVE BLOCK = <string list>block_names
       #
       # function command
       FUNCTION = <string>function_name
       #
       # user subroutine commands
       NODE SET SUBROUTINE = <string>subroutine_name
       SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
       SUBROUTINE REAL PARAMETER: <string>param_name
         = <real>param_value
       SUBROUTINE INTEGER PARAMETER: <string>param_name
         = <integer>param_value
       SUBROUTINE STRING PARAMETER: <string>param_name
         = <string>param_value
       #
       # read variable commands
       READ VARIABLE = <string>variable_name
       TIME = <real>time
       #
       # additional commands
       SCALE FACTOR = <real>scale_factor(1.0)
       ACTIVE PERIODS = <string list>period_names
    END [PRESCRIBED ACCELERATION]
```

The `PRESCRIBED TEMPERATURE` command block prescribes a temperature field for a given set of nodes. The prescribed temperature is for each node in the node set. The temperature field may vary over time and space. If the temperature field has only a time-varying component, the function option in the above command block can be used to specify the acceleration field. If the temperature field has both time-varying and spatially varying components, a user subroutine option can be used to specify the temperature field. Finally, you may also read the temperature as a variable from the mesh file. You can select only one of these options—function, user subroutine, or read variable—in a command block.

Temperature is applied to nodes, but it is frequently used at the element level, such as in the case for thermal strains. If the temperatures are used at the element level, the nodal values are averaged (depending on element) connectivity to produce an element temperature. The temperatures must be defined for all the nodes defining the connectivity for any given element. For this reason, we use block commands to derive a set of nodes at which to define temperatures. If the temperatures are used on an element basis, then the temperature at all the necessary nodes will be defined.

The `PRESCRIBED TEMPERATURE` command block contains four groups of commands—block set, function, user subroutine, and read variable. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are two additional command lines: `SCALE FACTOR` and `ACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with the function option, the user subroutine option, or the read variable option. The `ACTIVE PERIODS` command line is used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the `SCALE FACTOR` and `ACTIVE PERIODS` command lines.

6.6.1 Block Set Commands

The {`block set commands`} portion of the `PRESCRIBED TEMPERATURE` command block defines a set of nodes associated with the prescribed temperature field and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes derived from some combination of element blocks. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `BLOCK` or `INCLUDE ALL BLOCKS` command line in the command block.

6.6.2 Function Commands

If the function option is used, the temperature at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED TEMPERATURE` command block. The command line

```
FUNCTION = <string>function_name
```

references a `function_name` (defined in the domain scope in a `DEFINITION FOR FUNCTION` command block) that specifies the temperature as a function of time. The temperature can be scaled by use of the `SCALE FACTOR` command line described in Section 6.6.5.

6.6.3 User Subroutine Commands

If the user subroutine option is used, the temperature field may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED TEMPERATURE` command block. The user subroutine option allows for a more complex description of the temperature field than does the function option, but the user subroutine option also requires that you write a user subroutine to implement this capability. The user subroutine will be used to define a temperature for every node to which the boundary condition will be applied. The subroutine will be called by Presto at the appropriate time to generate the temperature field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

The temperature set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.6.5.

See Chapter 9 for more details on implementing the user subroutine option.

6.6.4 Read Variable Commands

If the read variable option is used, the temperature field will be read from a variable defined in the mesh file. The following command lines are used for the read variable option:

```
READ VARIABLE = <string>mesh_var_name
TIME = <real>temp_time
```

The string `mesh_var_name` must correspond to the variable name for the temperature field present in the mesh file. The temperature field may be specified at several different times on the mesh file. You may select the temperature field at a specific time by using the `TIME` command line. If the specified time on the `TIME` command line does not correspond exactly to a time on the mesh file, the data on the mesh file will be interpolated as needed to obtain the nodal temperatures.

The temperature set by the read variable option can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.6.5.

6.6.5 Additional Commands

These command lines in the `PRESCRIBED TEMPERATURE` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all temperature values of the field defined by the function option, the user subroutine option, or the read variable option. For example, if the temperature in a time history function is given as 100.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the temperature from time 1.0 to 2.0 is 50.25. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` command line determines when the boundary condition is active. See Section 2.5 for more information about this command line.

6.7 Specialized Boundary Conditions

There are a number of specialized boundary conditions implemented in Presto. Some of them enforce kinematic conditions, and some result in the application of loads.

6.7.1 Gravity

```
BEGIN GRAVITY
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  DIRECTION = <string>defined_direction
  FUNCTION = <string>function_name
  GRAVITATIONAL CONSTANT = <real>g_constant
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
END [GRAVITY]
```

The `GRAVITY` command block is used to specify a gravity load that is applied to all nodes selected within a command block. The gravity load boundary condition uses the function and scale (gravitational constant and scale factor) information to generate a body force at a node based on the mass of the node. Multiple `GRAVITY` command blocks can be defined on different sets of nodes. If two different `GRAVITY` command blocks reference the same node, the node will have gravity loads applied by both of the command blocks. Care must be taken to make sure you do not apply multiple gravity loads to one block if you only want one gravity load condition applied.

The {`node set commands`} portion of the `GRAVITY` command block defines a set of nodes associated with the gravity load and can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 6.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

The magnitude of the gravity load is specified in some arbitrary direction via the `DIRECTION` command line. The string `defined_direction` uses a `direction_name` defined in the domain scope in a `DEFINE DIRECTION` command line.

The strength of the gravitational field can be varied with time by using the `FUNCTION` command line. This command line references a `function_name` defined in the domain scope in a `DEFINITION FOR FUNCTION` command block.

A gravitational constant is specified by the `GRAVITATIONAL CONSTANT` command line in the real value `g_constant`. For example, the gravitational constant in units of inches and seconds would be 386.4 inches per second squared. You must set this quantity based on the actual units for your model.

The dependent variables in the function can be scaled by the real value `scale_factor` in the `SCALE FACTOR` command line. At any given time, the strength of the gravitational field is a product of the gravitational constant, the value of the function at that time, and the scale factor.

The `ACTIVE PERIODS` command line provides an additional option for the gravity load condition. This command line can activate or deactivate the gravity load for certain time periods. See Section 2.5 for more information about this command line.

6.7.2 Cavity Expansion

```
BEGIN CAVITY EXPANSION
  EXPANSION RADIUS = <string>SPHERICAL|CYLINDRICAL
    (spherical)
  SURFACE = <string list>surface_ids
  REMOVE SURFACE = <string list>surface_ids
  FREE SURFACE = <real>top_surface_zcoord
    <real>bottom_surface_zcoord
  NODE SETS TO DEFINE BODY AXIS =
    <string>nodelist_1 <string>nodelist_id2
  TIP RADIUS = <real>tip_radius
  BEGIN LAYER <string>layer_name
    LAYER SURFACE = <real>top_layer_zcoord
      <real>bottom_layer_zcoord
    PRESSURE COEFFICIENTS = <real>c0 <real>c1 <real>c2
    SURFACE EFFECT = <string>NONE|SIMPLE_ON_OFF(NONE)
    FREE SURFACE EFFECT COEFFICIENTS = <real>coeff1
      <real>coeff2
  END [LAYER <string>layer_name]
  ACTIVE PERIODS = <string list>period_names
END [CAVITY EXPANSION]
```

The `CAVITY EXPANSION` command block is used to apply a cavity expansion boundary condition to a surface on a body. This boundary condition is typically used for earth penetration studies where some type of projectile (penetrator) strikes a target. For a more detailed explanation of the numerical implementation of the cavity expansion boundary condition and the parameters for this boundary condition, consult Reference 1. The cavity expansion boundary condition is a complex boundary condition with several options, and the detailed explanation of the implementation of the boundary condition in the above reference is required reading to fully understand the input parameters for this boundary condition.

There are two types of cavity expansion—cylindrical expansion and spherical expansion. You can select either the spherical or cylindrical option by using the `EXPANSION RADIUS` command line; the default is `SPHERICAL`. Reference 1 describes these two types of cavity expansion.

The boundary condition is applied to the surfaces (`surface_ids`) in the finite element model specified by the `SURFACE` command line. (Any surface specified on the `SURFACE` command line can be removed from the list of surfaces by using a `REMOVE SURFACE` command line.) This boundary condition generates a pressure at a node based on the velocity and surface geometry at the node. Since cavity expansion is essentially a pressure boundary condition, cavity expansion must be specified for a surface.

The target has a top free surface with a normal in the global positive z-direction; the target has a bottom free surface with a normal in the global negative z-direction. The point on the global z-axis intersected by the top free surface is given by the parameter `top_surface_zcoord` on the `FREE SURFACE` command line. The point on the global z-axis intersected by the bottom free surface is given by the parameter `bottom_surface_zcoord` on the `FREE SURFACE` command line.

It is necessary to define two points that lie on the axis (usually the axis of revolution) of the penetrator. These two nodes are specified with the `NODE SETS TO DEFINE BODY AXIS` command line. The first node should be a node toward the tip of the penetrator (`nodelist_1`), and the second node should be a node toward the back of the penetrator (`nodelist_2`). Only one node is allowed in each node set.

It is necessary to compute either a spherical or cylindrical radius for nodes on the surface where the cavity expansion boundary condition is applied. This is done automatically for most nodes. The calculations for these radii break down if the node is close to or at the tip of the axis of revolution of the penetrator. For nodes where the radii calculations break down, a user-defined radius can be specified with the `TIP RADIUS` command line. For more information, consult Reference 1.

Embedded within the target can be any number of layers. Each layer is defined with a `LAYER` command block. The command block begins with

```
BEGIN LAYER <string>layer_name
```

and is terminated with

```
END [LAYER <string>layer_name] ,
```

where the string `layer_name` is a user-selected name for the layer. This name must be unique to all other layer names defined in the `CAVITY EXPANSION` command blocks. The layer properties are defined by several different command lines—`LAYER SURFACE`, `PRESSURE COEFFICIENTS`, `SURFACE EFFECT`, and `FREE SURFACE EFFECT COEFFICIENTS`. These command lines are described next.

```
- LAYER SURFACE = <real>top_layer_zcoord
    <real>bottom_layer_zcoord
```

The layer has a top surface with a normal in the global positive z-direction; the layer has a bottom surface with a normal in the global negative z-direction. In the `LAYER SURFACE` command line, the point on the global z-axis intersected by the top layer surface is given by the parameter `top_layer_zcoord`, and the point on the global z-axis intersected by the bottom layer surface is given by the parameter `bottom_layer_zcoord`.

```
- PRESSURE COEFFICIENTS = <real>c0 <real>c1 <real>c2
```

The value of the pressure at a node is derived from an equation that is quadratic based on some scalar value derived from the velocity vector at the node. The three coefficients for the quadratic equation (`c0`, `c1`, `c2`) in the `PRESSURE COEFFICIENTS` command line define the impact properties of a layer.

- `SURFACE EFFECT = <string>NONE|SIMPLE_ON_OFF(NONE)`

  There can be no surface effects associated with a layer, or there can be a simple on/off surface effect model associated with a layer. The type of surface effect is determined by the `SURFACE EFFECT` command line. The default is no surface effects. If the `SIMPLE_ON_OFF` model is chosen, it is necessary to specify free surface effect coefficients with the `FREE SURFACE EFFECT COEFFICIENTS` command line.

- `FREE SURFACE EFFECT COEFFICIENTS = <real>coeff1 <real>coeff2`

  All the parameters defined in a `LAYER` command block apply to that layer. If a simple on/off surface effect is applied to a layer, the surface effect coefficients are associated with the layer values. The surface effect parameter associated with the top of the layer is `coeff1`; the surface effect parameter associated with the bottom of the layer is `coeff2`.

The `ACTIVE PERIODS` command line provides an additional option for cavity expansion. This command line can activate or deactivate cavity expansion for certain time periods. See Section 2.5 for more information about this command line.

6.7.3 Silent Boundary

```
BEGIN SILENT BOUNDARY
  SURFACE = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  ACTIVE PERIODS = <string list>period_names
END [SILENT BOUNDARY]
```

The `SILENT BOUNDARY` command block is also referred to as a nonreflecting surface boundary condition. A wave striking this surface is not reflected. This boundary condition is implemented with the techniques described in Reference 2. The method described in this reference is excellent at transmitting the low- and medium-frequency content through the boundary. While the method does reflect some of the high-frequency content, the amount of energy reflected is usually minimal. On the whole, the silent boundary condition implemented in Presto is highly effective.

In the `SURFACE` command line, you can list a series of surfaces through the string list `surface_names`. There must be at least one `SURFACE` command line in the command block. The `REMOVE SURFACE` command line allows you to delete surfaces from the set specified in the `SURFACE` command line(s) through the string list `surface_names`. See Section 6.1 for more information about the use of these command lines for creating a set of surfaces used by the boundary condition.

The `ACTIVE PERIODS` command line provides an additional option for the boundary condition. This command line is used to activate or deactivate the boundary condition for certain time periods. See Section 2.5 for more information about this command line.

6.7.4 Spot-Weld

```
BEGIN SPOT WELD
  NODE SET = <string list>nodelist_ids
  REMOVE NODE SET = <string list>nodelist_ids
  SURFACE = <string list>surface_ids
  REMOVE SURFACE = <string list>surface_ids
  SECOND SURFACE = <string>surface_id
  NORMAL DISPLACEMENT FUNCTION =
    <string>function_nor_disp
  NORMAL DISPLACEMENT SCALE FACTOR =
    <real>scale_nor_disp[1.0]
  TANGENTIAL DISPLACEMENT FUNCTION =
    <string>function_tang_disp
  TANGENTIAL DISPLACEMENT SCALE FACTOR =
    <real>scale_tang_disp[1.0]
  FAILURE ENVELOPE EXPONENT = <real>exponent
  FAILURE FUNCTION = <string>fail_func_name
  FAILURE DECAY CYCLES = <integer>number_decay_cycles
  SEARCH TOLERANCE = <real>search_tolerance
  ACTIVE PERIODS = <string list>period_names
END [SPOT WELD]
```

The spot-weld option lets the user model an “attachment” between a node on one surface and a face on another surface. This option models a weld or a small screw or bolt with a force-displacement curve like that shown in Figure 6.1. The displacement shown in the figure is the distance that the node moves from the nearest point on the face as measured in the original configuration. The force shown in the figure is the force at the attachment as a function of the distance between the two attachment points. (The force-displacement curve assumes the two attachment points are originally at the same location and the initial distance is zero.) Two force-displacement curves are required for the spot-weld model. One curve models normal behavior, and the other curve models tangential behavior.

The attachment in Presto is defined between a node on one surface and the closest point on an element face on the other surface. Since a face is used to define one of the attachment points, it is possible to compute a normal vector and a tangent vector associated with the face. This allows us to resolve the displacement (distance) and force into normal and tangential components. With normal and tangential vectors associated with the attachment, the attachment can be characterized for the case of pure tension and pure shear.

Presto includes two mechanisms for determining failure for cases that fall between pure tension and pure shear. In the first case, failure is governed by the equation

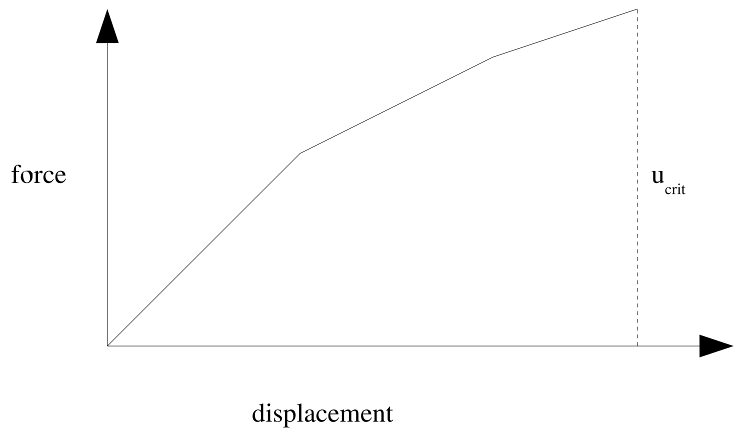


Figure 6.1: Force-displacement curve for spot-weld.

$$(u_n/u_{n_{crit}})^p + (u_t/u_{t_{crit}})^p < 1.0\,. \tag{6.1}$$

In Equation 6.1, the distance from the node to the original attachment point on the face as measured normal to the face is u_n, which is defined as the normal distance. The maximum value given for u_n in the normal force-displacement curve is $u_{n_{crit}}$. The distance from the node to the original attachment point on the face as measured along a tangent to the face is u_t, which is defined as the tangential distance. The maximum value given for u_t in the tangential force-displacement curve is $u_{t_{crit}}$. In Figure 6.1, the maximum value for the displacement is u_{crit}. The value p is a user-specified exponent that controls the shape of the failure surface.

Alternatively, Presto permits a user-specified function to determine the failure surface. The function defines the ratio of $u_t/u_{t_{crit}}$ at which failure will occur as a function of $u_n/u_{n_{crit}}$. The function must range from 0.0 to 1.0, and have a value of 1.0 at 0.0 and a value of 0.0 at 1.0. These restrictions preserve proper failure for the cases of pure tension and pure shear.

To use the spot-weld option in Presto, a `SPOT WELD` command block begins with the input line

```
BEGIN SPOT WELD
```

and is terminated with the input line

```
END [SPOT WELD] .
```

Within the command block, it is necessary to specify the set of nodes on one side of the spot-weld with the `NODE SET` command line. The `NODE SET` command line can list one or more node sets. Any node set listed on the `NODE SET` command line can be deleted from the list of node sets by using a `REMOVE NODE SET` command line. A set of element faces on an opposing side of the spot-weld (which we will refer to as the *first* surface) is specified with the `SURFACE` command line. The `SURFACE` command line can list one or more surfaces. Any surface listed on the `SURFACE` command line can be deleted from the list of surface by using a `REMOVE SURFACE` command line. For any node in the node set, the closest point to this node on the opposing surface should lie within the element faces specified by the `SURFACE` command line.

The normal force-displacement curve is specified by a function named by the value `function_nor_disp` in the `NORMAL DISPLACEMENT FUNCTION` command line. This function can be scaled by the real value `scale_nor_disp` in the `NORMAL DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0. The tangential force-displacement curve is specified by a function named by the string `function_tang_disp` in the `TANGENTIAL DISPLACEMENT FUNCTION` command line. This function can be scaled by the real value `scale_tang_disp` given in the `TANGENTIAL DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0.

The failure surface between pure tension and pure shear is controlled by specifying either the failure envelope exponent, p in Equation 6.1, or a failure function. The failure exponent is specified by the real value exponent in the `FAILURE ENVELOPE EXPONENT` command line. The failure function is specified by the `FAILURE FUNCTION` command line. If both a failure function and a failure exponent are given, then the failure function is used.

For an explicit, transient dynamics code like Presto, it is better to remove the force for the spot-weld over several load steps rather than over a single load step once the failure criterion is exceeded. The `FAILURE DECAY CYCLES` command line controls the number of load steps over which the final force is removed. To remove the final force at a spot-weld over five load increments, the integer specified by `number_decay_cycles` would be set to 5. Once the force at the spot-weld is reduced to zero, it remains zero for all subsequent time.

The spot-weld can take on area-based behavior by specifying a surface in place of a set of nodes. The identifier of this surface is specified by the string `surface_id` in the `SECOND SURFACE` command line. The area-based spot-weld creates a weld between all nodes on the second surface and the faces of the first surface. The load-resistance curve at each node is derived from the tributary area of the node times the given force-displacement curves. Thus, for the area-based spot-welds, the force-displacement curves give the force per unit area resisted by the weld. The user must set a tolerance for the node-to-face search with the

```
SEARCH TOLERANCE = <real>search_tolerance
```

command line. The value you select for `search_tolerance` will depend upon the distance between the nodes and surfaces used to define the spot-weld.

The `ACTIVE PERIODS` command line provides an additional option for the boundary condition. This command line is used to activate or deactivate the boundary condition for certain time periods. See Section 2.5 for more information about this command line.

6.7.5 Line Weld

```
BEGIN LINE WELD
  SURFACE = <string list> surface_names
  REMOVE SURFACE = <string list> surface_names
  BLOCK = <string list> block_names
  REMOVE BLOCK = <string list>block_names
  SEARCH TOLERANCE = <real>search_tolerance
  R DISPLACEMENT FUNCTION = <string>r_disp_function_name
  R DISPLACEMENT SCALE FACTOR = <real>r_disp_scale
  S DISPLACEMENT FUNCTION = <string>s_disp_function_name
  S DISPLACEMENT SCALE FACTOR = <real>s_disp_scale
  T DISPLACEMENT FUNCTION = <string>t_disp_function_name
  T DISPLACEMENT SCALE FACTOR = <real>t_disp_scale
  R ROTATION FUNCTION = <string>r_rotation_function_name
  R ROTATION SCALE FACTOR = <real>r_rotation_scale
  S ROTATION FUNCTION = <string>s_rotation_function_name
  S ROTATION SCALE FACTOR = <real>s_rotation_scale
  T ROTATION FUNCTION = <string>t_rotation_function_name
  T ROTATION SCALE FACTOR = <real>t_rotation_scale
  FAILURE ENVELOPE EXPONENT = <real>k
  FAILURE DECAY CYCLES = <integer>number_decay_cycles
  ACTIVE PERIODS = <string list>period_names
END LINE WELD
```

The line-weld capability is used to weld the edge of a shell to the face of another shell. The bond can transmit both translational and rotational forces. When failure of the line weld occurs, it breaks and no longer transmits any forces.

The edge of the shell that is tied to a surface is modeled with a block of one-dimensional elements (truss, beam, spring, etc.). The edge of the shell and the one-dimensional elements will share the same nodes. We will refer to the shell edge and the one-dimensional elements associated with it as the one-dimensional part of the line-weld model. The element blocks with the one-dimensional elements are specified by using the `BLOCK` command line. More than one element block can be listed on this command line. The element blocks referenced by the `BLOCK` command line must be one-dimensional elements—truss, beam, spring, etc.

The other part of the line weld is a set of faces defined by shell elements; this set of faces is the two-dimensional part of the line weld. The surface (the two-dimensional part of the model) to which the nodes (from the one-dimensional part of the model) are to be bonded is defined by any surface of element faces derived from shell elements. The line weld will bond each node in the element blocks listed in the `BLOCK` command line to the closest face (or faces) of element faces in the surfaces listed in the `SURFACE` command line. More than one surface can be listed on this command line.

The command line `SEARCH TOLERANCE` sets a tolerance on the search for node-to-face interactions. For a given node, only those faces within the distance set by the `search_tolerance` parameter will be searched to determine whether the node should be welded to the face.

Each section of the line weld has its own local coordinate system (r, s, t). The r-direction lies along a one-dimensional element (and hence on the surface). The s-direction lies on the surface and is tangential to the one-dimensional element. The t-direction lies normal to the face and is orthogonal to the r- and s-directions. Force-displacement functions and moment-rotation functions may be specified for all axes in the local coordinate system. If one of the functions is left out, the resistance is zero for that axis. These functions are similar to the ones used for the spot-weld (see Figure 6.1).

The force-displacement function in the r-direction represents shear resistance in the direction of the weld; this function is specified by a SIERRA function name on the `R DISPLACEMENT FUNCTION` command line. The force-displacement in the s-direction represents shear resistance tangential to the weld; this function is specified by a SIERRA function name on the `S DISPLACEMENT FUNCTION` command line. The force-displacement in the t-direction function represents tearing resistance normal to the surface; this function is specified by a SIERRA function name on the `T DISPLACEMENT FUNCTION` command line. The moment-rotation about the r-axis represents a rotational tearing resistance; this is specified by a SIERRA function name on the `R ROTATION FUNCTION` command line. The rotational resistances about the s-direction and the t-direction are likely not very meaningful, as rotations along these axes should be well constrained by the normal and tangential displacement relations. These two rotational resistances, if used, are defined with SIERRA function names on the `S ROTATION FUNCTION` and `T ROTATION FUNCTION` command lines. Note that each SIERRA function used in this command block is defined via a `DEFINITION FOR FUNCTION` command block in the domain scope.

Any of the above functions can be scaled by using a corresponding scale factor. For example, the force-displacement function on the `R DISPLACEMENT FUNCTION` command line can be scaled by the parameter `r_disp_scale` on the `R DISPLACEMENT SCALE FACTOR` command line. Only the force values of the force-displacement curve will be scaled.

The failure function for the line weld is similar to that for the spot-weld. Denote the displacement or rotation associated with a line weld as δ. Suppose that δ_i is a displacement in the r-direction. The force-displacement curve specified on the `R DISPLACEMENT FUNCTION` command line has a maximum value η. This is the maximum displacement the weld can endure in the r-direction before breaking. Associate this value of η with δ_i by designating it as η_i. Repeat this pairing process for all the displacements and rotations defining the line weld. Each displacement component in the line weld will be paired with one of the three maximum displacement values associated with the line weld. Each rotation component in the line weld will be paired with one of the three maximum rotation values associated with the line weld. Breaking of the weld under combined loading is calculated the same as the spot-weld. The weld breaks if

$$\sqrt[k]{\sum \left(\frac{\delta_i}{\eta_i}\right)^k} > 1\,. \tag{6.2}$$

In the above equation, the parameter k is set by the user. A typical value for k is 2. The summation takes place over all the failure functions (force-displacement and moment-rotation) for all the nodes. (The value for k is specified on the `FAILURE ENVELOPE EXPONENT` command line.)

For an explicit, transient dynamics code like Presto, it is better to remove the forces for the line weld over several time steps rather than over a single time step once the failure criterion is exceeded. The `FAILURE DECAY CYCLES` command line controls the number of time steps over which the final force is removed. To remove the final force at a line weld over five time steps, the integer specified by `number_decay_cycles` would be set to 5. Once the force in the line weld is reduced to zero, it remains zero for all subsequent time.

The `ACTIVE PERIODS` command line provides an additional option for the boundary condition. This command line is used to activate or deactivate the boundary condition for certain time periods. See Section 2.5 for more information about this command line.

6.7.6 Viscous Damping

```
BEGIN VISCOUS DAMPING <string>damp_name
  # {block set commands}
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  MASS DAMPING COEFFICIENT = <real>mass_damping
  STIFFNESS DAMPING COEFFICIENT = <real>stiff_damping
  #
  # additional command
  ACTIVE PERIODS = <string list>period names
END [VISCOUS DAMPING <string>damp_name]
```

The `VISCOUS DAMPING` command block adds simple Rayleigh viscous damping to mesh nodes. At each node, Presto computes a damping coefficient, which is then multiplied by the node velocity to create a damping force. The damping coefficient is the sum of the mass times a mass damping coefficient and the nodal stiffness times a stiffness damping coefficient. In general, the mass damping portion damps out low-frequency modes in the mesh, while the stiffness damping portion damps out higher-frequency terms. Appropriate values for the damping coefficients depend on the frequencies of interest in the mesh. The general expression for the critical damping fraction, c_d, for a given frequency is

$$c_d = (k_d * \omega + m_d/\omega)/2\,, \tag{6.3}$$

where k_d is the stiffness damping coefficient, m_d is the mass damping coefficient, and ω is the frequency of interest. The stiffness damping portion must be used with caution. Because this term depends on the stiffness, it can affect the critical time step. Thus certain ranges of values for the stiffness damping coefficient can change the critical time step for the mesh. As Presto does not currently modify the critical time step based on the selected values for this coefficient, some choices for this parameter can cause solution instability.

6.7.6.1 Block Set Commands

The {block set commands} portion of the `VISCOUS DAMPING` defines a set of element blocks associated with the viscous damping and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of element blocks. See Section 6.1 for more information about the use of these command lines for creating a set of element blocks used by viscous damping. There must be at least one `BLOCK` or `INCLUDE ALL BLOCKS` command line in the command block.

All the nodes associated with the elements specified by the block set commands will have viscous damping forces applied.

6.7.6.2 Viscous Damping Coefficient

The mass damping coefficient in Equation 6.3, m_d, is specified as the parameter `mass_damping` on the command line

```
MASS DAMPING COEFFICIENT = <real>mass_damping .
```

Mass damping most strongly damps the low-frequency modes.

The stiffness damping coefficient command line in Equation 6.3, k_d, is specified as the parameter `stiff_damping` on the command line

```
STIFFNESS DAMPING COEFFICIENT = <real>stiff_damping .
```

Stiffness damping most strongly damps high-frequency modes. Large values for the stiffness damping coefficient can affect the critical time step. Since Presto does not modify the critical time step based on the stiffness damping coefficient, it may be necessary for the user to manually decrease the critical time step.

6.7.6.3 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `VISCOUS DAMPING` command block:

```
ACTIVE PERIODS = <string list>period_names
```

This command line can activate or deactivate the viscous damping for certain time periods. See Section 2.5 for more information about this command line.

6.8 References

1. Brown, K. H., J. R. Koteras, D. B. Longcope, and T. L. Warren. *CavityExpansion: A Library for Cavity Expansion Algorithms, Version 1.0*, in review. Albuquerque, NM: Sandia National Laboratories, 2003.

2. Lysmer, J., and R. L. Kuhlmeyer. "Finite Dynamic Model for Infinite Media." *Journal of the Engineering Mechanics Division, Proceedings of the American Society of Civil Engineers* (August 1979): 859–877.

3. Cook, R. D., Malkus, D. S., and Plesha, M. E. *Concepts and Applications of Finite Element Analysis, Third Edition.* New York: John Wiley and Sons, 1989.

Intentionally Left Blank

Chapter 7

Contact

This chapter describes the input syntax for defining interactions of contact surfaces in a Presto analysis. For more information on contact and its computational details, consult References 1 and 2.

Contact refers to the interaction of one or more bodies when they physically touch. This can include the interaction of one part of a surface against another part of the same surface, the surface of one body against the surface of another body, and so forth. The contact algorithms within Presto are designed to ensure that surfaces do not interpenetrate in a nonphysical way, and that the interface behavior is computed correctly according to any user-specified surface-physics models (e.g., energy dissipation from a friction model). Presto uses a kinematic approach rather than a penalty approach to eliminate the interpenetration of surfaces. In the kinematic approach, a series of constraint equations are satisfied that remove interpenetration. A penalty approach can be thought of as introducing "stiff" springs between contact surfaces as a means of preventing interpenetration.

In the current version of Presto, contact between surfaces is computed as node-face interactions. To establish some key definitions for node-face contact and node-face interactions, we consider the simple two-dimensional contact problem shown in Figure 7.1. There are two blocks, a and b. Block a is enclosed by surface a, and block b is enclosed by surface b. In finite element models, a surface is defined by a collection of finite element faces. The surface of a block of hexahedral elements, for example, is defined by a collection of quadrilateral faces on the surface of the block. For our two-dimensional example, the faces are a straight line between two nodes. We only show the faces on the portions of the surfaces that will come into contact.

Figure 7.1 shows the two blocks at time step n. Figure 7.2 shows the two blocks at time step $n+1$. The blocks have moved and deformed under the influence of external forces. Contact has not been taken into account, and we now observe interpenetration of the two blocks. We remove this interpenetration by applying our contact algorithm.

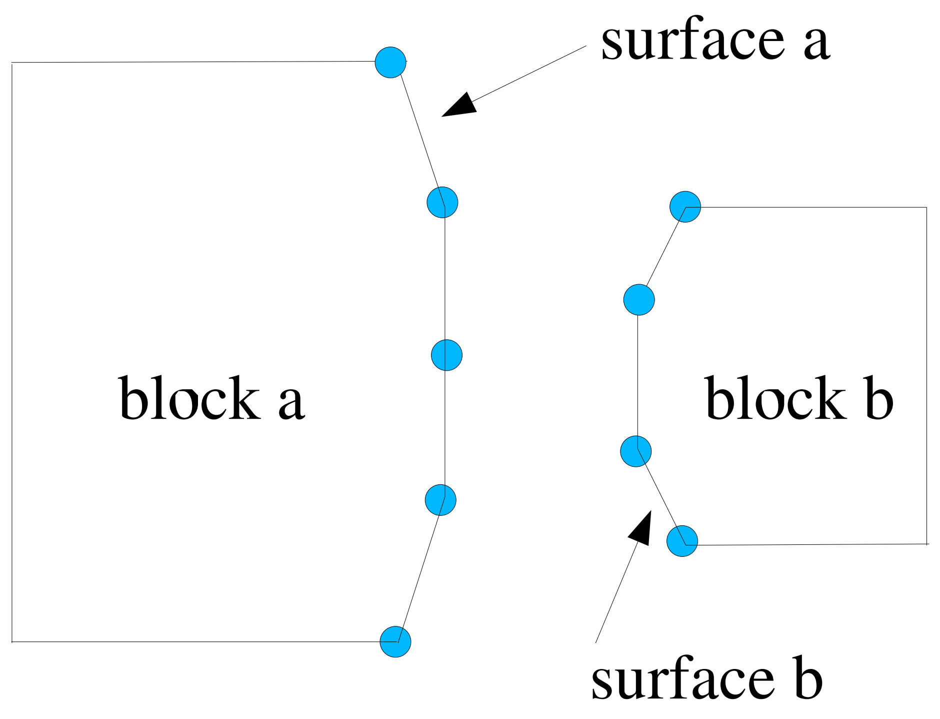


Figure 7.1: Two blocks at time step n before contact.

For interpenetration to occur as shown in Figure 7.2, any node on surface a that interpenetrates surface b must pass through some face on surface b. Likewise, each node on surface b that interpenetrates surface a must pass through some face on surface a. We could push all the nodes on surface a so that they lie on surface b, where surface b has the configuration shown in Figure 7.2. Or we could push all the nodes on surface b so that they lie on surface a, where surface a has the configuration shown in Figure 7.2. In some cases, we do use one of these two options, a to b or b to a. However, we typically do something “in between” and move nodes to what can be described as an interface surface, which is shown by a thick black line in Figure 7.2. The interface surface is shown as a straight line, but it would really be a curved line in all but the most unusual cases for a two-dimensional problem like the one shown. For three-dimensional problems, the interface surface will be a complex surface in three-dimensional space.

For our “in between” solution, each node on surface a that has penetrated some face on surface b, we compute some set of forces (based on the amount of node penetration) on the node on a and the nodes associated with the face on b to remove “some part” of the interpenetration. Likewise, for each node on surface b that has penetrated some face on surface a, we can compute some set of forces (based on

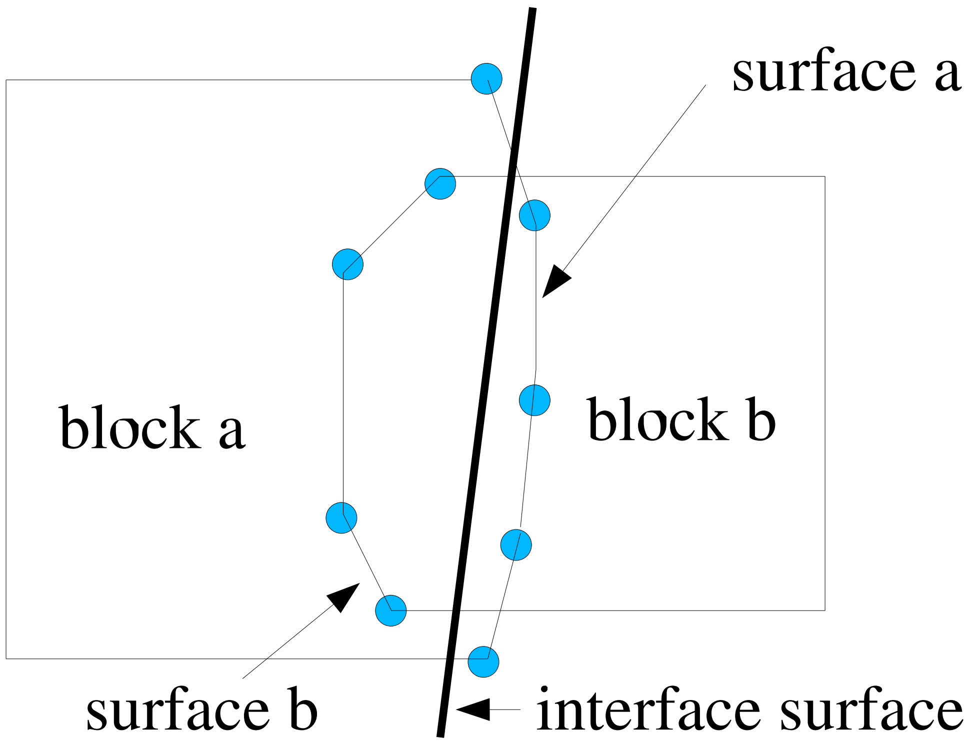


Figure 7.2: Two blocks at time step $n + 1$, after penetration.

the amount of node penetration) on surface b and the nodes associated with the face on surface a to remove "another part" of the interpenetration. This node-face interaction from both contact surfaces is typically what is encountered in Presto, and it is referred to as "symmetric" contact. After the nodes on both surfaces, a and b, have been moved, we have defined an interface surface. A more detailed discussion of how we move the nodes on both surfaces is given in those sections related to kinematic partitioning, Section 7.14.4 and Section 7.15.2.

The simple two-dimensional example we have just discussed is analogous to much of the contact that is encountered when contact in Presto is used in an analysis. Surfaces are generated that consist of a collection of faces, each face being defined by a nodal connectivity. Node-face interactions from both contact surfaces (symmetric contact) are used to move nodes to account for any interpenetration of the two surfaces. Interpenetration means we have a node on a surface that has "moved through" a face on an opposing surface.

Contact in Presto will handle the node-face contact just presented. It will also handle variations of the node-face contact we have just discussed. Some of these variations are as follows:

- In some cases, you may want one surface of a surface pair to determine the interface surface. One surface will be designated as the master surface. The opposing surface will be designated as the slave surface. The nodes on the slave surface will be moved to the master surface. The master surface sets the interface surface. This arrangement would be a pure master-slave situation. You might want to use this arrangement if you had a very stiff surface like steel contacting a very weak surface like foam.

- In some cases, you may want one surface of a surface pair to be more influential in determining the interface surface than its opposing surface. This arrangement is done by "weighting" the more influential surface and involves a concept called kinematic partitioning. The above case of pure master-slave represents the limiting case for kinematic partitioning.

- A special case of contact called "tied contact" allows you to tie two surfaces on different objects together. The two surfaces that are tied together share a coincident surface or are in very close proximity at time 0.0. The initial point of contact between a tied node and an opposing face at time 0.0 is maintained for all times. At each time step, the node is moved so that it as the same point on the face regardless of where the faces move or how the face deforms.

- One of the surfaces in a contact pair can be an analytic surface. An analytic surface is defined by an algebraic expression, not by a collection of faces derived from elements. The algebraic expression that defines the surface of a cylinder is an example of an analytic surface. The nodes on the opposing surface cannot penetrate the analytic surface.

- Instead of having two surfaces in contact, you can have a set of nodes not associated with faces that contacts a surface. We refer to this set of nodes as a "contact node set." The nodes in the contact node set can contact a surface that is a collection of faces (the usual surface definition) or an analytic surface. The nodes in the contact node set cannot penetrate the surface.

- A mesh could have an initial interpenetration of two surfaces due to the meshing process. We refer to this situation as "initial overlap." You have the option of removing this initial overlap.

- An element block can contact itself. A block of elements may deform to such an extent that a part of the surface of the block comes into contact with another part of the surface of the block. This is referred to as "self-contact." For self-contact, a node that is part of an element block can contact a face that is exterior to the same element block.

There are some special considerations for contact with structural elements (shells, springs, trusses, beams) with the current implementation of contact. A shell element has both a top face and a bottom face that are defined by the same geometric entity. One-dimensional elements (springs, trusses, and beams) have no faces.

Shell elements are handled by the contact algorithm, but they are much more difficult to handle than solid elements. Determining whether a node has penetrated a shell element is more difficult than determining whether a node has penetrated a solid. For a solid element with an external face, there is only one normal for the face. For a shell element, there are two faces—one on each side of the geometric entity that defines the shell. Each face has a normal, and the two normals for the shell element point in opposite directions. For shell elements, two faces are constructed for the element within the contact algorithm. The faces, each with a unique outward normal, can be coincident, or they can be separated by the thickness of the shell. Separating the two shell faces that are originally coincident at the geometric plane of the shell by the thickness of the shell is referred to as "lofting." To implement lofting, we need information about the thickness of the shell. This information is specified in the `SHELL SECTION` command block described in Section 5.2.2. For more information on lofting, see Section 7.9.

Contact for shell elements is only considered on shell faces; shell edges are currently not considered. The contact of a shell edge with another shell edge is not detected, and the contact of a shell edge with a continuum element edge is not detected. A shell element can coincide with the face of a continuum element. The contact algorithm will properly account for this situation. Two shell elements can also overlay each other, i.e., share the same set of nodes. The contact algorithm will also properly account for this situation. For a block of shell elements, two surfaces are created in contact.

Contact for one-dimensional elements (springs, trusses, beams) is currently implemented only for one-dimensional elements contacting a surface. The contact algorithm will not detect contact of a one-dimensional element with the edge of a continuum element, with the edge of a shell element, or with another one-dimensional element. Contact of one-dimensional elements is discussed in Section 7.2.4.

Contact in Presto is implemented in two distinct phases: a search algorithm and an enforcement algorithm. The search algorithm identifies nodes that have penetrated a face, while the enforcement algorithm computes the forces to remove penetration and the forces that observe the user-specified surface physics. The contact search within Presto focuses on large-scale global contact in a massively parallel environment. This processing step can be quite expensive, taking upwards of 60% of the analysis time, especially on multiprocessor analyses. The search algorithm relies on normal and tangential tolerances to describe a region around each face within which any nodes found are identified as potential interactions. The size of these tolerances is problem dependent.

The enforcement algorithm is based on a kinematic approach that satisfies momentum balance by default or, optionally, a penalty approach. A kinematic approach with momentum balance enforcement, where iterations are used to ensure normal impact momentum balance and frictional response, is always more accurate than the penalty approach. Consequently, when the surface interaction involves a frictional response, the kinematic approach with momentum balance is recommended. Currently, the penalty approach is under development. Use the kinematic approach with momentum balance (the default) until the penalty approach has been fully developed and tested.

A number of friction models are available to describe the surface interactions. In this chapter on contact, we will use the term *friction model* for what is really a surface-physics model.

Contact within a Presto analysis is defined within a `CONTACT DEFINITION` command block. Within the contact definition scope, there are command lines and command blocks that define the specifics for the interaction of surfaces via the contact algorithm. Some of the command lines and command blocks within the contact scope set up default parameters that affect all contact calculations. Some of the command blocks in the contact scope affect only the interaction between a pair of surfaces.

There are three approaches that can be used to define a contact problem:

1. Accept all the Presto default parameters for a problem.
2. Accept the Presto default parameters for some of the contact surfaces. For the rest of the contact surfaces, the user can change some of the Presto default settings.
3. Define all surface-pair interactions separately.

Note that the speed of contact is based primarily on the number of nodes and faces in the contact surfaces and, to a much lesser extent, on the number of interactions specified. Consequently, choosing the third approach above is not likely to reduce the run time significantly.

The general pattern of syntax for describing contact is as follows:

- Identify all surfaces that need to be considered for contact. This is done with command lines (or command blocks) within the contact scope.
- Specify any analytic surface used for contact. Analytic surfaces are described with a command block.
- Specify any special contact options such as initial overlap removal or angle for multiple interactions. This is done with command lines within the contact scope.

- Describe friction models used in the surface interactions for this analysis. Currently, there are 11 types of primary friction models. The Shared Interface MODels (SIMOD) and user subroutines can also be used as friction models. A friction model is described with a command block.
- Set contact search options that will serve as defaults for all the surface interactions. These values are set in the `SEARCH OPTIONS` command block.
- Set contact enforcement options that will apply to all the surface interactions. These values are set in the `ENFORCEMENT OPTIONS` command block.
- Set default interaction values that apply to all the surface interactions. These values are set in the `INTERACTION DEFAULTS` command block.
- Specify values for interactions between specific contact surfaces. This is done within an `INTERACTION` command block. Values specified in this command block override the defaults for the particular pair of surface interactions.

7.1 Contact Definition Block

All commands for contact occur within a `CONTACT DEFINITION` command block. A summary of these commands follows.

```
BEGIN CONTACT DEFINITION <string>name
  #
  # contact surface and node set definition
  CONTACT SURFACE <string>name
    CONTAINS <string list>surface_names
  #
  SKIN ALL BLOCKS = <string>ON|OFF(OFF)
    [EXCEPT <string list> block_names]
  #
  BEGIN CONTACT SURFACE <string>name
    BLOCK = <string list>block_names
    SURFACE = <string list>surface_names
    NODE SET = <string list>node_set_names
    REMOVE BLOCK = <string list>block_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE NODE SET = <string list>nodelist_names
  END [CONTACT SURFACE <string>name]
  #
  CONTACT NODE SET <string>surface_name
    CONTAINS <string>nodelist_names
  #
  # analytic surfaces
  BEGIN ANALYTIC PLANE <string>name
    NORMAL = <string>defined_direction
    POINT = <string>defined_point
  END [ANALYTIC PLANE <string>name]
  #
  BEGIN ANALYTIC CYLINDER <string>name
    CENTER = <string>defined_point
    AXIAL DIRECTION = <string>defined_axis
    RADIUS = <real>cylinder_radius
    LENGTH = <real>cylinder_length
    CONTACT NORMAL = <string>OUTSIDE|INSIDE
  END [ANALYTIC CYLINDER <string>name]
  #
  BEGIN ANALYTIC SPHERE <string>name
    CENTER = <string>defined_point
    RADIUS = <real>sphere_radius
  END [ANALYTIC SPHERE <string>name]
  # end contact surface and node set definition
```

```
#
UPDATE ALL SURFACES FOR ELEMENT DEATH = <string>ON|OFF(ON)
#
BEGIN REMOVE INITIAL OVERLAP
  OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
  OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
  SHELL OVERLAP ITERATIONS = <integer>max_iter(10)
  SHELL OVERLAP TOLERANCE = <real>shell_over_tol(0.0)
END [REMOVE INITIAL OVERLAP]
#
MULTIPLE INTERACTIONS = <string>ON|OFF(ON)
MULTIPLE INTERACTIONS WITH ANGLE = <real>angle(60.0)
#
SURFACE NORMAL SMOOTHING = <string>ON|OFF(OFF)
#
ERODED FACE TREATMENT = <string>NONE|ALL(ALL)
#
# shell lofting
BEGIN SHELL LOFTING
  LOFTING ALGORITHM = <string>ON|OFF(ON)
  COINCIDENT SHELL TREATMENT = <string>DISALLOW|IGNORE|
    SIMPLE(DISALLOW)
  COINCIDENT SHELL HEX TREATMENT = <string>DISALLOW|
    IGNORE|TAPERED|EMBEDDED(DISALLOW)
END [SHELL LOFTING]
# end shell lofting
#
CONTACT VARIABLES = <string>ON|OFF(OFF)
#
# surface-physics models
BEGIN FRICTIONLESS MODEL <string>name
END [FRICTIONLESS MODEL <string>name]
#
BEGIN CONSTANT FRICTION MODEL <string>name
  FRICTION COEFFICIENT = <real>coeff
END [CONSTANT FRICTION MODEL <string>name]
#
BEGIN TIED MODEL <string>name
END [TIED MODEL <string>name]
#
BEGIN SPRING WELD MODEL <string>name
  NORMAL DISPLACEMENT FUNCTION = <string>func_name
  NORMAL DISPLACEMENT SCALE FACTOR =
    <real>scale_factor(1.0)
  TANGENTIAL DISPLACEMENT FUNCTION = <string>func_name
```

```
    TANGENTIAL DISPLACEMENT SCALE FACTOR =
      <real>scale_factor(1.0)
    FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
    FAILURE DECAY CYCLES = <integer>num_cycles(1)
    FAILED MODEL = <string>failed_model_name|FRICTIONLESS
      (FRICTIONLESS)
  END [SPRING WELD MODEL <string>name]
  #
  BEGIN SURFACE WELD MODEL <string>name
    NORMAL CAPACITY = <real>normal_cap
    TANGENTIAL CAPACITY = <real>tangential_cap
    FAILURE DECAY CYCLES = <integer>num_cycles(1)
    FAILED MODEL = <string>failed_model_name|FRICTIONLESS
      (FRICTIONLESS)
  END [SURFACE WELD MODEL <string>name]
  #
  BEGIN AREA WELD MODEL <string>name
    NORMAL CAPACITY = <real>normal_cap
    TANGENTIAL CAPACITY = <real>tangential_cap
    FAILURE DECAY CYCLES = <integer>num_cycles(1)
    FAILED MODEL = <string>failed_model_name|FRICTIONLESS
      (FRICTIONLESS)
  END [AREA WELD MODEL <string>name]
  #
  BEGIN ADHESION MODEL <string>name
    ADHESION FUNCTION = <string>func_name
    ADHESION SCALE FACTOR = <real>scale_factor(1.0)
  END [ADHESION MODEL <string>name]
  #
  BEGIN COHESIVE ZONE MODEL <string>name
    TRACTION DISPLACEMENT FUNCTION = <string>func_name
    TRACTION DISPLACEMENT SCALE FACTOR =
      <real>scale_factor(1.0)
    CRITICAL NORMAL GAP = <real>crit_norm_gap
    CRITICAL TANGENTIAL GAP = <real>crit_tangential_gap
  END [COHESIVE ZONE MODEL <string>name]
  #
  BEGIN JUNCTION MODEL <string>name
    NORMAL TRACTION FUNCTION = <string>func_name
    NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
    TANGENTIAL TRACTION FUNCTION = <string>func_name
    TANGENTIAL TRACTION SCALE FACTOR =
      <real>scale_factor(1.0)
    NORMAL CUTOFF DISTANCE FOR TANGENTIAL TRACTION =
      <real>distance
```

```
END [JUNCTION MODEL <string>name]
#
BEGIN THREADED MODEL <string>name
  NORMAL TRACTION FUNCTION = <string>func_name
  NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
  TANGENTIAL TRACTION FUNCTION = <string>func_name
  TANGENTIAL TRACTION SCALE FACTOR =
    <real>scale_factor(1.0)
  TANGENTIAL TRACTION GAP FUNCTION = <string>func_name
  TANGENTIAL TRACTION GAP SCALE FACTOR =
    <real>scale_factor(1.0)
  NORMAL CAPACITY = <real>normal_cap
  TANGENTIAL CAPACITY = <real>tangential_cap
  FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
  FAILURE DECAY CYCLES = <integer>num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
END [THREADED MODEL <string>name]
#
BEGIN PV_DEPENDENT MODEL <string>name
  STATIC COEFFICIENT = <real>stat_coeff
  DYNAMIC COEFFICIENT = <real>dyn_coeff
  VELOCITY DECAY = <real>vel_decay
  REFERENCE PRESSURE = <real>p_ref
  OFFSET PRESSURE = <real>p_off
  PRESSURE EXPONENT = <real>p_exp
END [PV_DEPENDENT MODEL <string>name]
#
BEGIN SIMOD SHARED MODEL <string>name
  USE SIMOD MODEL = <string>name
  FAILED MODEL = <string>name|FRICTIONLESS
    (FRICTIONLESS)
END [SIMOD SHARED MODEL <string>name]
#
BEGIN SIMOD UNKNOWN MODEL <string>name
  USE SIMOD MODEL = <string>name
  FAILED MODEL = <string>name|FRICTIONLESS
    (FRICTIONLESS)
END [SIMOD UNKNOWN MODEL <string>name]
# end surface physics models
#
BEGIN USER SUBROUTINE MODEL <string>name
  INITIALIZE MODEL SUBROUTINE = <string>init_model_name
  INITIALIZE TIME STEP SUBROUTINE = <string>init_ts_name
  INITIALIZE NODE STATE DATA SUBROUTINE =
    <string>init_node_data_name
  LIMIT FORCE SUBROUTINE = <string>limit_force_name
  ACTIVE SUBROUTINE = <string>active_name
  INTERACTION TYPE SUBROUTINE = <string>interaction_name
END [USER SUBROUTINE MODEL <string>name]
#
# search options command block
BEGIN SEARCH OPTIONS [<string>name]
  GLOBAL SEARCH INCREMENT = <integer>num_steps(1)
  GLOBAL SEARCH ONCE = <string>ON|OFF(OFF)
  SEARCH TOLERANCE = <string>AUTOMATIC|USER_DEFINED
    (AUTOMATIC)
  NORMAL TOLERANCE = <real>norm_tol
  TANGENTIAL TOLERANCE = <real>tang_tol
  SECONDARY DECOMPOSITION = <string>ON|OFF(ON)
END [SEARCH OPTIONS <string>name]
#
# enforcement
BEGIN ENFORCEMENT OPTIONS [<string>name]
  ENFORCEMENT ALGORITHM = <string>MOMENTUM_BALANCE|
    PENALTY(MOMENTUM_BALANCE)
  MOMENTUM BALANCE ITERATIONS = <integer>num_iter(5)
END [ENFORCEMENT OPTIONS <string>name]
#
BEGIN INTERACTION DEFAULTS [<string>name]
  SURFACES = <string list>surface_names
  SELF CONTACT = <string>ON|OFF(OFF)
  GENERAL CONTACT = <string>ON|OFF(OFF)
  AUTOMATIC KINEMATIC PARTITION = <string>ON|OFF(OFF)
  INTERACTION BEHAVIOR = <string>SLIDING|
    INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
  FRICTION MODEL = <string>friction_model_name|
    FRICTIONLESS(FRICTIONLESS)
END [INTERACTION DEFAULTS <string>name]
#
BEGIN INTERACTION [<string>name]
  SURFACES = <string>surface1 <string>surface2
  MASTER = <string>surface
  SLAVE = <string>surface
  KINEMATIC PARTITION = <real>kin_part
  NORMAL TOLERANCE = <real>norm_tol
  TANGENTIAL TOLERANCE = <real>tang_tol
  OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
  OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
  FRICTION MODEL = <string>friction_model_name|
        FRICTIONLESS(FRICTIONLESS)
      AUTOMATIC KINEMATIC PARTITION
      INTERACTION BEHAVIOR = <string>SLIDING|
        INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
    END [INTERACTION <string>name]
    # end enforcement
    #
  END [CONTACT DEFINITION <string>name]
```

The command block begins with the input line

```
BEGIN CONTACT DEFINITION <string>name
```

and is terminated with the input line

```
END [CONTACT DEFINITION <string>name] ,
```

where `name` is a name for this contact definition. The name should be unique among all the contact definitions in an analysis. All other contact commands are encapsulated within this command block, as shown in the summary of the block presented previously. These other contact commands are described in Section 7.2 through Section 7.15. Section 7.16 explains how to implement contact for several example problems.

A typical analysis will have only one `CONTACT DEFINITION` command block. However, more than one contact definition can be used. As each `CONTACT DEFINITION` command block creates its own contact entity, fewer of these command blocks provide more efficient contact processing.

7.2 Descriptions of Contact Surfaces

In general, contact determines whether one surface has interpenetrated another surface. As indicated previously, a surface is defined by an analytic representation or a collection of finite element faces. This section describes how to define a surface composed of finite element faces. It also describes how to define a set of nodes (zero-dimensional entities) not associated with faces that can contact a surface, the surface being composed of finite element faces or the surface being an analytic surface. This latter case (a node not associated with a face contacting a surface) is useful for models where, for example, we have both continuum and SPH elements. (We will refer to nodes not associated with faces as “unassociated” nodes.) Defining rigid analytic surfaces is discussed in Section 7.3.

Generally, a surface is defined as a collection of finite element faces. Both continuum elements and shell elements have faces. For a continuum element, any face that is not shared with another element can be considered for contact. For a shell element, one element can have both a top face and a bottom face. These top and bottom surfaces are automatically created for the contact algorithm and may be lofted by a user-specified thickness. Shell contact is done by computing the contact forces on the top and bottom surfaces of the shells and then moving the resulting forces back to the original shell nodes.

At this point, it is important to introduce the concept of “skinning” a block of elements. We can generate a surface (a collection of faces) from a block of continuum elements by skinning the block of elements. All exterior faces (any face not shared by two elements) will be associated with the surface for that block when the block is skinned. If we have two blocks of continuum elements that are connected (some of the element faces in one block are shared by the element faces in the other block) and we skin both of these blocks, then the skinned surface for each block will consist of faces that are exterior to both blocks. For this case, we will have generated two surfaces. The set of external faces from skinning the first block will have a unique surface name, and the set of external faces generated by skinning the second block will have a unique surface name. Any face shared by the two blocks will not be in the surfaces derived by skinning the two blocks. If we have a single block of shell elements and we skin the block, then all the top faces of all the shell elements will be one surface and the bottom faces of all the shell elements will be another surface. (In Presto, we do not have to be concerned with naming two distinct surfaces for shell elements. This is handled internally by the code.) Suppose we have two blocks of shell elements in which none of the elements in one block overlap the elements in another block; the two blocks are joined only at the shell edges. In this case, we will get a unique surface identifier that references both the top and bottom faces of all the shell elements in the first block, and we will get a unique surface identifier that references both the top and bottom faces of all the shell elements in the second block.

Skinning becomes more complicated when we have a shell surface that overlays the surface of a block of continuum elements. If a shell surface overlays the surface of a block of continuum elements, we can have shell elements that are coincident with the external faces of the continuum elements. Coincident in this case means that a shell element has the same nodal connectivity as the nodal connectivity defining an external face of a continuum element. Skinning also becomes more complicated when we have two shell blocks with coincident elements. Coincident in this case means we have two shell elements in different blocks that have the same nodal connectivity. See Section 7.9 for more information about skinning with shells.

A face can only be associated with a single contact surface. However, in the process of defining contact surfaces, you might create a situation where one face appears on more than one contact surface. If a face appears on more than one contact surface, an ambiguous situation arises. The following example should help to explain this ambiguous situation.

First, let us establish a situation where there is a face that appears on two contact surfaces. One contact surface is defined by skinning an element block of hexahedral elements. The name of this surface obtained by skinning a block is `block_1024`. One of the faces in surface `block_1024` is defined by the node connectivity {100, 101, 1002, 1001}. A surface on the same element block is defined by specifying a side set definition. The name of this surface obtained by using a side set definition is `surface_1000`. One of the faces in surface `surface_1000` is defined by the node connectivity {100, 101, 1002, 1001}. In this example, we have the same face, {100, 101, 1002, 1001}, defined on two different surfaces, `block_1024` and `surface_1000`. The definition of the surface with the side set includes one of the faces in the surface obtained by skinning the block.

Now let us show how an ambiguous situation can arise. Suppose, in our example, that the friction model specified for surface `block_1024` is different from the friction model specified for surface `surface_1000`. Furthermore, suppose that the tolerances specified for surface `block_1024` are different from the tolerances specified for surface `surface_1000`. For contact, only one friction model and only one set of tolerances can be applied to face {100, 101, 1002, 1001}. The question arises as to which friction model and which set of tolerances should be applied to the face. Any face, in general, can have only one type of a given contact property—friction model, tolerances, etc.,—applied to the face.

To handle the case of a face defined in more than one contact surface, any face defined in more than one contact surface will be assigned to the first contact surface defined in the `CONTACT DEFINITION` command block that includes the face. For example, if contact surface `block_1024` is defined before contact surface `surface_1000` in the `CONTACT DEFINITION` command block, then face {100, 101, 1002, 1001} will be assigned to contact surface `block_1024`. Face {100, 101, 1002, 1001} will not be assigned to contact surface `surface_1000`. In general, then, any face appearing on multiple contact surfaces will be assigned to the first contact surface defined in the `CONTACT DEFINITION` command block that includes the face. (The ordering of the contact surface definitions in the `CONTACT DEFINITION` command block will determine how faces defined on multiple contact surfaces are assigned to a contact surface.) If a face is defined on multiple contact surfaces, a warning is generated.

For the case where unassociated nodes are contacting a surface, you will need to define some collection of unassociated nodes and a surface that can be contacted by these nodes. SPH particles contacting a surface is an example of nodes contacting a surface. The contact of one-dimensional elements (springs, trusses, beams) with a surface can also be modeled as unassociated nodes contacting a surface, although, as in the case of shells, there are some limitations. The contact algorithm cannot detect a one-dimensional element cutting through the edge of a shell element, the edge of a continuum element, or through another one-dimensional element.

To describe surfaces defined by finite element faces that can be considered for contact, you can use the `CONTACT SURFACE` command line, the `SKIN ALL BLOCKS` command line, or the `CONTACT SURFACE` command block. To describe unassociated nodes that can come into contact with surfaces, you should use the `CONTACT NODE SET` command line or the `CONTACT SURFACE` command block. A `CONTACT DEFINITION` command block can contain any combination of these command lines and command blocks provided that no two of these commands have the same name. The `CONTACT DEFINITION` command block MUST include some type of surface definition. Any element faces or unassociated nodes that you want to use for contact interaction must be identified as contact faces or contact nodes, respectively.

Section 7.2.1 through Section 7.2.4 describe the command lines and command blocks for defining contact surfaces composed of finite element faces and node sets that can contact surfaces.

7.2.1 Contact Surface Command Line

```
CONTACT SURFACE <string>name
  CONTAINS <string list>surface_names
```

This command line identifies a set of surfaces (specified as side sets) and element blocks that will be considered as a single contact surface; the string `name` is the unique name for this contact surface. The list denoted by `surfaces_names` is a list of strings identifying surfaces that are to be associated with this contact surface `name`. The surfaces can be side sets, element blocks, or any combination of the two. Any specified element blocks are “skinned,” i.e., a surface is created from the exterior of the element block. See the previous discussion on skinning. Blocks of shell elements will be skinned, and the shell surfaces generated from a `CONTACT SURFACE` command line will be lofted for contact if the lofting algorithm is `ON` in the `SHELL LOFTING` command block.

If a block of one-dimensional elements (springs, trusses, beams) is included in the list of `surface_names`, the element block will be ignored. Thus, to include the one-dimensional elements for contact, a `CONTACT NODE SET` command line should be used. See Section 7.2.4.

The `name` you create for a surface can be referenced in command blocks that specify how that surface will interact with another contact surface or with itself. See Section 7.14.1 and Section 7.15.1.

The surfaces can contain a heterogeneous set of face types as well as any number of side sets and element blocks.

If a face appears in a side set and also in a set of faces generated by the skinning of an element block, that face will produce an error. As indicated previously, any given face may not appear in more than one contact surface.

7.2.2 Skin All Blocks

```
SKIN ALL BLOCKS = <string>ON|OFF(OFF)
  [EXCEPT <string list>block_names]
```

You may wish to consider contact between the external surfaces of all the element blocks in the mesh. The `SKIN ALL BLOCKS` command line causes all element blocks to be “skinned,” i.e., a surface is created from the exterior of each element block. The skinned surfaces are then given contact surface names identical to the name of the element block. For instance, if a mesh contained the element blocks `block_1`, `block_10`, and `block_11`, then `SKIN ALL BLOCKS` would create three contact surfaces from these blocks with the names `block_1`, `block_10`, and `block_11`, respectively.

You can selectively delete some blocks from skinning by using the `EXCEPT` option. Any blocks you do not want to be skinned will be included in a list of block names following `EXCEPT`.

The `SKIN ALL BLOCKS` is useful for large models in which the individual specification of contact surfaces would be unwieldy.

If the `SKIN ALL BLOCKS` command line is used without the `EXCEPT` option, contact surfaces cannot be defined by the above `CONTACT SURFACE` command line or the `CONTACT SURFACE` command block. The use of the `SKIN ALL BLOCKS` command line without the `EXCEPT` option would include all exterior faces for all element blocks in the set of contact surfaces generated by the SKIN ALL BLOCKS command line. The added use of a CONTACT SURFACE command line or CONTACT SURFACE command block would then generate a new surface that would have to include at least one exterior face. But all exterior faces have been included in the surfaces generated by the SKIN ALL BLOCKS command line (without the EXCEPT option). This creates a situation where we have the same face in two different surfaces. Specifying the same face in two different contact surfaces is not allowed. See the example discussed in the introductory part of Section 7.2.

If you use the EXCEPT option, you can use a CONTACT SURFACE command line or the CONTACT SURFACE command block as long as you do not reference the same face on different surfaces when defining the various contact surfaces.

The CONTACT SURFACE command block, if it is used to defined a set of unassociated nodes for a contact node set, and the CONTACT NODE SET command line can be used with the SKIN ALL BLOCKS command line regardless of whether or not it uses the EXCEPT option.

If the mesh includes blocks of shell elements, the shell surfaces generated from a SKIN ALL BLOCKS command line will be lofted for contact according to the lofting algorithm specified in the SHELL LOFTING command block.

If the mesh includes blocks of one-dimensional elements (beams, trusses), the element blocks with one-dimensional elements are ignored in contact. Thus, to include the one-dimensional elements for contact, a CONTACT NODE SET command line should be used. See Section 7.2.4.

7.2.3 Contact Surface Command Block

```
BEGIN CONTACT SURFACE <string>name
  BLOCK = <string list>block_names
  SURFACE = <string list>surface_names
  NODE SET = <string list>node_set_names
  REMOVE BLOCK = <string list>block_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE NODE SET = <string list>node_set_names
END [CONTACT SURFACE <string>name]
```

The CONTACT SURFACE command block can be used to define a contact surface consisting of a collection of finite element faces or a set of unassociated nodes that will be a contact node set. We can use some combinations of the above command lines as a set of Boolean operations to define our collection of faces or collection of unassociated nodes. The result of this command block must be either a set of faces or a set of nodes.

If you want to define a surface named `name` that is a set of faces, you can use some combination of the command lines `BLOCK`, `SURFACE`, `REMOVE BLOCK`, and `REMOVE SURFACE`. For this case, however, the `BLOCK` and `REMOVE BLOCK` command lines must refer to element blocks that are continuum or shell elements. If the element block referred to is a block of continuum elements, the block is skinned. If the element block referred to is a block of shell elements, the top and bottom faces of the shell elements will form the contact faces.

Suppose you specify a `BLOCK` command line that references several continuum blocks. The set of faces defining the surface will consist of the exterior faces for all the element blocks. If you want to preserve the list of element blocks on the `BLOCK` command line while removing the exterior faces associated with one or more of the blocks, you could simply add a `REMOVE BLOCK` command line listing only those blocks whose associated faces are to be removed from the contact surface.

Suppose you specify a `BLOCK` command line that references a block of continuum elements and a `SURFACE` command line that references a side set. Then the contact surface produced by the command block will be the union of the faces defined by the skinning of the block of continuum elements and the faces defined in the side set.

Suppose you specify a `BLOCK` command line that references a block of continuum elements and a `REMOVE SURFACE` command line that references a side set. Furthermore, suppose that the side set is a set of faces that is a subset of the set of faces obtained from skinning the continuum block. Then the contact surface produced by the command block will be the set of faces obtained by skinning the continuum block minus the faces in the side set.

As can be seen from the above examples, we can use the command lines `BLOCK`, `SURFACE`, `REMOVE BLOCK`, and `REMOVE SURFACE` as Boolean operators to construct a set of finite element faces defining a surface. The `BLOCK` and `REMOVE BLOCK` command lines should produce (or remove) faces, however, so that we are performing the Boolean operations on like topological entities. See Section 7.2.4 for further information about using a node set that contacts a surface.

If you want to define a set of unassociated nodes for contact with a surface, you can use some combination of the command lines `BLOCK`, `NODE SET`, `REMOVE BLOCK`, and `REMOVE NODE SET`. For this case, however, the `BLOCK` and `REMOVE BLOCK` command lines must refer to element blocks that are SPH elements, which are topologically equivalent to a node.

Suppose you specify a `BLOCK` command line that references a block of SPH elements and a `NODE SET` command line that references a node set within a command block. Then the node set produced by the command block will be the union of the nodes defined by the SPH elements and the nodes defined in the node set.

Suppose you specify a `BLOCK` command line that references a block of SPH elements and a `REMOVE NODE SET` command line that references a node set. Furthermore, suppose that the node set is a set of nodes that is a subset of the set of nodes in the SPH block. Then the set of nodes produced by the command block will be the set of nodes obtained from the SPH block minus the nodes in the node set.

There must be at least one `BLOCK`, `SURFACE`, or `NODE SET` command line in the command block.

7.2.4 Contact Node Set

```
CONTACT NODE SET <string>surface_name
  CONTAINS <string list>nodelist_names
```

As indicated previously, contact interactions may also be defined between a surface and a set of nodes. The `CONTACT NODE SET` command line names a set of nodes (the parameter `surface_name` in the above command line) as a collection of nodes in various node sets specified by the string list `nodelist_names`. All the nodes in the node set can then interact with a contact surface. If a node in the node set defined as `surface_name` attempts to penetrate a contact surface, the node will be moved to the surface through the contact calculations.

The node defined by the `CONTACT NODE SET` command line will be paired with either a mesh surface or an analytic surface when contact interactions are defined. In defining interactions between a contact node set and another surface, the interaction must be defined as a pure master-slave interaction, where the nodes in the contact node set are the slave nodes. The master-slave interaction is defined in the `INTERACTION` command block (see Section 7.15). The easiest way to define the correct relation between the nodes in the node set and the faces in the actual surface is to pair the surface with the `MASTER` command line and the node set with the `SLAVE` command line. Suppose the set of nodes is named `beam_nodes` on the `CONTACT NODE SET` command line and the surface these nodes are paired with is named `plate`. Then the `INTERACTION` command block for the interaction of the node set and surface would contain the command lines below.

```
MASTER = plate
SLAVE = beam_nodes
```

Presto will not detect whether or not you have specified a master-slave relation between a surface and a set of nodes. If the interaction between a surface and a set of nodes defaults to a kinematic partition value of 0.5 and there is only one enforcement iteration, then any nodes that have penetrated the surface will only be moved one-half the penetration distance. Therefore, you should check your input carefully if you have an interaction between a surface and a node set to make sure that the master-slave relation has been properly defined for this interaction.

The `CONTACT NODE SET` command line is used to define contact interactions between SPH particles and other contact surfaces—faces on solid elements, shell/membrane faces, and analytic surfaces. The `CONTACT NODE SET` command line also presents a simple approach for contact between one-dimensional elements (beams, trusses) and other contact surfaces—faces on solid elements, shell/membrane faces, and analytic surfaces. In this case, contact processing will seek to remove interpenetration of the nodes of the one-dimensional elements into the other contact surfaces. The contact capabilities in Presto will not currently handle any contact between two one-dimensional elements.

7.3 Analytic Contact Surfaces

Presto permits the definition of rigid analytic surfaces for use in contact. Contact evaluation between a deformable body and a rigid analytic surface is much faster than contact evaluation between two deformable bodies. Therefore, using a rigid analytic surface is more efficient than using a very stiff deformable body to try to approximate a rigid surface. The commands for defining the rigid analytic surfaces currently available in Presto—plane, cylinder, and sphere—are described next.

7.3.1 Plane

```
BEGIN ANALYTIC PLANE <string>name
  NORMAL = <string>defined_direction
  POINT = <string>defined_point
END [ANALYTIC PLANE <string>name]
```

Analytic planes are not deformable, they cannot be moved, and two analytic planes will not interact with each other. The `ANALYTIC PLANE` command block for defining an analytic plane begins with the input line

```
BEGIN ANALYTIC PLANE <string>name
```

and is terminated with the input line

```
END [ANALYTIC PLANE <string>name] ,
```

where the string `name` is some user-selected name for this particular plane. This name is used to identify the surface in the interaction definitions. The string `defined_direction` in the `NORMAL` command line refers to a vector that has been defined with a `DEFINE DIRECTION` command line; this vector defines the outward normal to the plane. The string `defined_point` in the `POINT` command line refers to a point in a plane that has been defined with a `DEFINE POINT` command line. The deformable body should initially be on the side of the plane defined by the outward normal.

7.3.2 Cylinder

```
BEGIN ANALYTIC CYLINDER <string>name
  CENTER = <string>defined_point
  AXIAL DIRECTION = <string>defined_axis
  RADIUS = <real>cylinder_radius
  LENGTH = <real>cylinder_length
  CONTACT NORMAL = <string>OUTSIDE|INSIDE
END [ANALYTIC CYLINDER <string>name]
```

Analytic cylindrical surfaces are not deformable, they cannot be moved, and two analytic cylindrical surfaces will not interact with each other. The `ANALYTIC CYLINDER` command block for defining an analytic cylindrical surface begins with the command line

```
BEGIN ANALYTIC CYLINDER <string>name
```

and is terminated with the command line

```
END [ANALYTIC CYLINDER <string>name] ,
```

where the string `name` is some user-selected name for this particular cylindrical surface. This name is used to identify the surface in the interaction definitions. The cylindrical surface has a finite length; the cylindrical surface is not an infinitely long surface. To fully specify the location of the cylindrical surface, therefore, you must specify the center point of the cylindrical surface in addition to the axial direction of the cylinder. These quantities, center point and direction, are defined by the `CENTER` and `AXIAL DIRECTION` command lines, respectively. The string `defined_point` in the `CENTER` command line refers to a point that has been defined with a `DEFINE POINT` command line; the string `defined_axis` in the `AXIAL DIRECTION` command line refers to a vector that has been defined with a `DEFINE DIRECTION` command line. The radius of the cylinder is the real value `cylinder_radius` specified with the `RADIUS` command line, and the length of the cylinder is the real value `cylinder_length` specified by the `LENGTH` command line. The length of the cylinder (`cylinder_length`) extends a distance of `cylinder_length` divided by 2 along the cylinder axis in both directions from the center point. If the rigid surface is the outside of the cylinder, you should specify

```
CONTACT NORMAL = OUTSIDE .
```

If the rigid surface is the inside of the cylinder, you should specify

```
CONTACT NORMAL = INSIDE .
```

7.3.3 Sphere

```
BEGIN ANALYTIC SPHERE <string>name
  CENTER = <string>defined_point
  RADIUS = <real>sphere_radius
END [ANALYTIC SPHERE <string>name]
```

Analytic spherical surfaces are not deformable, they cannot be moved, and two analytic spherical surfaces will not interact with each other. The `ANALYTIC SPHERE` command block for defining an analytic spherical surface begins with the input line

```
BEGIN ANALYTIC SPHERE <string>name
```

and is terminated with the input line

```
    END [ANALYTIC SPHERE <string>name] ,
```

where the string `name` is some user-selected name for this particular spherical surface. This name is used to identify the surface in the interaction definitions. The center point of the sphere is defined by the `CENTER` command line, which references a point, `defined_point`, specified by a `DEFINE POINT` command line. The radius of the sphere is the real value `sphere_radius` specified with the `RADIUS` command line.

7.4 Update All Surfaces for Element Death

```
UPDATE ALL SURFACES FOR ELEMENT DEATH
  = <string>ON|OFF(ON)
```

When elements are killed in an analysis, contact surfaces may need to be updated to account for the removal of faces attached to killed elements or the addition of faces exposed by element death. The command line `UPDATE ALL SURFACES FOR ELEMENT DEATH` permits contact surfaces to be updated based on all the `ELEMENT DEATH` command block(s) specified in the input file (see Section 5.5). This update of contact surfaces is controlled by the command line being set to `ON`, the default. The update encompasses the full reinitialization of contact. Thus, surface-physics models that involve state data may lose some information when the new contact surfaces are created. If the command line is set to `OFF`, an element associated with a face on the contact surface could be killed, but the face would remain in the list of faces defining the contact surface, which may be unacceptable for your analysis.

7.5 Remove Initial Overlap

```
BEGIN REMOVE INITIAL OVERLAP
  OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
  OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
  SHELL OVERLAP ITERATIONS = <integer>max_iter(10)
  SHELL OVERLAP TOLERANCE = <real>shell_over_tol(0.0)
END REMOVE INITIAL OVERLAP
```

Meshes supplied for finite element analyses frequently have some level of initial mesh overlap, where finite element nodes rest inside the volume of elements. This can cause problems with contact; overlaps may cause initial forces that are nonphysical and produce erroneous stress waves. Presto provides a mechanism to modify the initial mesh to attempt to remove overlaps in surfaces defined for contact via the `REMOVE INITIAL OVERLAP` command block.

The process used to remove the initial overlap for three-dimensional solid elements involves changing the original coordinates of nodes on contact surfaces. Changing the coordinates yields a new mesh with the overlap removed; the overlap removal adds no initial stresses. Normal and tangential tolerances are specified by the user for all the contact surfaces in the `REMOVE INITIAL OVERLAP` command block. It is also possible to specify overlap normal and tangential tolerances on each surface pairing separately in the `INTERACTION` command block. In other words, overlap removal tolerances specified in `INTERACTION` command blocks will overwrite the tolerances specified in the `REMOVE INITIAL OVERLAP` command block—see Section 7.15). The `REMOVE INITIAL OVERLAP` command block only removes overlaps that are detected along the surfaces defined for contact and not all surfaces in the mesh.

Overlap tolerances are used to designate a box around each surface pair to search for overlaps. If the overlap of the mesh is larger than the box defined by the tolerances, then the overlap will not be found and thus will not be removed. However, if the specified tolerances are larger than an element length in the analysis, the overlap removal mechanism may invert elements, leading to analysis failure. This has two ramifications. First, the tolerances must be carefully specified to correct mesh overlaps and to not invert elements. Second, this mechanism is unable to remove initial overlaps that are greater than an element length. In such cases, the overlap must be removed manually using a meshing tool. The mesh modification done by the `REMOVE INITIAL OVERLAP` feature changes the meshed geometry, and thus can change the mass and time step of affected elements. The mesh returned in the results file includes the changed coordinates and should be checked to ensure that the modifications are acceptable. A summary of the overlap that is removed is reported in the log file. (See Section 1.7 for a discussion of the log file.) The log file lists each block in which the initial overlap has been removed as well as the maximum amount of overlap for each of these blocks. Additionally, you can request that a nodal variable called `RE-`

`MOVED_OVERLAP` be written to the results file. See Section 8.1.1.1 for a discussion of the output of nodal variables to the results file.

For contact between shell elements, a slightly different approach is used. Because the thickness of a shell must be preserved when shell lofting is requested, removing the initial overlap between nested shells becomes an iterative process whereby shell locations are adjusted to remove the overlap. This process is approximate and may not remove all the overlap in all cases. It is advised to check the corrected mesh to make sure that the mesh modifications are acceptable. In the input, two additional input lines, `SHELL OVERLAP ITERATIONS` and `SHELL OVERLAP TOLERANCE`, may be needed to properly remove the initial overlap.

- The `SHELL OVERLAP ITERATIONS` command line controls the maximum number of iterations that will be used by the overlap removal mechanism to resolve nested shells. By default, the value of `max_iter` is 10. If the mesh has only a few layers of shells that may overlap, a value of 10 should suffice. However, if the mesh has a number of layers of shells that may overlap, this value may need to be much larger.

- The `SHELL OVERLAP TOLERANCE` command line specifies an amount of overlap, `shell_over_tol`, that is permitted to be left in the shell elements. This helps to limit the actual number of iterations required to remove the shell overlap, and to spread any remaining overlap over a number of shells instead of concentrating it all in a single shell. If the default value of 0.0 for the shell overlap tolerance is used, iteration continues until either all the overlap is removed or the maximum number of iterations is reached. If a nonzero value for the shell overlap tolerance is used, iteration continues until the tolerance is reached or the maximum number of iterations is reached. Note that the overlap removal process is only done once during an analysis, so a large number of iterations will only affect the first time step, not every time step.

The `SHELL OVERLAP TOLERANCE` command line and the `SHELL OVERLAP ITERATIONS` command line have no meaning for analyses that do not have shell elements.

7.6 Angle for Multiple Interactions

```
MULTIPLE INTERACTIONS = <string>ON|OFF(ON)
MULTIPLE INTERACTIONS WITH ANGLE = <real>angle(60.0)
```

When a node lies on the edge of a body, that node may need to support contact interactions with more than one face at the same time. For instance, see Figure 7.3. In Figure 7.3a, three blocks are shown, with a single node identified. Through contact, this node can interact with both the block on the upper right and the block on the bottom. If the node only supports a single interaction, then it will be arbitrarily considered for contact between one of the blocks, but not the other, as in Figure 7.3b. In this case, contact enforcement will prevent penetration into the lower block, but may permit penetration into the upper right block. The proper way to deal with this case is shown in Figure 7.3c, where multiple interactions are considered at the node.

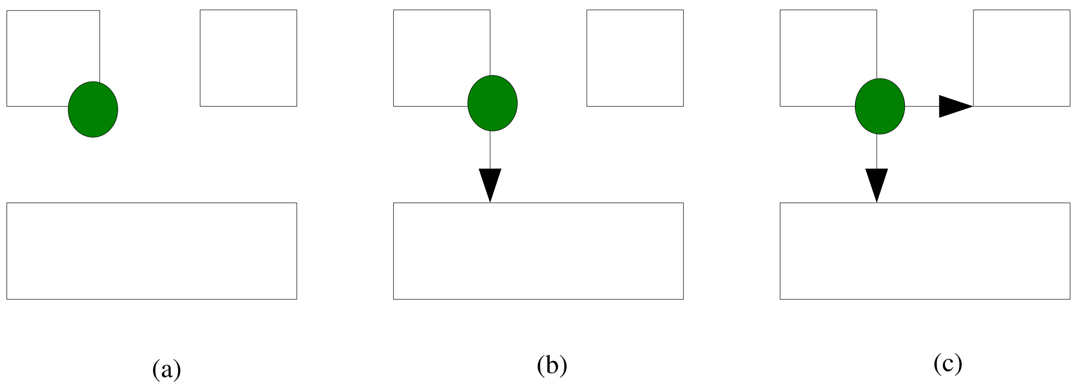


Figure 7.3: Illustrations of multiple interactions at a node: (a) initial configuration, with node of interest identified; (b) single interaction; and (c) multiple interactions.

By default, Presto permits multiple interactions at a node. However, these multiple interactions may incur extra cost in the contact algorithm by increasing the number of interactions in enforcement. Also, a local search algorithm (see Section 7.12), which uses various contact tracking approaches, may operate more efficiently when the node can only have one interaction. Finally, multiple interactions may lead to instabilities that can be eliminated by switching to single interactions. For these reasons, the `MULTIPLE INTERACTIONS` command line allows the user to choose whether multiple interactions should be considered at a node. A value of `OFF` indicates that a node can have only one interaction. This value affects all interactions in a contact definition. Presto does not currently have the capability to force single interactions for some surface pairs while allowing multiple interactions for other surface pairs.

When the `MULTIPLE INTERACTIONS` command line is `ON`, the number of interactions that can be considered at a node is dependent on the measure of curvature of those faces that are connected to the node. If the angle between two faces on which the node is attached is small, then only one interaction is allowed. However, in cases where the angle between the faces is large enough such that they form a discrete corner, multiple interactions are considered. The contact algorithms can properly handle only a limited number of interactions per node (currently three), so it is generally feasible to properly define interactions at a node, e.g., at the corner of a block.

Presto defines the critical angle for multiple interactions via the `MULTIPLE INTERACTIONS WITH ANGLE` command line, where `angle` is the angle over which an edge is considered sharp. If the angle between adjoining faces is greater than this critical angle, multiple interactions can be created. By default, this critical angle is 60 degrees, which works well for most analyses. This value can be changed in the contact input if needed.

7.7 Surface Normal Smoothing

```
SURFACE NORMAL SMOOTHING = <string>ON|OFF(OFF)
```

Surface normal smoothing is a feature that is primarily used in Adagio, a quasi-static code.

Suppose that a node has penetrated a face near an edge that is shared with an adjacent face. Now consider the case where the faces sharing that edge do not lie in a plane. In this case, the angle between the faces is other than 180 degrees, and there is a discontinuity in the normal at the edge. One face lying adjacent to the edge has one normal, and the other face lying adjacent to the same edge has another normal. If a node penetrates one of the faces close to the edge, an iterative solver, like a solver used in Adagio, might have a difficult time converging to a solution at the edge. On one iteration, the node might penetrate the node on one side of the edge, but, on the next iteration, the node might penetrate the face on the other side of the edge. For these two iterations, the push-back direction for the node will be different because of the different normals for each face. It is difficult for an iterative solver to converge to a solution in the case we have just described—contact (penetration) alternating between two faces at an edge with a discontinuous normal at the edge.

The surface-normal smoothing technique creates a smooth variation in the normal near the edge. The normal varies linearly from the value on one face to the value on the other face over a distance that spans the edge. A smoothly varying normal at the edge makes it much easier for an iterative solver to converge to a solution for a case where a node has penetrated a face somewhere near the edge.

Presto does not use an iterative solver and does not encounter the problems with nodes penetrating near an edge that are encountered by an iterative solver. Consequently, the `SURFACE NORMAL SMOOTHING` command line is not really required. However, if you couple a Presto analysis with an Adagio analysis (the Adagio analysis preceding the Presto analysis), you should use the `SURFACE NORMAL SMOOTHING` command line in Presto. Using the surface smoothing in Presto will create a consistent contact transition between the two codes, Presto and Adagio.

7.8 Eroded Face Treatment

```
ERODED FACE TREATMENT = <string>ALL|NONE(ALL)
```

The `ERODED FACE TREATMENT` command line is used to define what happens to newly exposed element faces when a contact surface erodes because of element death. This command line applies to the case in which a contact surface has been generated by the skinning of an element block (Section 7.2). Suppose we have a contact block that has been skinned to create a contact surface, and let us consider an element that contributes a face to the original contact surface. If this element is killed at some point by element death, the death of this element exposes new faces. If the `ALL` option in the command line has been selected, any newly exposed faces will be included in the updated contact definition. If the `NONE` option is used, the faces exposed by element death will not be included in the updated contact surface. Both options will remove any faces on killed elements from the contact definition, though the `NONE` option tends to be more robust on complex geometries, such as those containing equivalenced elements, shell elements sandwiched between solid elements, and degenerate elements.

The default option is `ALL`.

7.9 Shell Lofting

```
BEGIN SHELL LOFTING
  LOFTING ALGORITHM = <string>ON|OFF(ON)
  COINCIDENT SHELL TREATMENT = <string>DISALLOW|IGNORE|
    SIMPLE(DISALLOW)
  COINCIDENT SHELL HEX TREATMENT = <string>DISALLOW|
    IGNORE|TAPERED|EMBEDDED(DISALLOW)
END [SHELL LOFTING]
```

Presto can also assess contact on shell elements. Shell elements can interact with other shell elements, faces of solid elements, and contact node sets (such as SPH). Contact on shell elements can occur on either the meshed shell geometry, i.e., ignoring any shell thickness, or on the "lofted" geometry, i.e., a geometry that includes the thickness of the shell. Currently, contact appears to be more robust on the nonlofted geometry; however, for simulations in which the thickness is important, the lofted geometry can provide more-reasonable results. Also critical to shell contact is how shells that are fully coincident with other elements (i.e., share all their nodes with another element) are treated. These options are controlled by the user in the `SHELL LOFTING` command block.

The `LOFTING ALGORITHM` command line determines whether contact on a shell should be done on the lofted geometry or on the original shell geometry. If the `LOFTING ALGORITHM` command line is set to `ON`, shell contact uses the lofted geometry; if the command line is set to `OFF`, shell contact uses the original shell geometry.

The `COINCIDENT SHELL TREATMENT` command line identifies how shells that share the same nodes should be treated. If the `DISALLOW` option is selected, (the default), then any time that shells in contact are detected to share all the same nodes, the code will abort with an error message indicating which elements were found to be coincident. The `DISALLOW` option should be used if you do not want any coincident shells to be considered in the analysis. The option operates essentially as a check on the mesh. If the `IGNORE` option is selected, any contact faces attached to coincident shells are ignored for contact. This option is only provided as a backup approach if undiagnosed code problems arise from coincident shells. If such a case occurs, the `IGNORE` option may permit the user to continue with an analysis while the code team diagnoses the problem. The `SIMPLE` option enables coincident shells to be processed correctly. If lofting has been enabled and the `SIMPLE` option is selected, the thickness of the lofted coincident shell is taken as the largest thickness of all the coincident shells. If lofting is off and the `SIMPLE` option is selected, the coincident shell is treated as if only one of the shells is present.

The `COINCIDENT SHELL HEX TREATMENT` command line is similar to the `COINCIDENT SHELL TREATMENT` command line. The `COINCIDENT SHELL HEX TREAT-`

MENT command line, however, identifies how shells that are fully coincident with the hex elements are treated. If the DISALLOW option is selected (the default), then any time that a shell in contact is detected to share all the same nodes with the face of a continuum element, the code will abort with an error message indicating which elements were found to be coincident. The DISALLOW option should be used if you do not want any shells coincident with hexes to be considered in the analysis. The option operates essentially as a check on the mesh. If the IGNORE option is selected, any contact faces attached to shells that are coincident with faces of continuum elements are ignored for contact. This option is only provided as a backup approach if undiagnosed code problems arise from coincident shells and continuum elements. If such a case occurs, the IGNORE option may permit the user to continue with an analysis while the code team diagnoses the problem. The TAPERED and EMBEDDED options permit shells that are coincident with faces of continuum elements to be processed in contact. The TAPERED option does two things: it includes for contact any faces that are on the free surface and ignores faces sandwiched between the shell and the continuum element, and it automatically adjusts the lofting of the surfaces to provide a smooth transition between shells that are not coincident with the faces of the continuum elements and those that are coincident with the faces of the continuum elements. The EMBEDDED option includes for contact both free surface faces and those that are between the coincident shells and faces of the continuum elements; the option does not adjust thicknesses to make smooth transitions between shells that are not coincident with faces of continuum elements and those that are coincident with faces of continuum elements. In general, the TAPERED option is preferred; only use the EMBEDDED option if the TAPERED option causes a code problem.

7.10 Contact Output Variables

```
CONTACT VARIABLES = ON|OFF(OFF)
```

To provide more information about the enforcement of contact interactions, Presto can provide additional contact variables for output. The `CONTACT VARIABLES` command line permits the user to activate the computation of these variables. Because the computation of these variables incurs additional computational cost, the variables are deactivated, i.e., the default is set to `OFF`. Currently, information on only one interaction at each node is provided. If a node has more than one interaction, the last one in its internal interaction list is reported.

The additional nodal contact variables activated by the `CONTACT VARIABLES` command are listed in Table 7.1. The variables can be output in history files or results files; see Chapter 8 for more information on outputting nodal variables. Note that the `CONTACT VARIABLES` command line activates the computation of these variables. If this command is set to `OFF`, then the variables will all be zero when output.

Table 7.1: Nodal Variables for Output

Variable	**Description**
contact_status	Status of the interactions at the node. Possible values are as follows: 0.0 = Node is not a contact node (not in a defined contact surface) 0.5 = Node is not in contact 1, 2, or 3 = Node has 1, 2, or 3 interactions
contact_normal_direction	Vector direction of the constraint. This is, in general, the normal of the face in the interaction.
contact_tangential_direction	Velocity of the node relative to the face times the time step minus the component of this vector along the normal to the face. Note that this vector is NOT normalized.

Continued on next page

Table 7.1 – Continued from previous page

Variable	**Description**
contact_normal_force_magnitude	Magnitude of the contact force at the node in the direction normal to the contact face (contact_normal_direction).
contact_tangential_force_magnitude	Magnitude of the contact force at the node in the plane of the contact face (contact_tangential_direction).
contact_normal_traction_magnitude	Traction normal to the contact face, i.e., contact_normal_force_magnitude scaled by contact_area. If there are multiple interactions for this node, the traction only for the last interaction is given.
contact_tangential_traction_magnitude	Traction in the plane of the contact face, i.e., contact_traction_force_magnitude scaled by contact_area. If there are multiple interactions for this node, the traction only for the last interaction is given.
contact_slip_increment_current	Increment of slip tangential to the face that occurs over the time step.
contact_frictional_energy_dissipation	Amount of frictional energy dissipated over the time step.
contact_area	Contact area for the node. This is the tributary area around the node for this interaction. If there are multiple interactions, the reported area is the area associated with the last interaction.
contact_current_gap	Value of the gap for the current time step. If the node has multiple interactions, the reported gap is for the last interaction.
contact_previous_gap	Value of the gap for the previous time step. If the node has multiple interactions, the reported gap is for the last interaction.

7.11 Friction Models

To describe the physics of interactions that occur between contact surfaces, the Presto input for contact relies upon the definition of friction models. The user then relates these friction models to pairs of interactions in the interaction-definition blocks (see Section 7.14 and Section 7.15). During the search phase of contact, node-face interactions are identified, and the designated friction model is used to determine how the resulting contact forces are resolved between these pairs.

Currently, there are 11 primary friction models: frictionless contact, constant coulomb friction, tied contact, spring weld, surface weld, area weld, adhesion, cohesive zone, junction, threaded joint, and pressure-velocity–dependent friction. In addition, the SIMOD interface models can be used as friction models, as well as models defined by user subroutines. By default, interactions between contact surfaces that have not had friction models assigned are treated as frictionless. All friction models are command blocks, although some of the models do not have any command lines inside the command block. The commands for defining the available friction models are described next. Friction models are associated with specific pairings of contact surfaces through the interaction-definition blocks in Section 7.14 and Section 7.15. Presto uses the ACME library for contact enforcement. See the documentation for ACME to obtain a more in-depth description of the implementation and usage for the various friction models.

7.11.1 Frictionless Model

```
BEGIN FRICTIONLESS MODEL <string>name
END [FRICTIONLESS MODEL <string>name]
```

The `FRICTIONLESS MODEL` command block defines frictionless contact between surfaces. In frictionless contact, contact forces are computed normal to the contact surfaces to prevent penetration, but no forces are computed tangential to the contact surfaces. The string `name` is a user-selected name for this friction model that is used when identifying this model in the interaction definitions. No command lines are needed inside the command block.

7.11.2 Constant Friction Model

```
BEGIN CONSTANT FRICTION MODEL <string>name
  FRICTION COEFFICIENT = <real>coeff
END [CONSTANT FRICTION MODEL <string>name]
```

The `CONSTANT FRICTION MODEL` command block defines a constant coulomb friction coefficient between two surfaces as they slide past each other in contact. No resistance is provided to keep the surfaces together if they start to separate. The string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions, and `coeff` is the constant coulomb friction coefficient. There is no default value for the friction coefficient.

7.11.3 Tied Model

```
BEGIN TIED MODEL <string>name
END [TIED MODEL <string>name]
```

The `TIED MODEL` command block restricts nodes found in initial contact with faces to stay in the same relative location to the faces throughout the analysis. The string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. No command lines are needed inside the command block.

7.11.4 Spring Weld Model

```
BEGIN SPRING WELD MODEL <string>name
  NORMAL DISPLACEMENT FUNCTION = <string>func_name
  NORMAL DISPLACEMENT SCALE FACTOR = <real>scale_factor(1.0)
  TANGENTIAL DISPLACEMENT FUNCTION = <string>func_name
  TANGENTIAL DISPLACEMENT SCALE FACTOR =
    <real>scale_factor(1.0)
  FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
  FAILURE DECAY CYCLES = <integer> num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
 END [SPRING WELD MODEL <string>name]
```

The `SPRING WELD MODEL` command block defines a contact friction model that, when applied between two contact surfaces, connects a slave node to the nearest point of a corresponding master face with a spring. The spring behavior is defined by a force-displacement curve in the normal and tangential directions. If the motion of the problem generates displacement between the slave node and its corresponding master face and this motion is in purely the normal or tangential direction, the spring will fail once it passes the maximum displacement in the normal and tangential force-displacement curves, respectively. For displacements that include both normal and tangential components, the spring fails according to a failure criterion defined as the sum of the ratios of the normal and tangential components to their maximum values, raised to a power. If the criterion is greater than 1.0, the spring fails. Once the spring fails, its contact forces reduce over a number of load steps, and the contact evaluation reverts to another user-specified friction model (or frictionless contact if not specified).

In the above command block:

- The string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions.
- The normal force-displacement curve is specified by the `NORMAL DISPLACEMENT FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the domain scope. This function can be scaled by the real value `scale_factor` in the `NORMAL DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0.
- The tangential force-displacement curve is specified by the `TANGENTIAL DISPLACEMENT FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the domain scope. This function can be scaled by the real value `scale_factor` in the `TANGENTIAL DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0.
- The real value `exponent` in the `FAILURE ENVELOPE EXPONENT` command line specifies how normal and tangential failure criteria may be combined to yield failure of the weld, as described above. The default value for this exponent is 2.0.
- The `FAILURE DECAY CYCLES` command line describes how many cycles to ramp down the load in the spring weld after it fails through the integer value `num_cycles`. The default value for the number of decay cycles is 1.
- When the spring weld breaks, the friction model that contact reverts to when evaluating future node-face interactions between the surfaces is identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model listed in this command must have been previously defined in the input file. The default value for the model used after failure is the frictionless model.

The `SPRING WELD MODEL` command block is very similar to the Presto `SPOT WELD` command block, but permits greater flexibility in specifying a different friction model to be applied after failure.

7.11.5 Surface Weld Model

```
BEGIN SURFACE WELD MODEL <string>name
  NORMAL CAPACITY = <real>normal_cap
  TANGENTIAL CAPACITY = <real>tangential_cap
  FAILURE DECAY CYCLES = <integer>num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
END [SURFACE WELD MODEL <string>name]
```

The `SURFACE WELD MODEL` command block defines a contact friction model that behaves identically to the `TIED MODEL` until a maximum force between the node and face of an interaction is reached in the normal direction or the tangential direction. Once this maximum force is reached, the tied contact “fails” and the friction model switches to a different friction model, as specified by the user.

In the above command block, the string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The maximum allowed force in the normal direction is specified by the real value `normal_cap` in the `NORMAL CAPACITY` command line. The maximum allowed force in the tangential direction is specified by the real value `tangential_cap` in the `TANGENTIAL CAPACITY` command line. There are no defaults for these values. The surface weld will break when either the specified normal or tangential capacity is reached. Once the model fails, the applied forces decrease to zero over a number of time steps defined through the integer value `num_cycles` in the `FAILURE DECAY CYCLES` command line. The default for `num_cycles` is 1. The friction model that should be used after the weld fails is identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model designated in the `FAILED MODEL` command line must be defined within the `CONTACT DEFINITION` command block. The default model after failure is the frictionless contact model.

7.11.6 Area Weld Model

```
BEGIN AREA WELD MODEL <string>name
  NORMAL CAPACITY = <real>normal_cap
  TANGENTIAL CAPACITY = <real>tangential_cap
  FAILURE DECAY CYCLES = <integer> num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
END [AREA WELD MODEL <string>name]
```

The `AREA WELD MODEL` command block defines a contact friction model that behaves identically to the `TIED MODEL` until a maximum traction between a node and face in an interaction is reached in the normal direction or the tangential direction. Once this maximum traction is reached, the tied contact "fails" and the friction model switches to a different friction model, as specified by the user. This model is identical to the `SURFACE WELD MODEL` command block, except that tractions are used instead of forces.

In the above command block, the string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The maximum allowed traction in the normal direction is specified by the real value `normal_cap` in the `NORMAL CAPACITY` command line. The maximum allowed traction in the tangential direction is specified by the real value `tangential_cap` in the `TANGENTIAL CAPACITY` command line. There are no defaults for these values. The area weld will break when either the specified normal or tangential capacity is reached. Once the model fails, the applied tractions decrease to zero over a number of time steps defined through the integer value `num_cycles` in the `FAILURE DECAY CYCLES` command line. The default for `num_cycles` is 1. The friction model that should be used after the weld fails is identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model designated in the `FAILED MODEL` command line must be defined within the `CONTACT DEFINITION` command block. The default model after failure is the frictionless contact model.

7.11.7 Adhesion Model

```
BEGIN ADHESION MODEL <string>name
  ADHESION FUNCTION = <string>func_name
  ADHESION SCALE FACTOR = <real>scale_factor(1.0)
END [ADHESION MODEL <string>name]
```

The `ADHESION MODEL` command block defines a friction model that behaves like frictionless contact when two surfaces are in contact, but computes an additional force between the surfaces when they are not touching. The value of the additional force is given by a user-specified function of force versus distance, where the distance is the distance between a node and the closest point on the opposing surface.

In the above command block, the string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The force between surfaces that are not touching is given by the `ADHESION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. The values of this function are expected to be nonnegative. The function can be scaled by the real value `scale_factor` in the `ADHESION SCALE FACTOR` command line; the default for this factor is 1.0. Because contact forces are typically only given to node-face interactions if they touching, the contact search requires appropriate tolerances when this model is used. The normal and tangential tolerances specified in the interaction definitions should be set to the maximum distance at which the adhesion model should be applying force. However, setting this distance to be very large may cause excessive numbers of interactions to be identified in the search phase, causing the contact processing to be very slow and/or generate erroneous interactions.

7.11.8 Cohesive Zone Model

```
BEGIN COHESIVE ZONE MODEL <string>name
  TRACTION DISPLACEMENT FUNCTION = <string>func_name
  TRACTION DISPLACEMENT SCALE FACTOR =
    <real>scale_factor(1.0)
  CRITICAL NORMAL GAP = <real>crit_norm_gap
  CRITICAL TANGENTIAL GAP = <real>crit_tangential_gap
END [COHESIVE ZONE MODEL <string>name]
```

The `COHESIVE ZONE MODEL` command block defines a friction model that prevents penetration when contact surfaces are touching, but provides an additional force when the distance between the node and face in an interaction increases. This force is determined by a user-specified function. Once the distance exceeds a user-specified value in the normal direction or the tangential direction, the force is no longer applied. This model can be used to mimic the energy required to separate two surfaces that are initially touching.

In the above command block, the string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The displacement function for traction is given by the `TRACTION DISPLACEMENT FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. This function can be scaled by the real value `scale_factor` in the `TRACTION DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0. In the `CRITICAL NORMAL GAP` command line, the real value `crit_norm_gap` specifies the normal distance between the node and face past which the cohesive zone no longer provides a force. In the `CRITICAL TANGENTIAL GAP` command line, the real value `crit_tangential_gap` specifies the tangential distance between the node and face past which the cohesive zone no longer provides a force.

7.11.9 Junction Model

```
BEGIN JUNCTION MODEL <string>name
  NORMAL TRACTION FUNCTION = <string>func_name
  NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
```

```
    TANGENTIAL TRACTION FUNCTION = <string>func_name
    TANGENTIAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
    NORMAL CUTOFF DISTANCE FOR TANGENTIAL TRACTION =
      <real>distance
  END [JUNCTION MODEL <string>name]
```

The `JUNCTION MODEL` command block defines a model that prevents the interpenetration of contact surfaces and that also provides normal and tangential tractions to a node-face interaction when the surfaces are not touching. The normal tractions are defined as a function of the normal distance between the node and face of an interaction, while the tangential traction is given as a function of the relative tangential velocity. The tractions are defined by user-specified functions, and the tangential tractions from this model drop to zero once the normal distance between the node and the face exceeds a critical value. This friction model provides a simple way to model threaded connections, though the `THREADED MODEL` defined in Section 7.11.10 has more flexibility.

In the above command block, the string `name` is a user-selected name for this friction model that is used to identify this model in the interaction blocks. The normal traction curve is specified by the `NORMAL TRACTION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. This function defines a relation between the traction and the distance between the node and the face in the normal direction. This function can be scaled by the real value `scale_factor` in the `NORMAL TRACTION SCALE FACTOR` command line; the default for this factor is 1.0. Similarly, the tangential traction curve is specified by the `TANGENTIAL TRACTION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. This function defines a relation between the traction and the relative velocity of the node and face in the tangential direction. This function can be scaled by the real value `scale_factor` in the `TANGENTIAL TRACTION SCALE FACTOR` command line; the default for this factor is 1.0. Once the normal distance between a node and a face using this model reaches a critical distance, the tangential traction drops to zero; this distance is specified with the real value `distance` in the `NORMAL CUTOFF DISTANCE FOR TANGENTIAL TRACTION` command line.

7.11.10 Threaded Model

```
  BEGIN THREADED MODEL <string>name
    NORMAL TRACTION FUNCTION = <string>func_name
    NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
    TANGENTIAL TRACTION FUNCTION = <string>func_name
    TANGENTIAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
    TANGENTIAL TRACTION GAP FUNCTION = <string>func_name
    TANGENTIAL TRACTION GAP SCALE FACTOR =
      <real>scale_factor(1.0)
    NORMAL CAPACITY = <real>normal_cap
    TANGENTIAL CAPACITY = <real>tangential_cap
    FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
    FAILURE DECAY CYCLES = <integer>num_cycles(1)
    FAILED MODEL = <string>failed_model_name|FRICTIONLESS
      (FRICTIONLESS)
  END [THREADED MODEL <string>name]
```

The `THREADED MODEL` command block defines a friction model that is designed to mimic a threaded interface. This model prevents interpenetration of contact surfaces, and also supplies additional tractions when the surfaces are not touching. Tensile tractions in the normal direction are given by a user-specified function of force versus distance between the node and face. Tensile tractions in the tangential direction are computed as the product of a traction tangential-displacement curve and a scaling curve that is a function of the normal displacement. Maximum normal and tangential tractions are input such that the model "fails" at a node-face interaction once they are reached. For interactions that include both normal and tangential displacements, the model failure is defined according to a failure criterion defined as the sum of the ratios of the normal and tangential traction components to their maximum capacity values, raised to a power. After failure, interactions shift to a different user-specified friction model.

In the above command block:

- The string `name` is a user-selected name for this friction model that is used to identify the model in the interaction definitions.
- The traction-displacement relation in the normal direction is specified by the `NORMAL TRACTION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. This function can be scaled by the real value `scale_factor` in the `NORMAL TRACTION SCALE FACTOR` command line; the default for this factor is 1.0.
- The traction-displacement relation in the tangential direction is specified by two curves. The traction-displacement relation in the tangential direction when there is no displacement in the normal direction is defined by the `TANGENTIAL TRACTION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. This function can be scaled by the real value `scale_factor` in the `TANGENTIAL TRACTION SCALE FACTOR` command line; the default for this

factor is 1.0. When the distance in the normal direction is greater than zero, the tangential traction is scaled by the `TANGENTIAL TRACTION GAP FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command block in the domain scope. This function defines a scaling factor as a function of the normal displacement. The function can be scaled by the real value `scale_factor` in the `TANGENTIAL TRACTION GAP SCALE FACTOR` command line; the default for this factor is 1.0.

- The threaded model "fails" once the normal and tangential tractions reach a critical capacity value. The normal capacity is specified by the real value `normal_cap` in the `NORMAL CAPACITY` command line. The tangential capacity is specified by the real value `tangential_cap` in the `TANGENTIAL CAPACITY` command line. There are no default values for these parameters. These capacities are defined for pure normal or tangential displacements. In cases where there is a combination of tangential and normal displacements, a failure curve is used to determine the combination of tangential and normal tractions that determines model failure. This curve is defined as the sum of the ratios of the normal and tangential traction components to their maximum capacity values, raised to a power. The power in the function is defined by the real value exponent in the `FAILURE ENVELOPE EXPONENT` command line. The default value of the exponent is 2.0. Once the model fails, the applied tractions decrease to zero over a number of time steps defined through the integer value `num_cycles` in the `FAILURE DECAY CYCLES` command line. The default for `num_cycles` is 1. When the model exceeds the designated capacity, the contact surfaces using this model switch to a different friction model as identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model designated in the `FAILED MODEL` command line must be defined within the `CONTACT DEFINITION` command block. The default model is the frictionless model.

7.11.11 PV_Dependent Model

```
BEGIN PV_DEPENDENT MODEL <string>name
  STATIC COEFFICIENT = <real>stat_coeff
  DYNAMIC COEFFICIENT = <real>dyn_coeff
  VELOCITY DECAY = <real>vel_decay
  REFERENCE PRESSURE = <real>p_ref
  OFFSET PRESSURE = <real>p_off
  PRESSURE EXPONENT = <real>p_exp
END [PV_DEPENDENT MODEL <string> name]
```

The `PV_DEPENDENT MODEL` command block defines a friction model similar to a coulomb friction model, but which provides a frictional response that is dependent on the pressure and the velocity. The pressure-dependent portion of the model behaves similarly to the constant friction model except that the tangential traction is given by

$$\left[\frac{p + \text{p\_off}}{\text{p\_ref}}\right]^{\text{p\_exp}}. \tag{7.1}$$

The velocity-dependent part is given by

$$(\text{stat\_coeff} - \text{dyn\_coeff})\, e^{(-\text{vel\_decay}\|v\|)} + \text{dyn\_coeff}. \tag{7.2}$$

The `PV_DEPENDENT MODEL` command block multiplies the pressure and velocity effects together.

In the above command block:

- The string `name` is a name assigned to this friction model that is used to identify the model in the interaction definitions.
- The real value `p_ref` in the pressure-dependent part given in Equation (7.1) is specified with the `REFERENCE PRESSURE` command line.
- The real value `p_off` in the pressure-dependent part given in Equation (7.1) is specified with the `OFFSET PRESSURE` command line.
- The real value `p_exp` in the pressure-dependent part given in Equation (7.1) is specified with the `PRESSURE EXPONENT` command line.
- The real value `stat_coeff` in the velocity-dependent part given in Equation (7.2) is specified with the `STATIC COEFFICIENT` command line.
- The real value `dyn_coeff` in the velocity-dependent part given in Equation (7.2) is specified with the `DYNAMIC COEFFICIENT` command line.
- The real value `vel_decay` in the velocity-dependent part given in Equation (7.2) is specified with the `VELOCITY DECAY` command line.

7.11.12 SIMOD Friction Models

```
BEGIN SIMOD SHARED MODEL <string>name
  USE SIMOD MODEL = <string>simod_model_name
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
```

```
END [SIMOD SHARED MODEL <string>name]

BEGIN SIMOD UNKNOWN MODEL <string>name
  USE SIMOD MODEL = <string>simod_model_name
  FAILED MODEL = <string>failed_model_name|
    FRICTIONLESS(FRICTIONLESS)
END [SIMOD UNKNOWN MODEL <string>name]
```

The `SIMOD SHARED MODEL` and `SIMOD UNKNOWN MODEL` command blocks permit contact to use SIMOD models defined in the input file. The two command blocks handle standard shared SIMOD models and special unknown SIMOD models, respectively. See Chapter 10 for more information about the SIMOD models and how to specify them within the input file.

The string `name` is a user-selected name for the model that is used to identify the model in the interaction definitions. The `USE SIMOD MODEL` command line specifies the SIMOD model defined in the input file with the name `simod_model_name`. The friction model that should be used for contact evaluation after the SIMOD model fails is identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model designated in the `FAILED MODEL` command line must be defined within the `CONTACT DEFINITION` command block. The default failed model is the frictionless model.

7.11.13 User Subroutine Friction Models

```
BEGIN USER SUBROUTINE MODEL <string>name
  INITIALIZE MODEL SUBROUTINE = <string>init_model_name
  INITIALIZE TIME STEP SUBROUTINE = <string>init_ts_name
  INITIALIZE NODE STATE DATA SUBROUTINE =
    <string>init_node_data_name
  LIMIT FORCE SUBROUTINE = <string>limit_force_name
  ACTIVE SUBROUTINE = <string>active_name
  INTERACTION TYPE SUBROUTINE = <string>interaction_name
END [USER SUBROUTINE MODEL <string>name]
```

The `USER SUBROUTINE MODEL` command blocks permit contact to use a user subroutine to define a friction model between surfaces. This capability is in a test phase at this time; please contact a Presto developer for more information.

In this command block:

- The string `name` is a user-specified name that is used to identify this model in the interaction definitions.

- The command line `INITIALIZE MODEL SUBROUTINE` specifies a user subroutine to initialize the friction model. The name of the subroutine is given by `init_model_name`.
- The command line `INITIALIZE TIME STEP SUBROUTINE` specifies a user subroutine to initialize the time step. The name of the subroutine is given by `init_ts_name`.
- The command line `INITIALIZE NODE STATE DATA SUBROUTINE` specifies a user subroutine to initialize the node state data. The name of the subroutine is given by `init_node_data_name`.
- The command line `LIMIT FORCE SUBROUTINE` specifies a user subroutine to provide the limit force for the friction model. The name of the subroutine is given by `limit_force_name`.
- The command line `ACTIVE SUBROUTINE` specifies a user subroutine to compute forces for an active node-face interaction. The name of the subroutine is given by `active_name`.
- The command line `INTERACTION TYPE SUBROUTINE` specifies a user subroutine to define the type of the interaction. The name of the subroutine is given by `interaction_name`.

7.12 Search Options

```
BEGIN SEARCH OPTIONS [<string>name]
  # search algorithms
  GLOBAL SEARCH INCREMENT = <integer>num_steps(1)
  GLOBAL SEARCH ONCE = <string>ON|OFF(OFF)
  #
  # search tolerances
  SEARCH TOLERANCE = <string>AUTOMATIC|USER_DEFINED
    (AUTOMATIC)
  NORMAL TOLERANCE = <real>norm_tol
  TANGENTIAL TOLERANCE = <real>tang_tol
  #
  # secondary decomposition
  SECONDARY DECOMPOSITION = <string>ON|OFF(OFF)
END [SEARCH OPTIONS <string>name]
```

For this section, the command usage conforms to the 2.7 release of Presto.

Contact involves a search phase and an enforcement phase. The contact search algorithm used to detect interactions between contact surfaces is often the most computationally expensive part of an analysis. The user can exert some control over how the search phase is carried out via the `SEARCH OPTIONS` command block. By selecting different options in this command block, the user can make trade-offs between the accuracy of the search and computing time.

The most accurate approach to the search phase is a global search at every time step. For a global search, a box is drawn around each face. The box depends on the shape of the face, the location of the face in space, and search tolerances. Now suppose we want to determine whether some node has penetrated that face. We must first determine if the node lies in one or more boxes that surround a face. This search, although done with an optimal algorithm, is still time consuming. The search must be done for all nodes that may be in contact with a face. A less accurate approach for the search phase is to use what is called a local tracking algorithm. For the tracking algorithm approach, we first do a global search. When a node has contacted a face in the global search, we record the face (or faces) contacted by the node. Instead of using the global search on subsequent time steps, we simply rely on the record of the node-face interactions to compute the contact forces. The last face contacted by a node in the global search is assumed to remain in contact with that node for subsequent time steps. In actuality, the node may slide off the face it was contacting at the time of the global search. In this case, faces that share an edge with the original contact face are searched to determine whether they (the edge adjacent faces) are in contact with the node. If the node moves across a corner of the face (rather than an edge), we may lose the contact interaction for the node until the next global search. If we lose the contact interaction, we lose some of the accuracy in the contact calculations until we do the next global search. Furthermore, it is possible that additional nodes may actually come into contact in the time steps between global searches. These nodes are typically caught during the next global search, but inaccuracies can result from missing the exact time of contact. The tracking algorithm, under certain circumstances, can work quite well even though it is less accurate. We can encounter analyses where we can set the number of intervals (time steps) between global searches to a relatively small number (5) and lose only a few or none of the node-to-face contacts between global searches. Likewise, we can encounter analyses where we can set the interval between global searches to a large number (100 or more) and lose only a few or none of the node-to-face contacts between global searches. Finally, we can encounter problems where we may only have to do one global search at the beginning and rely solely on the tracking information for the rest of the problem (without losing any contact). What search approach is best for your problem depends on the geometry of your structure, the loads on your structure, and the amount of deformation of your structure. This section tells you how to control the search phase for your specific problem.

The `SEARCH OPTIONS` command block begins with the input line

```
BEGIN SEARCH OPTIONS [<string>name]
```

and ends with

```
END [SEARCH OPTIONS <string>name] .
```

The `name` for the command block is optional.

Without a `SEARCH OPTIONS` command block, the default search with associated default search parameters is used for all contact pairs. If you want to override the default search method for all contact pairs, you should add a `SEARCH OPTIONS` command block. By adding a `SEARCH OPTIONS` command block, you establish a new set of global defaults for the search for all contact pairs. The default for the search is that tracking is turned on and the number of intervals (time steps) between a global search is one (`GLOBAL SEARCH INCREMENT = 1` and `GLOBAL SEARCH ONCE = OFF`).

The valid command lines within a `SEARCH OPTIONS` command block are described in Section 7.12.1, Section 7.12.2, and Section 7.12.3. The values specified by these commands are applied by default to all interaction contact surfaces, unless overridden by a specific interaction definition.

7.12.1 Search Algorithms

```
GLOBAL SEARCH INCREMENT = <integer>num_steps(1)
GLOBAL SEARCH ONCE = <string>ON|OFF(OFF)
```

The above two command lines let you determine the frequency of the global search. Although these command lines are mutually exclusive, they provide for three search options:

1. If you want to do only one global search and have all subsequent searches be tracking searches, then you should use the `GLOBAL SEARCH ONCE` command line with the string parameter set to `ON`. By default, the `GLOBAL SEARCH ONCE` option is `OFF`. If you set `GLOBAL SEARCH ONCE` to `ON`, then this should be the only command line for the search algorithms in the command block. The `GLOBAL SEARCH INCREMENT` command line should not be used.
2. If you want to use the global search only intermittently, with the tracking search in between the global search, you should use the `GLOBAL SEARCH INCREMENT` set to some integer value greater than 1. The integer value `num_steps` determines the number of time steps between global searches. The `GLOBAL SEARCH ONCE` command line should not be used.
3. If you want to do a global search at every time step, you should use the `GLOBAL SEARCH INCREMENT` command line with `num_steps` set to `1` or just simply omit this line since the default for the search increment is `1`. The `GLOBAL SEARCH ONCE` command line should not be used.

In summary, you have three options for the global search. You can do a global search only once (the first time step), and do a tracking search for all subsequent searches by setting `GLOBAL SEARCH ONCE` to `ON`. You can do a global search for the beginning time step and intermittently thereafter; the time steps between the global searches will use a tracking search. For this approach, you will need only the `GLOBAL SEARCH INCREMENT` command line. Finally, you can set `GLOBAL SEARCH INCREMENT` to `1` and do a global search at every time step.

7.12.2 Search Tolerances

```
SEARCH TOLERANCE = <string>AUTOMATIC|USER_DEFINED(AUTOMATIC)
NORMAL TOLERANCE = <real>norm_tol
TANGENTIAL TOLERANCE = <real>tang_tol
```

As indicated previously, the contact functionality in Presto uses a box defined around each face to locate nodes that may potentially contact the face. This box is defined by a tolerance normal to the face and another tolerance tangential to the face (see Figure 7.4). The code adds to these tolerances the maximum motion over a time step when identifying interactions. In the above command lines, the parameter `norm_tol` is the normal tolerance (defined on the `NORMAL TOLERANCE` command line) for the search box and the parameter `tang_tol` is the tangential tolerance (defined on the `TANGENTIAL TOLERANCE` command line) for the search box.

By default, Presto will automatically calculate normal and tangential tolerances. The default value for the command line `SEARCH TOLERANCE` is `AUTOMATIC`. If you leave automatic search on and also specify normal and/or tangential tolerances with the `NORMAL TOLERANCE` and `TANGENTIAL TOLERANCE` command lines, the larger of the two tolerances—automatic or user specified—will be used. For example, suppose you specify a normal tolerance of 1.0×10^{-3} 1.0×10^{-3} and the automatic tolerancing computes a normal tolerance of 1.05×10^{-3}. Then Presto will use a normal tolerance of 1.05×10^{-3}.

When the `USER_DEFINED` option is specified for the `SEARCH TOLERANCE` command line, these normal and tangential tolerances must be specified. If these tolerances are not specified, code execution will be terminated with an error.

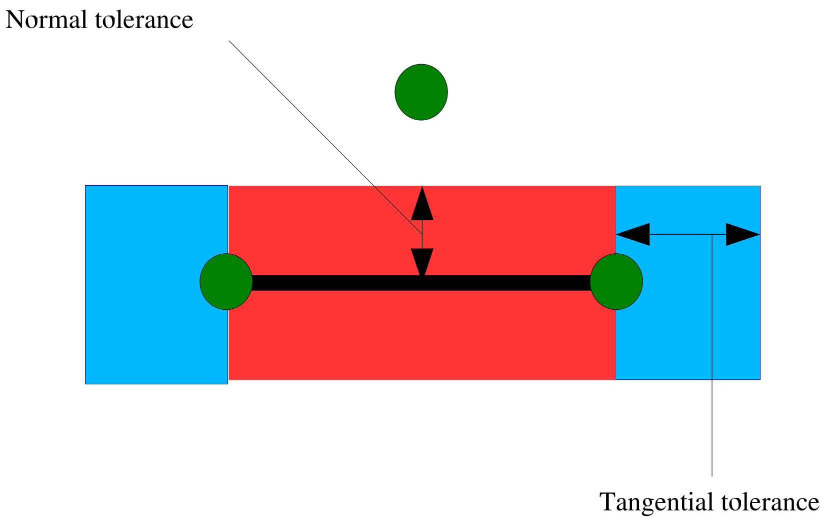


Figure 7.4: Illustration of normal and tangential tolerances.

Both of these tolerances are absolute distances in the same units as the analysis. The proper tolerances are problem dependent. If a normal or tangential tolerance is specified in the `SEARCH OPTIONS` command block, they apply to all interactions. These default search tolerances can be overwritten for a specific interaction by specifying a value for the normal tolerance and/or tangential tolerance for that interaction inside the `INTERACTION` command block (see Section 7.15).

7.12.3 Secondary Decomposition

```
SECONDARY DECOMPOSITION = <string>ON|OFF(ON)
```

The `SECONDARY DECOMPOSITION` command line controls internal options used by the ACME contact search algorithm. Computational results for secondary decomposition `ON` should be identical to those for secondary decomposition `OFF`. However, the computational time for these two distinct options may vary significantly.

When a mesh is divided for parallel processing, it is usually divided such that each processor has the same number of elements. The element-based load balance needs to achieve good parallel performance for element and material calculations. It is possible to have the number of elements per processor balanced but the number of contact faces per processor highly unbalanced. If contact is highly localized in one region of the model, it may happen that a small subset of the processors contains most of the contact interactions. A secondary decomposition is a parallel decomposition that balances the number of contact faces. When secondary decomposition is on, the contact algorithm first moves all data to the secondary decomposition and then it runs the contact calculations. When the secondary decomposition is off, all contact calculations are done in the primary decomposition.

The computational effort to move data to the secondary decomposition can be quite large. Thus, if the contact surfaces are well balanced in the primary decomposition, a large cost savings can be realized by turning off the secondary decomposition. Three conditions must be met for turning off the secondary decomposition to achieve cost savings. First, the number of contact faces per processor must be somewhat balanced in the primary decomposition. Second, faces in contact should be on the same processor as much as possible. Inertial and RCB decomposition tend to meet this condition of having contact faces in proximity on the same processor, while Multi-KL does not. Third, conditions one and two must persist throughout the entire analysis. An initially well balanced, well distributed mesh may become poorly balanced through element death or large deformations.

The best recommendation is to leave the secondary decomposition on.

7.13 Enforcement Options

```
BEGIN ENFORCEMENT OPTIONS [<string>name]
  ENFORCEMENT ALGORITHM = <string>MOMENTUM_BALANCE|
    PENALTY(MOMENTUM_BALANCE)
  MOMENTUM BALANCE ITERATIONS = <integer>num_iterations(5)
END [ENFORCEMENT OPTIONS <string>name]
```

Contact, as previously indicated, involves a search phase and an enforcement phase. The user can exert some control over how the enforcement phase is carried out via the `ENFORCEMENT OPTIONS` command block. By selecting different options in this command block, the user can make trade-offs between solution accuracy and computing time. The `ENFORCEMENT OPTIONS` command block begins with the input line

```
BEGIN ENFORCEMENT OPTIONS [<string>name]
```

and is terminated with the input line

```
END [ENFORCEMENT OPTIONS <string>name] .
```

The name for the command block, `name`, is optional

Only one `ENFORCEMENT OPTIONS` command block is permitted in a `CONTACT DEFINITION` command block. Without an `ENFORCEMENT OPTIONS` command block, the default enforcement algorithm with associated default enforcement options is used for all contact pairs. If you want to override the defaults for enforcement for all contact pairs, you should add an `ENFORCEMENT OPTIONS` command block. By adding this command block, you establish a new set of global defaults for enforcement for all contact pairs. You can override some of these global defaults for enforcement for a contact pair by inserting certain command lines in the `INTERACTION` command block (see Section 7.15) for that contact pair. It is possible, therefore to tailor the enforcement approach for individual contact pairs.

Currently, the enforcement option is of limited use. Options for user control will be expanded in future versions of Presto.

The `ENFORCEMENT ALGORITHM` command line lets the user select either a momentum balance enforcement algorithm (`MOMENTUM_BALANCE`) or a penalty method enforcement algorithm (`PENALTY`). The default value is `MOMENTUM_BALANCE`.

For the current release of Presto, users should rely on the default enforcement algorithm, the momentum balance approach. Consult with a Presto developer if you would like to use the penalty method.

The momentum balance algorithm for enforcement of the contact constraints uses an iterative process to ensure incremental momentum balance over a time step. Rather than making one pass to compute contact forces for node push-back, several passes are made to more accurately compute the normal contact force and, subsequently, the tangential (frictional) contact forces. The number of passes (iterations) is set by the value `num_iterations` in the `MOMENTUM BALANCE ITERATIONS` command line. The default value for the number of iterations is 5. This value is generally acceptable for removing overlap in the mesh. To get accurate results in a global sense in analyses that use friction, a value of 10 is more appropriate. To get accurate contact response at individual points in analyses with friction, a value of 20 or greater may be needed. Note that as the number of iterations increases, the expense of enforcement increases. Thus a user can balance execution speed and accuracy with this command line, though care must be taken to ensure that the appropriate level of accuracy is attained. This command line affects only the enforcement phase of the contact. A single search phase is used for contact detection, but the enforcement phase uses an iterative process.

7.14 Default Values for Interactions

```
BEGIN INTERACTION DEFAULTS [<string>name]
  SURFACES = <string list>surface_names
  SELF CONTACT = <string>ON|OFF(OFF)
  GENERAL CONTACT = <string>ON|OFF(OFF)
  AUTOMATIC KINEMATIC PARTITION = <string>ON|OFF(OFF)
  FRICTION MODEL = <string>friction_model_name|FRICTIONLESS
    (FRICTIONLESS)
  INTERACTION BEHAVIOR = <string>SLIDING|
    INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
END [INTERACTION DEFAULTS <string>name]
```

For this section, the command usage conforms to the 2.7 release of Presto.

This section discusses the `INTERACTION DEFAULTS` command block. Note that the name for the `INTERACTION DEFAULTS` command block, `name`, is currently not used or required. This command block lets you enforce contact on a subset of all contact surfaces or on all contact surfaces. You may overwrite predefined values defining surface interaction (for all surfaces defined by this command block) by using several different command lines.

It is important to note that unless some combination of the `INTERACTION DEFAULTS` command block and `INTERACTION` command blocks (Section 7.15) appears in your `CONTACT DEFINITION` command block, enforcement will not take place. Up to this point, all command lines and command blocks have provided information to set up the search phase and have provided details for surface interaction. However, contact enforcement for surfaces—the actual removal of interpenetration between surfaces and the calculation of surface forces consistent with friction models—will not take place unless some combination of the `INTERACTION DEFAULTS` command block and `INTERACTION` command blocks is used to set up surface interactions.

Contact between surfaces requires data to describe the interaction between these surfaces. You may specify defaults for the surface interactions for some or all surface pairs by using the `INTERACTION DEFAULTS` command block. Within this command block, you can provide a list of surfaces that are a subset of the contact surfaces. Any pair of surfaces listed in the `INTERACTION DEFAULTS` command block will acquire the default values that are defined within the `INTERACTION DEFAULTS` command block. If you omit the `SURFACES` command line, defaults in the `INTERACTION DEFAULTS` command block are applied to all surfaces. Any default set within an `INTERACTION DEFAULTS` command block can be overridden by commands in an `INTERACTION` command block. See Section 7.15.

If you consider only the use of the `INTERACTION DEFAULTS` command block (and not the use of the `INTERACTION` command block), you have three options for the surface interaction values:

- You can specify default surface interaction values for all the contact surface pairs by specifying all the contact surfaces in an `INTERACTION DEFAULTS` command block.
- You can specify default surface interaction values for some of the contact surface pairs by specifying a subset of the contact surfaces in an `INTERACTION DEFAULTS` command block.
- You can leave all interactions off by default by not specifying an `INTERACTION DEFAULTS` command block.

You can overwrite surface interaction values that you have set with an `INTERACTION DEFAULTS` command block by using an `INTERACTION` command block.

The valid commands within an `INTERACTION DEFAULTS` command block are described in Section 7.14.1 through Section 7.14.5. The values specified by the command lines in the `INTERACTION DEFAULTS` command block are applied by default to all interaction contact surfaces unless overridden by a specific interaction definition.

7.14.1 Surface Identification

```
SURFACES = <string list>surface_names
```

This command line identifies the contact surfaces to which the surface interaction values defined in the `INTERACTION DEFAULTS` command block will apply. The string list on the `SURFACES` command line specifies the names of these contact surfaces. The `SURFACES` command line can include any surface specified in a `CONTACT SURFACE` command line, a `CONTACT SURFACE` command block, or a `SKIN ALL BLOCKS` command line.

The `SURFACES` command line is optional. If you want the defaults to apply to all the surfaces you have defined, you will NOT use the `SURFACES` command line in this command block. If you want the defaults to apply to a subset of all contact surfaces, then you will list the specific set of surfaces on a `SURFACES` command line. The names of all the surfaces with the default values will be listed in the string list designated as `surface_names`.

7.14.2 Self-Contact and General Contact

```
SELF CONTACT = <string>ON|OFF(OFF)
GENERAL CONTACT = <string>ON|OFF(OFF)
```

The `SELF CONTACT` command line, if set to `ON`, specifies that the default values set in the command lines of the command block will apply to self-contact between the listed surfaces (or all surfaces if no surfaces are listed). The `GENERAL CONTACT` command line, if set to `ON`, specifies that the default values set in the command lines of this command block apply to contact between the listed surfaces (or all surfaces if no surfaces are listed) excluding self-contact. The default values for both of these command lines is `OFF`. If you want to enforce general contact between all surfaces specified in the `INTERACTION DEFAULTS` command block but no self-contact, you must include the line

```
    GENERAL CONTACT = ON .
```

If you want to enforce self-contact for all surfaces specified in the `INTERACTION DEFAULTS` command block, you must include the line

```
    SELF CONTACT = ON .
```

Suppose that you have only an `INTERACTION DEFAULTS` command block with no `INTERACTION` command block in your `CONTACT DEFINITION` command block. Unless you have a `GENERAL CONTACT` command line set to `ON`, a `SELF CONTACT` command line set to `ON`, or both the `GENERAL CONTACT` command line set to `ON` and the `SELF CONTACT` command line set to `ON`, no enforcement will occur.

Suppose you have turned on contact enforcement for all contact surface pairs (general contact and self-contact) in the `INTERACTION DEFAULTS` command block. You may turn off contact enforcement for a specific contact pair by use of the `INTERACTION BEHAVIOR` command line in the `INTERACTION` command block. (The same holds true if you have turned on contact enforcement for only a subset of contact surface pairs in the `INTERACTION DEFAULTS` command block.)

Suppose you have turned on self-contact enforcement for all contact surfaces in the `INTERACTION DEFAULTS` command block. You may override self-enforcement for a specific contact surface by use of the `INTERACTION BEHAVIOR` command line in the `INTERACTION` command block. (The same holds true if you have turned on contact enforcement for only a subset of contact surfaces in the `INTERACTION DEFAULTS` command block.)

7.14.3 Friction Model

```
    FRICTION MODEL = <string>friction_model_name|FRICTIONLESS
      (FRICTIONLESS)
```

The `FRICTION MODEL` command line permits the description of how surfaces interact with each other using a friction model defined in a friction-model command block (see Section 7.11). In the above command line, the string `friction_model_name` should match the name assigned to some friction model command block. For example, if you specified the name of an `AREA WELD` command block as `AW1` and wanted to reference that name in the `FRICTION MODEL` command line, the value of `friction_model_name` would be `AW1`.

The default interaction is frictionless contact.

7.14.4 Automatic Kinematic Partition

```
AUTOMATIC KINEMATIC PARTITION
```

If the `AUTOMATIC KINEMATIC PARTITION` command line is used, Presto will automatically compute the kinematic partition factors for pairs of surfaces. (See Section 7.15.2 for more information on kinematic partitioning.) The automatic kinematic partitions are computed from the impedance of each surface based on nodal average density and wave speed. Automatic computation of kinematic partition factors provides the best approach to exact enforcement of symmetric contact of opposing surfaces provided that these surfaces have the same mesh resolution. If the mesh resolution is disparate, you can specify the coarser meshed body as master, but generally it is better to use more contact iterations to deal with this case of a fine mesh contacting a coarse mesh.

For the interaction of any two surfaces, the sum of the partition factors for the surfaces must be 1.0. This is automatically taken care of when the `AUTOMATIC KINEMATIC PARTITION` command line is used. The default value for kinematic partition factors for all surfaces is 0.5.

The `AUTOMATIC KINEMATIC PARTITION` command line can be used to set the kinematic partitions for all interactions or to set the kinematic partitions for specific interactions. Thus the command line can appear in two different scopes:

1. The command line can be used within the `INTERACTION DEFAULTS` command block. In this case, all contact surface interactions defined in the command block will use the automatic kinematic partitioning scheme by default. This will override the default case that assigns a kinematic partition factor of 0.5 to all surfaces. For particular interactions, it is possible to override the use of the automatic kinematic partition factors by specifying kinematic partition values (with the `KINEMATIC PARTITION` command line) within the `INTERACTION` command blocks for those interactions.

2. The command line can be used inside an `INTERACTION` command block. If the automatic partitioning command line appears inside an `INTERACTION` command block, the kinematic partition factors for that particular interaction will be calculated by the automatic kinematic partition scheme.

The AUTOMATIC KINEMATIC PARTITION command line is not currently operational for shell elements. An error message is reported if this option is used with shell elements.

7.14.5 Interaction Behavior

```
INTERACTION BEHAVIOR = <string>SLIDING|
  INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
```

The INTERACTION BEHAVIOR command line specifies how the search will be done. For SLIDING contact, the search algorithm is constantly updating information that lets the code accurately track the sliding of the node over the face and any adjacent faces. The SLIDING option, which is the default, lets us handle the case where we have large relative sliding between a face and a node. A node contacting a face can slide over time by a significant amount over the face. The node can slide onto an adjacent face or onto a face on a nearby surface. For the case of INFINITESIMAL_SLIDING, search information is not updated to the extent that it is with the SLIDING option. In the case of INFINITESIMAL_SLIDING, it is assumed that there is very little slip over time of a node relative to its initial contact point on a face. Furthermore, it is assumed a node will not slide off the face that it initially contacts. The INFINITESIMAL_SLIDING option is not as accurate as the SLIDING option, but neither is it as expensive as the SLIDING option. For some cases, however, the INFINITESIMAL_SLIDING option may work quite well even though it is not as accurate as the SLIDING option. Finally, you may turn off the search completely by using the NO_INTERACTION option.

With the third option, NO_INTERACTION, you could turn off the search for all surfaces specified in the INTERACTION DEFAULTS command block. You could then turn on the search on a case-by-case basis for various contact pairs or for the self-contact of surfaces by using INTERACTION command blocks. This is a convenient way to set defaults for the friction model and automatic kinematic partitioning without turning on all the interactions. More likely, you will set contact interactions to default to the SLIDING option in the INTERACTION DEFAULTS command block, and then turn off specific contact interactions through INTERACTION command blocks.

Using the INTERACTION BEHAVIOR command line in the INTERACTION DEFAULTS command block represents a sophisticated application of this command line. Please consult with Presto developers for more information about this command line if it is used in an INTERACTION DEFAULTS command block.

7.15 Values for Specific Interactions

```
 BEGIN INTERACTION [<string>name]
   SURFACES = <string>surface1 <string>surface2
   MASTER = <string>surface
   SLAVE = <string>surface
   KINEMATIC PARTITION = <real>kin_part
   NORMAL TOLERANCE = <real>norm_tol
   TANGENTIAL TOLERANCE = <real>tang_tol
   OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
   OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
   FRICTION MODEL = <string>friction_model_name|FRICTIONLESS
     (FRICTIONLESS)
   AUTOMATIC KINEMATIC PARTITION
   INTERACTION BEHAVIOR = <string>SLIDING|
     INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
END [INTERACTION <string>name]
```

The Presto contact input also permits the setting of values for specific interactions using the `INTERACTION` command block. If an `INTERACTION DEFAULTS` command block is present within a `CONTACT DEFINITION` command block, the values provided by an `INTERACTION` command block override the defined defaults. If an `INTERACTION DEFAULTS` command block is not present, only those interactions described by `INTERACTION` command blocks are searched for contact, and values without system defaults must be specified.

The `INTERACTION` command block begins with

```
BEGIN INTERACTION [<string>name]
```

and ends with

```
END [INTERACTION <string>name] ,
```

where `name` is a name for the interaction. Note that this name is currently not used or required.

The valid commands within an `INTERACTION` command block are described in Section 7.15.1 through Section 7.15.6.

7.15.1 Surface Identification

```
SURFACES = <string>surface1 <string>surface2

MASTER = <string>surface
SLAVE = <string>surface
```

There are two methods to identify the surfaces described by a specific interaction. The standard method is to identify both surfaces in a single line with the `SURFACES` command line, where `surface1` and `surface2` are the names of the two contact surfaces to which the interaction refers. In this syntax, the values supplied for the interaction are defined for two-way contact, where contact is evaluated twice: once with the first surface as master and the second as slave, and once with the opposite arrangement. How the two contact enforcements are combined is defined with the `KINEMATIC PARTITION` command line (see Section 7.15.2). Two-way contact provides better-quality contact results for most problems.

To specify one-way contact, where the nodes of the "slave" surfaces are searched against the "master" surface, use the `MASTER` and `SLAVE` command lines, where `surface` is the name of a contact surface defined in the `CONTACT DEFINITION` command block (see Section 7.2). In this case, all interaction values specified in this command block are applied only to nodes of the slave surface interacting with faces of the master surface. You cannot specify two master-slave interactions in which the contact surfaces are switched. You also cannot specify a kinematic partition for a master-slave interaction; it is assumed to be 1.0.

In the `INTERACTION` command block, either the `SURFACES` syntax (`SURFACES` command line only) or the `MASTER`/`SLAVE` syntax (`MASTER` and `SLAVE` command lines) should be used, but not both.

For self-contact, use the `SURFACES` command lines. The two surfaces given in the `SURFACES` command line are the same contact surface.

7.15.2 Kinematic Partition

```
KINEMATIC PARTITION = <real>kin_part
```

To provide accurate contact evaluation, Presto typically computes two-way contact between two surfaces, where interactions are defined between the nodes of the first surface and the faces of the second surface, and also between the nodes of the second surface and the faces of the first surface. If the two surfaces have penetrated each other by a distance δ, then each of the contact evaluations will compute forces to move the surface a distance δ, so that the total resulting displacement would be 2δ if both sets of contact computations were fully applied. The `KINEMATIC PARTITION` command line defines a kinematic partition scaling factor, `kin_part`, for the two contact computations so that the total contact motion is correct. The kinematic partition factor, `kin_part`, is a value between 0.0 and 1.0. The factor scales the relative motion of the first surface, where `kin_part` $= 0.0$ means the first surface will move none of δ, while `kin_part` $= 1.0$ means the surface moves all of δ. The second surface moves the portion of δ that remains after the motion of the first surface (i.e., δ for second surface = 1.0 – `kin_part`). For instance, if `kin_part` is 0.2, the first surface will move 20% of the penetration distance (0.2δ in our example), and the second surface would move the remaining 80% (0.8δ in our example). The default value is 0.5, so that each surface would move half of the penetration distance. If `kin_part` is 0.0, the first surface does not move at all, and the second surface moves the full distance. This is exactly equivalent to a one-way master-slave contact definition, where the first surface is the master and the second is the slave. If `kin_part` is 1.0, the second surface is the master, and the first surface is the slave. Figure 7.5 illustrates how the kinematic partition factor varies from 0.0 to 1.0, with the specific example of `kin_part` being set to 0.2.

```
begin interaction
  surfaces = <first_surface> <second_surface>
  kinematic partition = 0.2
end
```

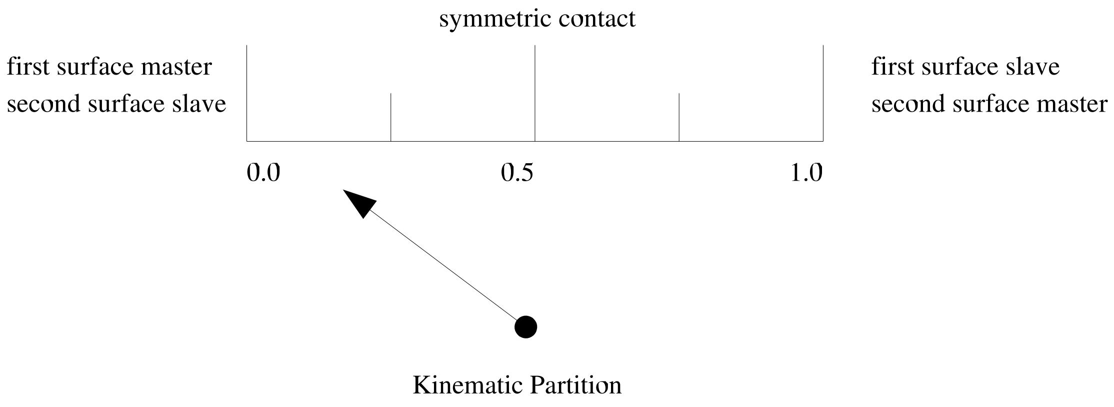


Figure 7.5: Illustration of kinematic partition values.

The capability provided by the `KINEMATIC PARTITION` command line is important in cases where contact occurs between two materials of disparate stiffness. Physically, we would expect a material with a higher stiffness to have more of an effect in determining the position of the contact surface than a more compliant material. In this case, we want the softer material to move more of the distance, and thus it should have a higher kinematic partition factor. The appropriate kinematic partition factor can be determined in closed form; see the ACME contact library reference [1] for more information. Alternately, the `AUTOMATIC KINEMATIC PARTITION` capability can automatically calculate the proper kinematic partition based on the stiffness of the materials.

Another case where the kinematic partition factor has traditionally been used is when meshes with dissimilar resolutions contact each other. If an interaction is defined with a fine mesh as the master surface and a coarse mesh as a slave surface, the contact algorithms will permit nodes on the master surface to penetrate the slave surface. In these cases, such problems can be alleviated by making the coarse mesh the master surface. However, the iterative approach implemented in the enforcement can also take care of this problem and is advised. If very few iterations are chosen, an appropriate kinematic partition factor may be needed to prevent unintentional penetration due to mesh discretization.

For self-contact, the kinematic partition factor should be 0.5.

A kinematic partition factor cannot be defined for interactions that use the pure master-slave syntax (see Section 7.15.1).

In general, it is best to use the automatic kinematic partition option to properly compute the kinematic partition for a pair of surfaces. However, in a few cases, master-slave interactions are preferred. These cases consist of (1) interaction between an analytic surface and a deformable body, where the analytic body should be the master surface; and (2) contact between shells and solids, where the solid should be the master surface.

7.15.3 Tolerances

```
NORMAL TOLERANCE = <real>norm_tol
TANGENTIAL TOLERANCE = <real>tang_tol
OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
```

You can set tolerances for the interaction for a specific contact surface pair or for self-contact of a surface by using the above tolerance-related command lines in a `INTERACTION` command block. See Section 7.12.2 on search tolerances and Section 7.5 on overlap tolerances for a complete discussion of tolerances for contact.

7.15.4 Friction Model

```
FRICTION MODEL = <string>friction_model_name|FRICTIONLESS
  (FRICTIONLESS)
```

You can set the friction model for the interaction for a specific contact surface pair or for self-contact of a surface by using the above command line in an `INTERACTION` command block. See Section 7.14.3 for a discussion of this command line.

7.15.5 Automatic Kinematic Partition

```
AUTOMATIC KINEMATIC PARTITION
```

You can turn on (or off) automatic kinematic partitioning for a specific contact surface pair by using the above command line in an `INTERACTION` command block. See Section 7.14.4 for a discussion of automatic kinematic partitioning.

7.15.6 Interaction Behavior

```
INTERACTION BEHAVIOR = <string>SLIDING|
  INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
```

You can set the search behavior for a specific contact surface pair or for self-contact of a surface by using the above command line in an `INTERACTION` command block. See Section 7.14.5 for a discussion of this command line.

A particular use of this command line in this particular command block is to set the interaction behavior to `NO_INTERACTION`. This deactivates enforcement between the surfaces specified in the `INTERACTION` command block.

7.16 Examples

This section has several example problems. We present the geometric configuration for the problems and the appropriate command lines to describe contact for the problems.

7.16.1 Example 1

Our first example problem has two blocks that come into contact due to initial velocity conditions. Block 1 has an initial velocity equal to $v1$, and block 2 has an initial velocity equal to $v2$. The geometric configuration for this problem is shown in Figure 7.6.

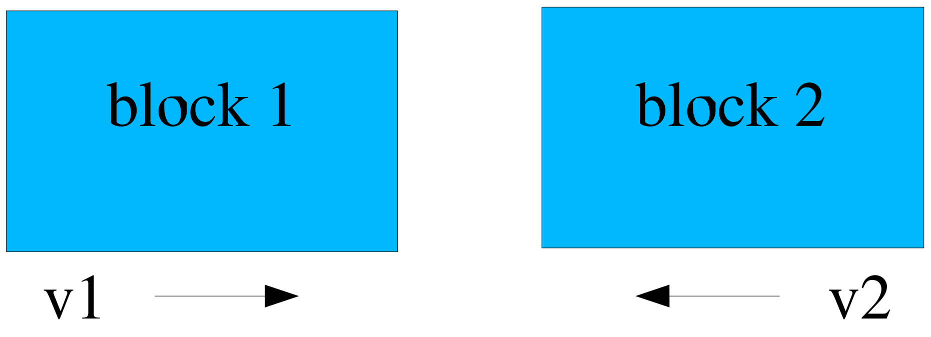


Figure 7.6: Problem with two blocks coming into contact.

The simplest input for this problem will be named `EXAMPLE1` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE1

  # define contact surfaces
  SKIN ALL BLOCKS = ON

  # set interactions
  BEGIN INTERACTION DEFAULTS
    GENERAL CONTACT = ON
  END INTERACTION DEFAULTS

END CONTACT DEFINITION EXAMPLE1
```

In our example, the `SKIN ALL BLOCKS` command line with its parameter set to

ON will create a surface named surface_1 (from the skinning of block_1 and a surface named surface_2 (from the skinning of block_2).

All the normal and tangential tolerances will be set automatically in the above example. Frictionless contact is assumed. The kinematic partition factor defaults to 0.5 for both surfaces, surface_1 and surface_2.

If you omitted the INTERACTION DEFAULTS command block with GENERAL CONTACT set to ON, then contact enforcement would not take place.

Now, let us consider the same problem (two blocks coming into contact) in which the contact definition for the problem is not defined simply by using all the default settings. The input for this variation of our two-block problem will be named EXAMPLE1A and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE1A

  # define contact surfaces
  SKIN ALL BLOCKS = ON

  # friction model
  BEGIN CONSTANT FRICTION MODEL ROUGH
    FRICTION COEFFICIENT = 0.5
  END CONSTANT FRICTION MODEL ROUGH

  # search options
  BEGIN SEARCH OPTIONS
    GLOBAL SEARCH INCREMENT = 10
    NORMAL TOLERANCE = 1.0E-3
    TANGENTIAL TOLERANCE = 1.0E-3
  END SEARCH OPTIONS

  # set interactions
  BEGIN INTERACTION DEFAULTS
    FRICTION MODEL = ROUGH
    GENERAL CONTACT = ON
  END INTERACTION DEFAULTS

END CONTACT DEFINITION EXAMPLE1A
```

As is the case of the EXAMPLE1 command block, the SKIN ALL BLOCKS command line with its parameter set to ON will create a surface named surface_1 (from the skinning of block_1) and a surface named surface_2 (from the skinning of block_2).

For EXAMPLE1A, we want to have frictional contact between the two blocks. For the frictional contact, we define a constant friction model with a `CONSTANT FRICTION MODEL` command block. We name this model `ROUGH`.

The `SEARCH OPTIONS` command block sets the interval between global searches to 10; the default value is 5. Also, in this command block, we have set values for the normal and tangential tolerances. The option to compute the search tolerance automatically has been left on. The larger of the two values—an automatically computed tolerance or the user-specified tolerance—will be selected as the search tolerance during the search phase.

In the `INTERACTION DEFAULTS` command block, we select the friction model `ROUGH` on the `FRICTION MODEL` command line. As in the case of the `EXAMPLE1` command block, if you omitted the `INTERACTION DEFAULTS` command block with `GENERAL CONTACT` set to `ON`, contact enforcement would not take place.

7.16.2 Example 2

Our second example problem has three blocks that come into contact due to initial velocity conditions. Block 1 has an initial velocity equal to $v1$, and block 3 has an initial velocity equal to $v3$. The geometric configuration for this problem is shown in Figure 7.7.

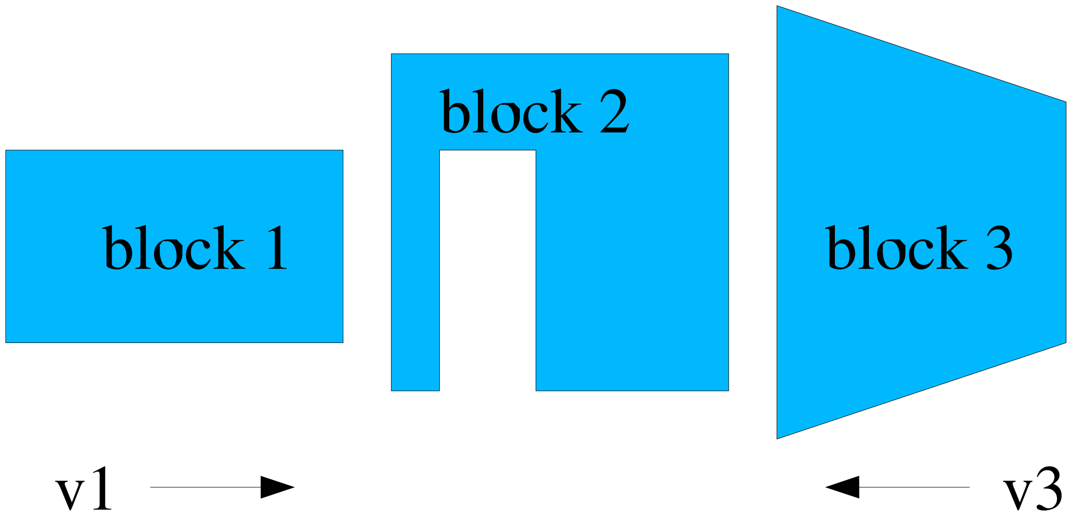


Figure 7.7: Problem with three blocks coming into contact.

The input for this three-block problem will be named `EXAMPLE2` and is shown as follows:

```
    BEGIN CONTACT DEFINITION EXAMPLE2
    # define contact surfaces
    CONTACT SURFACE surface_1 CONTAINS block_1
    CONTACT SURFACE surface_2 CONTAINS block_2
    CONTACT SURFACE surf_3 CONTAINS surface_3

    # friction model
    BEGIN CONSTANT FRICTION MODEL ROUGH
      FRICTION COEFFICIENT = 0.5
    END CONSTANT FRICTION MODEL ROUGH

    # search options
    BEGIN SEARCH OPTIONS
      GLOBAL SEARCH INCREMENT = 10
      NORMAL TOLERANCE = 1.0E-3
      TANGENTIAL TOLERANCE = 1.0E-3
    END SEARCH OPTIONS

    # set interactions
    BEGIN INTERACTION DEFAULTS
      FRICTION MODEL = ROUGH
      GENERAL CONTACT = ON
      SELF CONTACT = ON
    END INTERACTION DEFAULTS

    # set specific interaction
    BEGIN INTERACTION S2TOS3
      SURFACES = surface_2 surf_3
      KINEMATIC PARTITION = 0.4
      NORMAL TOLERANCE = 0.5E-3
      TANGENTIAL TOLERANCE = 0.5E-3
      FRICTION MODEL = FRICTIONLESS
    END INTERACTION S2TOS3

  END CONTACT DEFINITION EXAMPLE2
```

For the `EXAMPLE2` command block, we have defined three surfaces. The first surface, `surface_1`, is obtained by skinning `block_1`. The second surface, `surface_2` is obtained by skinning `block_2`. The third surface, `surf_3`, is the user-defined surface `surface_3`. The user-defined surface, `surface_3`, can contain a subset of the external element faces that define `block_3` or all the external element faces that define `block_3`.

The `SEARCH OPTIONS` command block sets the interval between global searches to 10; the default value is 5. Also, in this command block, we have set values for the normal and tangential tolerances. The option to compute the search tolerance automatically has been left on. The larger of the two values—an automatically computed tolerance or the user-specified tolerance—will be selected as the search tolerance during the search phase.

In the `INTERACTION DEFAULTS` command block, we select the friction model `ROUGH` on the `FRICTION MODEL` command line. Both `GENERAL CONTACT` and `SELF CONTACT` are set to `ON` in the `EXAMPLE2` command block. For this problem, `block_2` can undergo self-contact. Setting `GENERAL CONTACT` to `ON` will enforce contact between `surface_1` and `surface_2`, `surface_2` and `surf_3`, and `surface_1` and `surf_3`. Setting `SELF CONTACT` to `ON` will enforce self-contact for all three of the surfaces.

For this particular example, we want to override some of the Presto default values for surface interaction and some of the default values for surface interaction set by the `INTERACTION DEFAULTS` command block for the interaction between `surface_2` and `surf_3`. To override default values, we use an `INTERACTION` command block and indicate that it applies to `surface_2` and `surf_3` with a `SURFACES` command line. We override the Presto default for the kinematic partition factor by using a `KINEMATIC PARTITION` command line with the kinematic partition parameter set to a value of 0.4. We override the normal and tangential tolerances and the friction model set in the `INTERACTION` command block. The normal and tangential tolerances for interaction between `surface_2` and `surf_3` is set to `0.5E-3` rather than the global value of `1.0E-3`. The friction model for interaction between `surface_2` and `surf_3` is set to `FRICTIONLESS` rather than the default value of `ROUGH`.

7.17 References

1. Brown, K. H., R. M. Summers, M. W. Glass, A. S. Gullerud, M. W. Heinstein, and R. E. Jones. *ACME: Algorithms for Contact in a Multiphysics Environment*, API Version, 2.2, SAND2004-5486. Albuquerque, NM: Sandia National Laboratories, October 2001.

2. Heinstein, M. W., and T. E. Voth. *Contact Enforcement for Explicit Transient Dynamics*, Draft SAND report. Albuquerque, NM: Sandia National Laboratories, 2005.

Chapter 8

Output

Presto produces a variety of output. This chapter discusses how to control the three major types of output: results output, history output, and restart output. Results output lets the user select some set of variables (registered, user-defined, or some combination thereof). If the user selects a nodal variable such as displacement for results output, the displacements for all the nodes in a model will be output to a results file. If the user selects an element variable such as rotated stress for results output, the rotated stress for all elements in the model that calculate this quantity (rotated stress) will be output. The history output option lets the user select a very specific set of information for output. For example, if you know that the displacement at a particular node is critical, then you can select only the displacement at that particular node as history output. The restart output is written so that any calculation with Presto can be halted at some arbitrary analysis time and then restarted at this time. The user has no control over what is written to the restart file. When a restart file is written, it must be a complete state description of the calculations at some given time. A restart file contains a great deal of information, and is typically much larger than a results file. You need to carefully limit how often a restart file is written.

Section 8.1 describes the results output. Included in the results output is a description of commands for user-defined output (Section 8.1.2). User-defined output lets the user postprocess analysis results as the code is running to produce a reduced set of output information. Section 8.2 describes the history output, and Section 8.3 describes the restart output. All three types of output (results, history, and restart) can be synchronized for analyses with multiple regions. This scheduling functionality is discussed in Section 8.4. In Section 8.5, there is a list of key registered variables.

Unless otherwise noted, the command blocks and command lines discussed in Chapter 8 appear in the region scope.

8.1 Results Output

The results output capability lets you select some set of variables that will be written to a file at various intervals. As previously indicated, all the values for each selected variable will be written to the results file. (The interval at which information is written can be changed throughout the analysis time.) The name of the results file is set in the `RESULTS OUTPUT` command block.

8.1.1 Exodus Results Output File

```
BEGIN RESULTS OUTPUT <string>results_name
  DATABASE NAME = <string>results_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  TITLE <string>user_title
  NODE VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODAL VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]
  NODESET VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODESET VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>nodelist_names
        ... <string>variable_name
        [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
        <string list>nodelist_names
  FACE VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | FACE VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>surface_names
        ...  <string>variable_name
        [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
        <string list>surface_names
  ELEMENT VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | ELEMENT VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>block_names
        ... <string>variable_name
        [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
```

```
          <string list>block_names
      COMPONENT SEPARATOR CHARACTER = <string>character|NONE
      GLOBAL VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]
      START TIME = <real>output_start_time
      TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
      AT TIME <real>time_begin INCREMENT =
        <real>time_increment_dt
      ADDITIONAL TIMES = <real>output_time1
        <real>output_time2 ...
      AT STEP <integer>step_begin INCREMENT =
        <integer>step_increment
      ADDITIONAL STEPS = <integer>output_step1
        <integer>output_step2 ...
      TERMINATION TIME = <real>termination_time_value
      USE OUTPUT SCHEDULER <string>scheduler name
      OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
        SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
        SIGKILL|SIGILL|SIGSEGV
    END [RESULTS OUTPUT <string>results_name]
```

You can specify a results file, the results to be included in this file, and the frequency at which results are written by using a `RESULTS OUTPUT` command block. The command block appears inside the region scope.

More than one results file can be specified for an analysis. Thus for each results file, there will be one `RESULTS OUTPUT` command block. The command block begins with

```
BEGIN RESULTS OUTPUT <string>results_name
```

and is terminated with

```
END [RESULTS OUTPUT <string>results_name] ,
```

where `results_name` is a user-selected name for the command block. Nested within the `RESULTS OUTPUT` command block are a set of command lines, as shown in the block summary given above. The first two command lines listed (`DATABASE NAME` and `DATABASE TYPE`) give pertinent information about the results file. The command line

```
DATABASE NAME = <string>results_file_name
```

gives the name of the results file with the string `results_file_name`. If the results file is to appear in the current directory and is named `job.e`, this command line would appear as

```
DATABASE NAME = job.e .
```

If the results file is to be created in some other directory, the command line would have to show the path to that directory.

If the results file does not use the Exodus II format, you must specify the format for the results file using the command line

```
DATABASE TYPE = <string>database_type(exodusII) .
```

Currently, both the Exodus II database and the XDMF database [1] are supported in Presto. Exodus II is more commonly used than XDMF. Other options may be added in the future.

The `OVERWRITE` command line can be used to prevent the overwriting of existing results files.

```
OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
  (ON|TRUE|YES)
```

The `OVERWRITE` command line allows only a single value. If you set the value to `FALSE`, `NO`, or `OFF`, the code will terminate before existing results files can be overwritten. If you set the value to `TRUE`, `YES`, or `ON`, then existing results files can be overwritten (the default status). Suppose, for example, that we have an existing results file named `job21.e`. Suppose also that we have an input file with a `RESULTS OUTPUT` command block that contains the `OVERWRITE` command line set to `ON` and the `DATABASE NAME` command line set to

```
DATABASE NAME = job21.e .
```

If you run the code under these conditions, the existing results file `job21.e` will be overwritten.

Whether or not results files are overwritten is also impacted by the use of the automatic read and write option for restart files described in Section 8.3.1.1. If you use the automatic read and write option for restart files, the results files, like the restart files, are automatically managed. The automatic read and write option in restart adds extensions to file names and prevents the overwriting of any existing restart or results files. For the case of a user-controlled read and write of restart files (Section 8.3.1.2) or of no restart, however, the `OVERWRITE` command line is useful for preventing the overwriting of results files.

You may add a title to the results file by using the `TITLE` command line. Whatever you specify for the `user_title` will be written to the results file. Some of the programs that process the results file (such as various SEACAS programs [2]) can read and display this information.

The other command lines that appear in the `RESULTS OUTPUT` command block determine the type and frequency of information that is output. Descriptions of these command lines follow in Section 8.1.1.1 through Section 8.1.1.15.

8.1.1.1 Output Nodal Variables

```
NODE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | NODAL VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name] ...
      <string>variable_name [AS
      <string>dbase_variable_name]
```

Any nodal variable in Presto can be selected for output in the results file by using a command line in one of the two forms shown above. The only difference between the two forms is the use of `NODE` or `NODAL`. The string `variable_name` is the name of the nodal variable. The string `variable_name` can be either a registered variable listed in Section 8.5 or a user-defined variable (see Section 8.1.2 and Section 9.2.4).

For the above two command lines, any nodal variable requested for output is output for ALL nodes.

It is possible to specify an alias for any of the nodal variables by using the `AS` specification. Suppose, for example, you wanted to output the external forces in Presto, which are registered as `force_external`, with the alias `f_ext`. You would then enter the command line

```
NODE VARIABLES = force_external AS f_ext .
```

In this example, the external force is a vector quantity. For a vector quantity at a node, suffixes are appended to the variable name (or alias name) to denote each vector component. The results database would have three variable names associated with the external force: `f_ext_x`, `f_ext_y`, and `f_ext_z`. Consult with Table 8.4 for a list of component identifiers for vectors. You can change the component separator, an underscore in this example, by using the `COMPONENT SEPARATOR CHARACTER` command line (see Section 8.1.1.5).

The `NODE VARIABLES` command line can be used any number of times within a `RESULTS OUTPUT` command block. It is also possible to specify more than one nodal variable for output on a command line, as indicated by the ellipsis in the command line format. In the following example, two nodal variables are specified for output. Note that the internal forces are registered as `force_internal`.

```
NODE VARIABLES = force_external force_internal
```

Aliases can be specified for each of the variables in a single command line. Thus, If you wanted to output the alias `f_ext` for external forces and also wanted to output the alias `f_int` for internal forces, you would enter the command line

```
NODE VARIABLES = force_external AS f_ext
  force_internal AS f_int .
```

The specification of an alias is always optional.

8.1.1.2 Output Node Set Variables

```
NODESET VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | NODESET VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>nodelist_names
      ... <string>variable_name
      [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>nodelist_names
```

A nodal variable may be registered only on a subset of the total set of nodes defining a model. A nodal variable that is registered only on some subset of nodes is referred to as a node set variable. The `NODESET VARIABLES` command line lets you specify a node set variable for output to the results file.

There are two forms of the `NODESET VARIABLES` command line. Either form will let you output a node set variable.

The first form of the command line is as follows:

```
NODESET VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
```

Here, the string `variable_name` is a node set variable associated with one or more node sets. In this form, the node set variable is output for all node sets associated with that node set variable.

It is possible to specify an alias in the results file for any of the node set variables by using the `AS` specification. Suppose, for example, you wanted to output a node set variable registered as `force_nsetype` with the alias `fnsetype`. You would then enter the command line

```
NODESET VARIABLES = force_nsetype AS fnsetype .
```

The `NODESET VARIABLES` command line can be used any number of times within a `RESULTS OUTPUT` command block. It is also possible to specify more than one node set variable for output on a command line, as indicated by the ellipsis in the command line format. In the following example, two node set variables are specified for output. Here, the second node set variable is registered as `force_nsetype2`.

```
NODESET VARIABLES = force_nsetype force_nsetype2
```

Aliases can be specified for each of the variables in a single command line. Thus, If you wanted to output the alias `fnsetype` for node set variable `force_nsetype` and also wanted to output the alias `fnsetype2` for node set variable `force_nsetype2`, you would enter the command line

```
NODESET VARIABLES = force_nsetype AS fnsetype
  force_nsetype2 AS fnsetype2 .
```

The specification of an alias is always optional.

The second form of the command line is as follows:

```
NODESET VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>nodelist_names
  ... <string>variable_name
  [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
  <string list>nodelist_names
```

This form of the `NODESET VARIABLES` command line is similar to the first, except that the user can control which node sets are used for output. The user can include a specific list of node sets for output by using the `INCLUDE` option or the `ON` option. (The key word `INCLUDE` is synonymous with the key word `ON`.) Alternatively, the user can exclude a specific list of node sets for output by using the `EXCLUDE` option.

Suppose that the node set variable `force_nsetype` from the above example has been registered for `nodelist_10`, `nodelist_11`, `nodelist_20`, and `nodelist_21`. If we only want to output the node set variable for node sets `nodelist_10`, `nodelist_11`, and `nodelist_21`, then we could specify the `NODESET VARIABLES` command line as follows:

```
NODESET VARIABLES = force_nsetype AS fnsetype
  INCLUDE nodelist_10, nodelist_11,
  nodelist_21
```

(In the above command line, the keyword `ON` could be substituted for `INCLUDE`.) Alternatively, we could use the command line

```
NODESET VARIABLES = force_nsetype AS fnsetype
  EXCLUDE nodelist_20 .
```

In the above command lines, an alias for a node set can be substituted for a node set identifier. For example, if `center_case` is an alias for `nodelist_10`, then the string `center_case` could be substituted for `nodelist_10` in the above command lines. Because a node set identifier is a mesh entity, the alias for the node set identifier would be defined via an `ALIAS` command line in a `FINITE ELEMENT MODEL` command block.

Note that the list of identifiers uses a comma to separate one node set identifier from the next node set identifier.

8.1.1.3 Output Face Variables

```
FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | FACE VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>surface_names
      ...  <string>variable_name
      [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>surface_names
```

A variable may be registered on some set of faces that constitute a surface. A variable registered on a set of faces is referred to as a face variable. The `FACE VARIABLES` command line lets you specify a face variable for output to the results file.

There are two forms of the `FACE VARIABLE` command line. Either form will let you output a face variable.

The first form of the command line is as follows:

```
FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
```

Here, the string `variable_name` is a face variable associated with one or more surfaces. In this form, the face variable is output for all surfaces associated with that face variable.

It is possible to specify an alias in the results file for any face variable by using the `AS` specification. Suppose, for example, you wanted to output a face variable registered as `pressure_face` with the alias `pressuref`. You would then enter the command line

```
FACE VARIABLES = pressure_face AS pressuref .
```

The `FACE VARIABLES` command line can be used any number of times within a `RESULTS OUTPUT` command block. It is also possible to specify more than one face variable for output on a command line, as indicated by the ellipsis in the command line format. In the following example, two face variables are specified for output. Here, the second face variable is registered as `scalar_face2`.

```
FACE VARIABLES = pressure_face scalar_face2
```

Aliases can be specified for each of the variables in a single command line. Thus, If you wanted to output the alias `pressuref` for face variable `pressure_face` and also wanted to output the alias `scalarf2` for face variable `scalar_face2`, you would enter the command line

```
FACE VARIABLES = pressure_face AS pressuref
  scalar_face2 AS scalarf2 .
```

The specification of an alias is always optional.

The second form of the command line is as follows:

```
FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>surface_names
  ...  <string>variable_name
  [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
  <string list>surface_names
```

This form of the `FACE VARIABLES` command line is similar to the first, except that the user can control which surfaces are used for output. The user can include a specific list of surfaces for output by using the `INCLUDE` option or the `ON` option. (The key word `INCLUDE` is synonymous with the key word `ON`.) Alternatively, the user can exclude a specific list of surfaces for output by using the `EXCLUDE` option.

Suppose that the face variable `pressure_face` from the above example has been registered for `surface_10`, `surface_11`, `surface_20`, and `surface_21`. If we only want to output the face variable for `surface_10`, `surface_11`, and `surface_21`, then we could specify the `FACE VARIABLES` command line as follows:

```
FACE VARIABLES = pressure_face AS pressuref
  INCLUDE surface_10, surface_11,
  surface_21 .
```

(In the above command line, the keyword `ON` could be substituted for `INCLUDE`.) Alternatively, we could use the command line

```
FACE VARIABLES = pressure_face AS pressuref
  EXCLUDE surface_20 .
```

In the above command lines, an alias for a surface can be substituted for a surface identifier. For example, if `center_case` is an alias for `surface_10`, then the string `center_case` could be substituted for `surface_10` in the above command lines. Because a surface identifier is a mesh entity, the alias for the surface identifier would be defined via an `ALIAS` command line in a `FINITE ELEMENT MODEL` command block.

Note that the list of identifiers uses a comma to separate one surface identifier from the next surface identifier.

8.1.1.4 Output Element Variables

```
ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | ELEMENT VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>block_names
      ... <string>variable_name
      [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>block_names
```

Any element variable in Presto can be selected for output in the results file by using the `ELEMENT VARIABLES` command line.

There are two forms of the `ELEMENT VARIABLES` command line. Either form will let you output an element variable.

The first form of the command line is as follows:

```
ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
```

Here, the string `variable_name` is the name of the element variable. The string `variable_name` can be a registered variable listed in Section 8.5, a user-defined variable (see Section 8.1.2 and Section 9.2.4), or a derived output quantity. The derived output option is discussed in detail in the latter portion of this discussion of the `ELEMENT VARIABLES` command line.

In the first form of the `ELEMENT VARIABLES` command line, the element variable is output for ALL element blocks that have the element variable as a registered variable.

For example, all the solid elements have `rotated_stress` as a registered variable. If you had a mesh consisting of hexahedral and tetrahedral elements and requested output of the element variable `rotated_stress`, then `rotated_stress` would be output for all element blocks consisting of hexahedral and tetrahedral elements.

It is possible to specify an alias for any of the element variables by using the `AS` specification. Suppose, for example, you wanted to output the stress in Presto, which is registered as `rotated_stress`, with the alias `stress`. You would then enter the command line

```
ELEMENT VARIABLES = rotated_stress AS stress .
```

In this example, stress is a symmetric tensor quantity. For a symmetric tensor quantity, suffixes are appended to the variable name (or alias name) to denote each symmetric tensor component. The results database would have six variable names associated with the stress: `stress_xx`, `stress_yy`, `stress_zz`, `stress_xy`, `stress_xz`, and `stress_yz`. Consult with Table 8.4 for a list of component identifiers for symmetric and full tensors. You can change the tensor component separator, an underscore in this example, by using the `COMPONENT SEPARATOR CHARACTER` command line (see Section 8.1.1.5).

The `ELEMENT VARIABLES` command line can be used any number of times within a `RESULTS OUTPUT` command block. It is also possible to specify more than one element variable for output on a command line, as indicated by the ellipsis in the command line format. In the following example, two element variables are specified for output. Here, the second element variable is registered as `stretch`.

```
ELEMENT VARIABLES = rotated_stress stretch
```

Aliases can be specified for each of the variables in a single command line. Thus, If you wanted to output the alias `stress` for element variable `rotated_stress` and also wanted to output the alias `strch` for face variable `stretch`, you would enter the command line

```
ELEMENT VARIABLES = rotated_stress AS stress
  stretch AS strch .
```

The specification of an alias is always optional.

The second form of the command line is as follows:

```
ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>block_names
  ... <string>variable_name
  [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
  <string list>block_names
```

This form of the `ELEMENT VARIABLES` command line is similar to the first, except that the user can control which element blocks are used for output. The user can include a specific list of element blocks for output by using the `INCLUDE` option or the `ON` option. (The key word `INCLUDE` is synonymous with the key word `ON`.) Alternatively, the user can exclude a specific list of element blocks for output by using the `EXCLUDE` option.

Suppose that the element variable `rotated_stress` from the above example exists for element blocks `block_10`, `block_11`, `block_20`, and `block_21`. If we only want to output the element variable for `block_10`, `block_11`, and `block_21`, then we could specify the `ELEMENT VARIABLES` command line as follows:

```
ELEMENT VARIABLES = rotated_stress AS stress
  INCLUDE block_10, block_11,
  block_21 .
```

(In the above command line, the keyword `ON` could be substituted for `INCLUDE`.) Alternatively, we could use the command line

```
ELEMENT VARIABLES = rotated_stress AS stress
  EXCLUDE block_20 .
```

In the above command lines, an alias for an element block can be substituted for an element block identifier. For example, if `center_case` is an alias for `block_10`, then the string `center_case` could be substituted for `block_10` in the above command lines. Because an element block identifier is a mesh entity, the alias for the element block identifier would be defined via an `ALIAS` command line in a `FINITE ELEMENT MODEL` command block.

Note that the list of identifiers uses a comma to separate one element block identifier from the next element block identifier.

As mentioned previously, you can use the `ELEMENT VARIABLES` command line for the output of a derived quantity. Derived quantities are calculated for solid elements and for shell elements. A derived quantity is identified by supplying one of the available options listed in Table 8.1, Table 8.2, or Table 8.3 for the string `variable_name`. For example, you would use the following command to compute and output von Mises stress on all solid elements:

```
ELEMENT VARIABLES = von_mises
```

Note that the `AS` specification can be included in the command line when you output derived quantities. The above command line could be written as

```
ELEMENT VARIABLES = von_mises AS vm .
```

Table 8.1 gives the complete set of derived stresses for solid elements and for shell elements. For the shell elements, the derived quantities from the stress tensor are given at each of the integration points. For the shell elements, the derived quantities from the stress are given at each of the integration points. A suffix ranging from 1 to the number of integration points is attached to the derived quantity to indicate the corresponding integration point. The suffix is padded with leading zeros. If the number of integration points is less than 10, the suffix has the form _*i*, where *i* ranges from 1 to the number of integration points. If the number of integration points is greater than or equal to 10 and less than 100, the sequence of suffixes takes the form `_01`, `_02`, `_03`, and so forth. Finally, if the number of integration points is greater than or equal to 100, the sequence of suffixes takes the form `_001`, `_002`, `_003`, and so forth. As an example, if the von Mises stress is requested for a shell element with 15 integration points, then the derived quantities `von_mises_01`, `von_mises_02`, . . . , `von_mises_15` are output for the shell element.

Table 8.1: Derived Stress Output for Elements

Option	**Option Description**
von_mises	Von Mises stress norm.
hydrostatic_stress	One-third the trace of the stress sensor.
stress_invariant_1	Trace of the stress tensor.
stress_invariant_2	Second invariant of the stress tensor.
stress_invariant_3	Third invariant of the stress tensor.
max_principal	Largest eigenvalue of the stress tensor.
intermediate_principal	Middle eigenvalue of the stress tensor.
min_principal	Smallest eigenvalue of the stress tensor
max_shear	Maximum shear stress from Mohr's circle.
octahedral_shear	Octahedral shear norm of the stress tensor.
effective_log_strain	Effective log strain.

Most solid elements use only one integration point. For solid elements with multiple integration points, the conventions used for multiple integration points in shells are also used for multiple integration points in solids.

In the above discussion concerning the format for output at multiple integration points, the underscore character preceding the integration point number can be replaced by another delimiter or the underscore character can be eliminated by use of the `COMPONENT SEPARATOR CHARACTER` command line (see Section 8.1.1.5).

Table 8.2 gives the complete set of quantities derived from the log strain for solid elements. (Solid elements generate log strain information, while shell elements generate integrated strain information.)

Table 8.2: Derived Log Strain Output for Solid Elements

Option	**Option Description**
effective_log_strain	Effective log strain.
log_strain_invariant_1	Trace of the log strain tensor.
log_strain_invariant_2	Second invariant of the log strain tensor.
log_strain_invariant_3	Third invariant of the log strain tensor.
max_principal_log_strain	Largest eigenvalue of the log strain tensor.
intermediate_principal_log_strain	Middle eigenvalue of the log strain tensor.
min_principal_log_strain	Smallest eigenvalue of the log strain tensor.
max_shear_log_strain	Maximum shear log strain from Mohr's circle.
octahedral_shear_log_strain	Octahedral strain norm of the log strain tensor.

Most solid elements use only one integration point. For solid elements with multiple integration points, the conventions used for multiple integration points in shells are also used for multiple integration points in solids.

Table 8.3 gives the complete set of quantities derived from the integrated strain for shell elements. (Solid elements generate log strain information, while shell elements generate integrated strain information.) For the shell elements, the derived quantities from the integrated strain are given at each of the integration points. A suffix ranging from 1 to the number of integration points is attached to the derived quantity to indicate the corresponding integration point. The suffix is padded with leading zeros. If the number of integration points is less than 10, the suffix has the form $\_i$, where i ranges from 1 to the number of integration points. If the number of integration points is greater than or equal to 10 and less than 100, the sequence of suffixes takes the form `_01`, `_02`, `_03`, and so forth. Finally, if the number of integration points is greater than or equal to 100, the sequence of suffixes takes the form `_001`, `_002`, `_003`, and so forth. As an example, if the effective strain is requested for a shell element with 15 integration points, then the derived quantities `effective_strain_01`, `effective_strain_02`, . . . , `effective_strain_15` are output for the shell element.

In the above discussion concerning the output format for multiple integration points for shells, the underscore character preceding the integration point number can be replaced by another delimiter, or the underscore character can be eliminated by use of the `COMPONENT SEPARATOR CHARACTER` command line (see Section 8.1.1.5).

Table 8.3: Derived Strain Output for Shell Elements

Option	**Option Description**
effective_strain	Effective strain.
strain_invariant_1	Trace of the strain tensor.
strain_invariant_2	Second invariant of the strain tensor.
strain_invariant_3	Third invariant of the strain tensor.
max_principal_strain	Largest eigenvalue of the strain tensor.
intermediate_principal_strain	Middle eigenvalue of the strain tensor.
min_principal_strain	Smallest eigenvalue of the strain tensor.
max_shear_strain	Maximum shear strain from Mohr's circle.
octahedral_shear_strain	Octahedral strain norm of the strain tensor.

8.1.1.5 Component Separator Character

```
COMPONENT SEPARATOR CHARACTER = <string>character|NONE
```

The component separator character is used to separate an output-variable base name from any suffixes. For example, the variable `stress` can have the suffixes `xx`, `yy`, etc. By default, the base name is separated from the suffixes with an underscore character so that we have `stress_xx`, `stress_yy`, etc. in the results output file.

You can replace the underscore as the default separator by using the above command line. If you wanted to use the period as the separator, then you would use the following command line:

```
COMPONENT SEPARATOR CHARACTER = .
```

For our example with stress, the stress components would then be `stress.xx`, `stress.yy`, etc. in the results output file. If the stress is for a shell element, there is also an integration point suffix preceded, by default, with an underscore. The above command line also resets the underscore character that precedes the integration point suffix. For our example with the `stress` base name and the underscore replaced by the period, the results file would have `stress.xx.01`, `stress.xx.02`, etc., for the shell elements.

You can eliminate the separator with an empty string or `NONE`.

8.1.1.6 Output Global Variables

```
GLOBAL VARIABLES = <string>variable_name
```

```
    [AS <string>dbase_variable_name
    <string>variable_name AS <string>dbase_variable_name ...]
```

Any global variable in Presto can be selected for output in the results file by using the `GLOBAL VARIABLES` command line. The string `variable_name` is the name of the global variable. The string `variable_name` can be either a registered variable listed in Section 8.5 or a user-defined variable (see Section 8.1.2 and Section 9.2.4).

With the `AS` specification, you can specify the variable and select an alias for this variable in the results file. Suppose, for example, you wanted to output the time steps in Presto, which are identified as `timestep`, with the alias `tstep`. You would then enter the command line

```
GLOBAL VARIABLES = timestep AS tstep .
```

The `GLOBAL VARIABLES` command line can be used any number of times within a `RESULTS OUTPUT` command block. It is also possible to specify more than one global variable for output on a command line. If you also wanted to output the kinetic energy, which is registered as `KineticEnergy`, with the alias `ke`, you would enter the command line

```
GLOBAL VARIABLES = timestep as tstep
  KineticEnergy as ke .
```

The specification of an alias is always optional.

8.1.1.7 Set Begin Time for Results Output

```
START TIME = <real>output_start_time
```

Using the `START TIME` command line, you can write output to the results file beginning at time `output_start_time`. No results will be written before this time. If other commands set times for results (`AT TIME`, `ADDITIONAL TIMES`) that are less than `output_start_time`, those times will be ignored, and results will not be written at those times.

8.1.1.8 Adjust Interval for Time Steps

```
TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
```

This command line is used to specify that the output will be at exactly the times specified. To hit the output times exactly in an explicit, transient dynamics code, it is necessary to adjust the time step as the time approaches an output time. The integer value `steps` in the `TIMESTEP ADJUSTMENT INTERVAL` command line specifies the number of time steps to look ahead in order to adjust the time step.

If this command line does not appear, results are output at times closest to the specified output times.

8.1.1.9 Output Interval Specified by Time Increment

```
AT TIME <real>time_begin INCREMENT = <real>time_increment_dt
```

At the time specified by `time_begin`, results will be output every time increment given by the real value `time_increment_dt`.

8.1.1.10 Additional Times for Output

```
ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.1.1.9, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional output times.

8.1.1.11 Output Interval Specified by Step Increment

```
AT STEP <integer>step_begin INCREMENT =
  <integer>step_increment
```

At the step specified by `step_begin`, results will be output every step increment given by the integer value `step_increment`.

8.1.1.12 Additional Steps for Output

```
ADDITIONAL STEPS = <integer>output_step1
  <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.1.1.11, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of additional output steps.

8.1.1.13 Set End Time for Results Output

```
TERMINATION TIME = <real>termination_time_value
```

Results will not be written to the results file after time `termination_time_value`. If other commands set times for results (`AT TIME`, `ADDITIONAL TIMES`) that are greater than `termination_time_value`, those times will be ignored, and results will not be written at those times.

8.1.1.14 Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

In an analysis with multiple regions, it can be difficult to synchronize output such as results files. To help synchronize output for analyses with multiple regions, you can define an `OUTPUT SCHEDULER` command block at the domain level. The scheduler can then be referenced in the `RESULTS OUTPUT` command block via the `USE OUTPUT SCHEDULER` command line. The string `scheduler_name` must match a name used in an `OUTPUT SCHEDULER` command block. See Section 8.4 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.1.1.15 Write Results If System Error Encountered

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The `OUTPUT ON SIGNAL` command line is used to initiate the writing of a results file when the system encounters a type of system error. Only one error type in the list of error types should be entered for this command line. Generally, these system errors cause the code to terminate before the code can add any current results output (results output past the last results output time step) to the results output file. If the code encounters the specified type of error during execution, a results file will be written before execution is terminated.

This command line can also be used to force the writing of a results file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

`kill -s SIGUSR2` *pid*

to terminate execution and force the writing of a results file. In the above system command line, *pid* is the process identifier, which is an integer.

Note that the `OUTPUT ON SIGNAL` command line is primarily a debugging tool for code developers.

8.1.2 User-Defined Output

```
BEGIN USER OUTPUT
  # {mesh-entity set commands}
  NODE SET = <string_list>nodeset_names
  SURFACE = <string_list>surface_names
  BLOCK = <string_list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list> surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # compute global result command
  COMPUTE GLOBAL <string>results_var_name AS
    <string>SUM|AVERAGE|MAX|MIN OF <string>NODAL|ELEMENT
    <string>value_var_name [(<integer>component_num)]
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name |
    SURFACE SUBROUTINE = <string>subroutine_name |
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # copy command
  COPY ELEMENT VARIABLE <string>ev_name TO NODAL VARIABLE
    <string>nv_name
  #
  # compute for element death
  COMPUTE AT EVERY TIME STEP
  #
  # additional command
  ACTIVE PERIODS = <string list>period_names
END [USER OUTPUT]
```

The `USER OUTPUT` command block lets the user generate specialized output information derived from analysis results such as element stresses, displacements, and velocities. For example, the `USER OUTPUT` command block could be used to sum the contact forces in a particular direction in the global axes and on a certain surface to give a net resultant contact force on that surface. In this example, we essentially postprocess contact information and reduce it to a single value for a surface (or set of surfaces). This, then, is one of the purposes of the `USER OUTPUT` command block—to postprocess analysis results as the code is running and produce a reduced set of specialized output information. The `USER OUTPUT` command block offers an alternative to writing out large quantities of data and then postprocessing them with some external code in order to produce specialized output results. Another use of the `USER OUTPUT` command block is to generate variables that can be used for element death. An element can be killed by using some criterion based on a user variable defined in the `USER OUTPUT` command block.

There are three options for calculating user-defined quantities. In the first option, a single command line in the command block is used to compute reductions of registered variables on subsets of the mesh. This option makes use of the `COMPUTE GLOBAL` command line. The above example of the contact force represents an instance where we can accomplish the desired result simply by using the `COMPUTE GLOBAL` command line. In the second option, the command block specifies a user subroutine to run immediately preceding output to calculate any desired variable. This option makes use of a `NODE SET`, `SURFACE`, or `ELEMENT BLOCK SUBROUTINE` command line. Finally, there is an option to copy an element variable for an element to the nodes associated with the element, via the `COPY ELEMENT VARIABLE` command line. This copy option is a specialized option that has been made available primarily for creating results files for some of the postprocessing tools used with Presto. You can use only one of the three options—compute global result, user subroutine, or copy—in a given command block.

For the compute global result option, a user-defined variable is automatically generated. This user-defined variable is given whatever name the user selects for `results_var_name` in the above specification for the `COMPUTE GLOBAL` command line. If the user subroutine or copy option is used, the user will need to define some type of user variable with the `USER VARIABLE` command block described in Section 9.2.4.

User-defined variables, whether they are generated via the compute global result option or the `USER VARIABLE` command block, are not automatically written to a results or history file. If the user wants to output any user-defined variables, these variables must be referenced in a results or history output specification (see Section 8.1.1 and Section 8.2, which describe the output of variables to results files and history files, respectively). If the user wants to use any user-defined variable for element death, the user must include the `COMPUTE AT EVERY TIME STEP` command line.

The `USER OUTPUT` command block contains five groups of commands—mesh-entity set, compute global result, user subroutine, copy, and compute for element death. Each of these command groups is basically independent of the others. In addition to the command lines in the five command groups, there is an additional command line: `ACTIVE PERIODS`. The following sections provide descriptions of the different command groups and the `ACTIVE PERIODS` command line.

8.1.2.1 Mesh-Entity Set Commands

The {`mesh-entity set commands`} portion of the `USER OUTPUT` command block specifies the nodes, element faces, or elements associated with the variable to be output. This portion of the command block can include some combination of the following command lines:

```
NODE SET = <string_list>nodeset_names
SURFACE = <string_list>surface_names
BLOCK = <string_list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list> surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes, element faces, or elements. See Section 6.1 for more information about the use of these command lines for mesh entities. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

8.1.2.2 Compute Global Result Command

If the compute global result option is selected, Presto returns a single global value or a set of global values by examining the current values for a named registered nodal or element variable and then calculating the output according to a user-specified operation. A single global value, for example, might be the maximum value of one of the stress components of all the elements in our specified set; a set of global values would be the maximum value of each stress component of all elements in our specified set. Importantly, this option can only be used with a variable that is registered in Presto, not a variable that is created by the user via the `USER VARIABLE` command block.

The following command line is related to the compute global result option.

```
COMPUTE GLOBAL <string>results_var_name AS
  <string>SUM|AVERAGE|MAX|MIN OF <string>NODAL|ELEMENT
  <string>value_var_name [(<integer>component_num)]
```

In the above command line, the following definitions apply:

- The string `results_var_name` is the name of a new global variable in which to store the reduced results. To output this variable in a results file or a history file, you will simply use whatever you have selected for `results_var_name` as the variable name in a `GLOBAL VARIABLES` command line.

- Four different methods (or reduction types) are available for specifying the operation that will be performed on the values retrieved from the registered variable: `SUM`, `AVERAGE`, `MAX`, and `MIN`. Only one of these methods can be selected in a `GLOBAL COMPUTE` command line, however. `SUM` adds the variable value of all included mesh entities. `AVERAGE` takes the average value of the variable over all included mesh entities. `MAX` finds the maximum value over all included mesh entities. `MIN` finds the minimum value over all included mesh entities.

- The registered variable used to compute the global variable must be either a nodal quantity or an element quantity, as specified by the `NODAL` or `ELEMENT` option.

- The string `value_var_name` is the name of the registered variable (see Section 8.5 for a listing of the registered variables).

- There is an optional input, `component_num` (meaning component number), on the command line that allows the user to specify a particular (and single) value that will be returned for the new global variable. If `component_num` is not included in the command line, the global variable will have as many components as the registered variable. For example, if `component_num` was not specified and the registered variable was a displacement (which has three components—x, y, and z), the global variable that is returned would have three values. Each component of the registered variable will be reduced independently and placed in the corresponding position of the returned global variable. In the output file, the returned values will begin with the name of the global variable and be appended with the identification of the kind of component. For example, if `myresults` was specified for `results_var_name` and the registered variable was a displacement, the output values would be displayed as `myresults_x`, `myresults_y`, and `myresults_z`.

  Usage of `component_num`, which must be enclosed in parentheses, requires that you enter an integer number that corresponds to the position of the desired value in the set of possible values for the named registered variable. In other words, this number does not indicate how many components are stored for the variable. See the section below titled “Determining the Component Number” for further information on obtaining the required value for `component_num`.

The following is an example of using the `GLOBAL COMPUTE` command line to compute the net x-direction reaction force:

```
COMPUTE GLOBAL wall_x_reaction AS SUM OF NODAL reactions(1)
```

Determining the Component Number: If you want to specify that a specific value is returned for the global variable, the one named `results_var_name`, select an integer that corresponds to the position of that value in Table 8.4. Thus, for example, if you only wanted the ZZ component of a registered variable that was a symmetric tensor, the value for `component_num` would be specified as "(3)" in the command line.

Table 8.4: Selection of Component Number

Variable Type	**component_num and Description**	**Notes**
Vector	1 X component 2 Y component 3 Z component	A vector has three components. Displacements, for example, are handled as vectors.
Symmetric Tensor	1 XX component 2 YY component 3 ZZ component 4 XY component 5 YZ component 6 ZX component	Symmetric tensors have six components. Stresses for most solid elements are symmetric tensors.
Full Tensor	1 XX component 2 YY component 3 ZZ component 4 XY component 5 YZ component 6 ZX component 7 YX component 8 ZY component 9 XZ component	Full tensors are used occasionally by Presto. Examples include velocity gradients and material rotations.

8.1.2.3 User Subroutine Commands

If the user subroutine option is used, the user-defined output quantities will be calculated by a subroutine that is written by the user explicitly for this purpose. The subroutine will be called by Presto at the appropriate time to perform the calculations. User subroutines allow for more generality in computing user-defined results than the `COMPUTE GLOBAL` command line. Suppose, for example, you had an analytic solution for a problem and wanted to compute the difference between some analytic value and a corresponding computed value throughout an analysis. The user subroutine option would allow you to make this comparison. The full details for user subroutines are given in Chapter 9.

The following command lines are related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line, the `SURFACE SUBROUTINE` command line, or the `ELEMENT BLOCK SUBROUTINE` command line. The particular command line selected depends on the mesh-entity type of the variable for which the result quantities are being calculated. For example, variables associated with nodes would be calculated by using a `NODE SET SUBROUTINE` command line, variables associated with faces by using a `SURFACE SUBROUTINE` command line, and variables associated with elements by using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user. A user subroutine in the `USER OUTPUT` command block returns no values. Instead, it performs its operations directly with commands such as `aupst_put_nodal_var`, `aupst_put_elem_var`, and `aupst_put_global_var`. Consult with Chapter 9 for further discussion of these various put commands.

Following the selected command line (`NODE SET SUBROUTINE`, `SURFACE SUBROUTINE`, or `ELEMENT BLOCK SUBROUTINE`) are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided in Chapter 9.

Importantly, to implement the user subroutine option and output the calculated information, you would also need to do the following:

1. Create the user-defined variable with a `USER VARIABLE` command block.

2. Calculate the results for the user-defined variable in the user subroutine.
3. Write the results for the user-defined variable to an output file by referencing it in a `RESULTS OUTPUT` command block and/or a `HISTORY OUTPUT` command block. In the `RESULTS OUTPUT` command block, you would use a `NODAL VARIABLES` command line, an `ELEMENT VARIABLES` command line, or a `GLOBAL VARIABLES` command line, depending on how you have defined the variable in the `USER VARIABLE` command block. Similarly, in the `HISTORY OUTPUT` command block, you would use the applicable form of the `VARIABLE` command line, depending on how you have defined the variable in the `USER VARIABLE` command block.

8.1.2.4 Copy Command

```
COPY ELEMENT VARIABLE <string>ev_name TO NODAL VARIABLE
  <string>nv_name
```

The `COPY ELEMENT VARIABLE` command line copies the value of an element variable to a node associated with the element. The element variable to be copied is specified by `ev_name`; the name of the nodal variable to which the value is being transferred is `nv_name`. The nodal variable must be specified as a user-defined variable.

8.1.2.5 Compute for Element Death Command

```
COMPUTE AT EVERY TIME STEP
```

The `COMPUTE AT EVERY TIME STEP` command line is required if a user-defined variable is used in a criterion for element death. (Section 9.2.4 discusses user-defined variables, and Section 5.5 discusses element death.) If this command line appears in the `USER OUTPUT` command block, a user-defined variable in the command block will be written at every time step. For element death, a user-defined variable must be calculated at every time step.

8.1.2.6 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `USER OUTPUT` command block:

```
ACTIVE PERIODS = <string list>period_names
```

This command line determines when the boundary condition is active. See Section 2.5 for more information about this optional command line.

8.2 History Output

```
BEGIN HISTORY OUTPUT <string>history_name
  DATABASE NAME = <string>history_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  TITLE <string>user_title
  #
  # for global variables
  VARIABLE = GLOBAL
    <string>variable_name
    [AS <string>history_variable_name]
  #
  # for mesh entity - node, edge, face,
  # element - variables
  VARIABLE =
    NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
    [AS <string>history_variable_name]
  #
  # for nearest point output of mesh entity - node,
  # edge, face, element - variables
  VARIABLE =
    NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    NEAREST LOCATION <real>global_x,
      <real>global_y>, <real>global_z
    [AS <string>history_variable_name]
  START TIME = <real>output_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
END [HISTORY OUTPUT <string>history_name]
```

A history file gives nodal variable results (displacements, forces, etc.) for specific nodes, edge variable results for specific edges, face variable results for specific faces, element results (stress, strain, etc.) for specific elements, and global results at specified times. You can specify a history file, the results to be included in this file, and the frequency at which results are written by using a `HISTORY OUTPUT` command block. The command block appears inside the region scope. For history output, you will typically work with node and element variables, and, on some occasions, face variables.

More than one history file can be specified for an analysis. For each history file, there will be one `HISTORY OUTPUT` command block. The command block for a history file description begins with

```
    BEGIN HISTORY OUTPUT <string>history_name
```

and is terminated with

```
    END [HISTORY OUTPUT <string>history_name] ,
```

where `history_name` is a user-selected name for the command block. Nested within the `HISTORY OUTPUT` command block are a set of command lines, as shown in the block summary given above. The first two command lines listed (`DATABASE NAME` and `DATABASE TYPE`) give pertinent information about the history file. The command line

```
    DATABASE NAME = <string>history_file_name
```

gives the name of the history file with the string `history_file_name`. If the history file is to appear in the current directory and is named `job.h`, this command line would appear as

```
    DATABASE NAME = job.h .
```

If the history file is to be created in some other directory, the command line would have to show the path to that directory.

If the history file does not use the Exodus II format, you must specify the format for the history file using the command line

```
    DATABASE TYPE = <string>database_type(exodusII) .
```

Currently, both the Exodus II database and the XDMF database [1] are supported in Presto. Exodus II is more commonly used than XDMF. Other options may be added in the future.

The `OVERWRITE` command line can be used to prevent the overwriting of existing history files.

```
    OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
      (ON|TRUE|YES)
```

The `OVERWRITE` command line allows only a single value. If you set the value to `FALSE`, `NO`, or `OFF`, the code will terminate before existing history files can be overwritten. If you set the value to `TRUE`, `YES`, or `ON`, then existing history files can be overwritten (the default status). Suppose, for example, that we have an existing history file named `job21.h`. Suppose also that we have an input file with a `HISTORY OUTPUT` command block that contains the `OVERWRITE` command line set to `ON` and the `DATABASE NAME` command line set to

```
DATABASE NAME = job21.h .
```

If you run the code under these conditions, the existing history file `job21.h` will be overwritten.

Whether or not history files are overwritten is also impacted by the use of the automatic read and write option for restart files described in Section 8.3.1.1. If you use the automatic read and write option for restart files, the history files, like the restart files, are automatically managed. The automatic read and write option in restart adds extensions to file names and prevents the overwriting of any existing restart or history files. For the case of a user-controlled read and write of restart files (Section 8.3.1.2) or of no restart, however, the `OVERWRITE` command line is useful for preventing the overwriting of history files.

You may add a title to the history file by using the `TITLE` command line. Whatever you specify for the `user_title` will be written to the history file. Some of the programs that process the history file (such as various SEACAS programs [2]) can read and display this information.

The other command lines that appear in the `HISTORY OUTPUT` command block determine the type and frequency of information that is output. Descriptions of these command lines follow in Section 8.2.1 through Section 8.2.11. Note that the command lines for controlling the frequency of history output (in Section 8.2.1 through Section 8.2.11) are the same as those for controlling the frequency of results output. These frequency-related command lines are repeated here for convenience.

8.2.1 Output Variables

The `VARIABLE` command line is used to select variables for output in the history file. One of several types of variables—`GLOBAL`, `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT`—can be selected for output. The form of the command line varies depending on the type of variable that is selected for output.

8.2.1.1 Global Output Variables

```
VARIABLE = GLOBAL
```

```
    <string>variable_name
    [AS <string>history_variable_name]
```

This form of the `VARIABLE` command line lets you select any global variable for output in the history file. The variable is selected with the string `variable_name`. The string `variable_name` is the name of the global variable and can be either a registered variable listed in Section 8.5 or a user-defined variable (see Section 8.1.2 and Section 9.2.4).

You can also specify an arbitrary name, `history_variable_name`, for the selected entity following the `AS` key word. For example, suppose you want to output the kinetic energy (KineticEnergy) as KE. The command line to obtain the kinetic energy in the history file would be

```
VARIABLE = GLOBAL KineticEnergy AS KE .
```

The specification of an alias is always optional.

8.2.1.2 Mesh Entity Output Variables

```
VARIABLE =
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
  AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
  [AS <string>history_variable_name]
```

This form of the `VARIABLE` command line lets you select any nodal, edge, face, or element variable for a specific mesh entity for output in the history file. For example, this form of the `VARIABLE` command line will let you pick the displacement at a specific node and output the displacement to the history file using an alias that you have chosen.

For this form of the `VARIABLE` command line, the mesh entity type following the delimiter (=) is set to `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT` depending on the variable (set by `variable_name`) to be output. If the mesh entity type is set to `NODE` (or `NODAL`), `EDGE`, or `FACE`, the string `variable_name` can be either a registered variable listed in Section 8.5 or a user-defined variable (see Section 8.1.2 and Section 9.2.4). If the mesh entity type is set to `ELEMENT`, the string `variable_name` can be a registered variable listed in Section 8.5, a user-defined variable (see Section 8.1.2 and Section 9.2.4), or a derived output quantity. See the latter portion of Section 8.1.1.4 for a detailed discussion of derived output. A complete list of derived output quantities is given in Tables 8.1, 8.2, and 8.3 in Section 8.1.1.4.

Selection of a specific mesh entity follows the `AT` key word. You select a mesh entity type (`NODE` [or `NODAL`], `EDGE`, `FACE`, or `ELEMENT`) followed by the specific integer identifier, `entity_id`, for the mesh entity. You can specify an arbitrary name, `history_variable_name`, for the selected entity following the `AS` key word. For example, suppose you want to output the accelerations at node 88. The command line to obtain the accelerations at node 88 for the history file would be

```
VARIABLE = NODE ACCELERATION AT NODE 88 AS accel_88 ,
```

where `accel_88` is the arbitrary name that will be used for this history variable in the history file.

Note that either the key word `NODE` or `NODAL` can be used for nodal quantities. The specification of an alias is always optional.

As an example of derived output, suppose you wanted to output the von Mises stress for solid element 1024. The command line to obtain the von Mises stress for element 1024 for the history file would be

```
VARIABLE = ELEMENT VON_MISES AT ELEMENT 1024 AS vm_1024 ,
```

where `vm_1024` is the arbitrary name that will be used for this history variable in the history file.

8.2.1.3 Nearest Point Output Variables

```
VARIABLE =
   NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
   NEAREST LOCATION <real>global_x,
     real<global_y>, real<global_z>
   [AS <string>history_variable_name]
```

This form of the `VARIABLE` command line lets you select any nodal, edge, face, or element variable for output in the history file using a nearest point criterion. The command line described in this subsection is an alternative to the command line described in the preceding section, Section 8.2.1.2, for obtaining history output. The command line in this section or the command line in Section 8.2.1.2 produces history files with variable information. The difference in these two command lines (Section 8.2.1.3 and Section 8.2.1.2) is simply in how the variable information is selected.

For the above form of the `VARIABLE` command line, the mesh entity type following the delimiter (=) is set to `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT` depending on the variable (set by `variable_name`) to be output. If the mesh entity type is set to `NODE` (or `NODAL`), `EDGE`, or `FACE`, the string `variable_name` can be either a registered variable listed in Section 8.5 or a user-defined variable (see Section 8.1.2 and Section 9.2.4). If the mesh entity type is set to `ELEMENT`, the string `variable_name` can be a registered variable listed in Section 8.5, a user-defined variable (see Section 8.1.2 and Section 9.2.4), or a derived output quantity. See the latter portion of

Section 8.1.1.4 for a detailed discussion of derived output. A complete list of derived output quantities is given in Tables 8.1, 8.2, and 8.3 in Section 8.1.1.4.

The specific mesh entity used for output is determined by global coordinates specified by the `NEAREST LOCATION` key word and its associated input parameters—`global_x`, `global_y`, `global_z`. The specific mesh entity chosen for output is as follows:

- If the mesh entity has been set to `NODE` (or `NODAL`), the node in the mesh selected for output is the node whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.
- If the mesh entity has been set to `EDGE`, the edge in the mesh selected for output is the edge with a center point (the average location of the two end points of the edge) whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.
- If the mesh entity has been set to `FACE`, the face in the mesh selected for output is the face with a centroid whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.
- If the mesh entity has been set to `ELEMENT`, the element in the mesh selected for output is the element with a centroid whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.

Note that, in all the above cases, the original model coordinates are used when selecting the nearest entity, not the current coordinates.

You can specify an arbitrary name, `history_variable_name`, for the selected entity following the `AS` key word. As an example, suppose you want to output the accelerations at a node closest to the point with global coordinates (1012.0, 54.86, 103.3141). The command line to obtain the accelerations at the node closest to this location for the history file would be

```
VARIABLE = NODE ACCELERATION
  NEAREST LOCATION 1012.0, 54.86, 103.3141 AS accel_near ,
```

where `accel_near` is the arbitrary name that will be used for this history variable in the history file.

Note that either the key word `NODE` or `NODAL` can be used for nodal quantities. The specification of an alias is always optional.

8.2.2 Outputting History Data on a Node Set

It is commonly desired to output history data on a single-node node set. If a mesh file is slightly modified, the node and element numbers will completely change. The node associated with a node set, however, remains the same, i.e., the node in the node set retains the same initial geometric location with the same connectivity to other elements even when its node number changes. Therefore, we might want to specify the history output for a node set with a single node rather than with the global identifier for a node. This can easily be accomplished, as follows:

```
 begin user output
   node set = nodelist_1
   compute global disp_ns_1 as average of nodal displacement
 end

begin history output
  variable = global disp_ns_1
end
```

If `nodelist_1` contains only a single node, the history output variable `disp_ns_1` will contain the displacement for the single node in the node set. If `nodelist_1` contains multiple nodes, the average displacement of the nodes will be output.

8.2.3 Set Begin Time for History Output

```
START TIME = <real>output_start_time
```

Using the `START TIME` command line, you can write history variables to the history file beginning at time `output_start_time`. No history variables will be written before this time. If other commands set times for history output (`AT TIME`, `ADDITIONAL TIMES`) that are less than `output_start_time`, those times will be ignored, and history output will not be written at those times.

8.2.4 Adjust Interval for Time Steps

```
TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
```

This command line is used to specify that the output will be at exactly the times specified. To hit the output times exactly in an explicit, transient dynamics code, it is necessary to adjust the time step as the time approaches an output time. The integer value `steps` in the `TIMESTEP ADJUSTMENT INTERVAL` command line specifies the number of time steps to look ahead in order to adjust the time step.

If this command line does not appear, history variables are output at times closest to the specified output times.

8.2.5 Output Interval Specified by Time Increment

```
AT TIME <real>time_begin INCREMENT = <real>time_increment_dt
```

At the time specified by `time_begin`, history variables will be output every time increment given by the real value `time_increment_dt`.

8.2.6 Additional Times for Output

```
ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.2.5, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional output times.

8.2.7 Output Interval Specified by Step Increment

```
AT STEP <integer>step_begin INCREMENT =
  <integer>step_increment
```

At the step specified by `step_begin`, history variables will be output every step increment given by the integer value `step_increment`.

8.2.8 Additional Steps for Output

```
ADDITIONAL STEPS = <integer>output_step1
  <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.2.7, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of g

8.2.9 Set End Time for History Output

```
TERMINATION TIME = <real>termination_time_value
```

History output will not be written to the history file after time `termination_time_value`. If other commands set times for history output (`AT TIME`, `ADDITIONAL TIMES`) that are greater than `termination_time_value`, those times will be ignored, and history output will not be written at those times.

8.2.10 Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

In an analysis with multiple regions, it can be difficult to synchronize output such as history files. To help synchronize output for analyses with multiple regions, you can define an `OUTPUT SCHEDULER` command block at the domain level. The scheduler can then be referenced in the `HISTORY OUTPUT` command block via the `USE OUTPUT SCHEDULER` command line. The string `scheduler_name` must match a name used in an `OUTPUT SCHEDULER` command block. See Section 8.4 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.2.11 Write History If System Error Encountered

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The `OUTPUT ON SIGNAL` command line is used to initiate the writing of a history file when the system encounters a type of system error. Only one error type in the list of error types should be entered for this command line. Generally, these system errors cause the code to terminate before the code can add any current history output (history output past the last history output time step) to the history file. If the code encounters the specified type of error during execution, a history file will be written before execution is terminated.

This command line can also be used to force the writing of a history file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

```
kill -s SIGUSR2 pid
```

to terminate execution and force the writing of a results file. In the above system command line, *pid* is the process identifier, which is an integer.

Note that the `OUTPUT ON SIGNAL` command line is primarily a debugging tool for code developers.

8.3 Restart Data

```
BEGIN RESTART DATA <string>restart_name
  DATABASE NAME = <string>restart_file
  INPUT DATABASE NAME = <string>restart_input_file
  OUTPUT DATABASE NAME = <string>restart_output_file
  DATABASE TYPE = <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  START TIME = <real>restart_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
  OVERLAY COUNT = <integer>overlay_count
  CYCLE COUNT = <integer>cycle_count
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
END [RESTART DATA <string>restart_name]
```

You can specify restart files, either to be written to or read from, and the frequency at which restarts are written by using a `RESTART DATA` command block. The command block appears inside the region scope. To initiate a restart, the `RESTART TIME` command line (see Section 2.1.3.1) or the `RESTART` command line (see Section 2.1.3.2) must also be used. These command lines appear in the domain scope.

NOTE: In addition to the times at which you request restart information to be written, restart information is automatically written when an element inverts.

The `RESTART DATA` command block begins with the input line

```
BEGIN RESTART DATA <string>restart_name
```

and is terminated with

```
END [RESTART DATA <string>restart_name] ,
```

where `restart_name` is a user-selected name for the `RESTART DATA` command block.

Nested within the `RESTART DATA` command block are a set of command lines, as shown in the block summary given above.

We begin the discussion of the `RESTART DATA` command block with various options regarding the use of restart in general. In Section 8.3.1, you will learn how to use the `DATABASE NAME`, `INPUT DATABASE NAME`, `OUTPUT DATABASE NAME`, and `DATABASE TYPE` command lines. Usage of the first three of these command lines is tied to the two restart-related command lines `RESTART` and `RESTART TIME`, which are found in the domain scope.

Section 8.3.2 discusses use of the `OVERWRITE` command line, which will prevent or allow the overwriting of existing restart files. (Note that this command line also appears in the command blocks for results output and history output.)

The other command lines that appear in the `RESTART DATA` command block determine the frequency at which restarts are written. Descriptions of these command lines follow in Section 8.3.3 through Section 8.3.13. Note that the command lines for controlling the frequency of restart output are the same as those for controlling the frequency of results output and history output. These frequency-related command lines are repeated here for convenience.

8.3.1 Restart Options

```
DATABASE NAME = <string>restart_file
INPUT DATABASE NAME = <string>restart_input_file
OUTPUT DATABASE NAME = <string>restart_output_file
DATABASE TYPE = <string>database_type(exodusII)
```

You can read from and create restart files in an automated fashion, the preferred method, or you can carefully control how you read from and create restart files. In our discussion of the overall options for the use of restart, we begin with the first three command lines listed above (`DATABASE NAME`, `INPUT DATABASE NAME`, and `OUTPUT DATABASE NAME`). All three of these command lines specify a parameter that is a file name or a directory path and file name. If the parameter begins with the "/" character, it is an absolute path; otherwise, the path to the current directory will be prepended to the parameter on the command line. Suppose, for example, that we want to work with a restart file named `component.rst` in the current directory. If we are using the `DATABASE NAME` command line, then this command line would appear as

```
DATABASE NAME = component.rst .
```

If we wanted to read or create files in some other directory, the command line would have to show the path to that directory in addition to our file name.

The `DATABASE NAME` command line will let you read restart information and write restart information to the same file. Section 8.3.1.1 through Section 8.3.1.4 show how this command line is used in particular instances.

You can specify a restart file to read from by using the command line

```
INPUT DATABASE NAME = <string>restart_input_file .
```

You can specify a restart file to write to by using the command line

```
OUTPUT DATABASE NAME = <string>restart_output_file .
```

Note that you must use either a `DATABASE NAME` command line or the `INPUT DATABASE NAME` command line/`OUTPUT DATABASE NAME` command line pair, but not both, in a `RESTART DATA` command block.

If the restart file does not use the Exodus II format, you must specify the format for the results file using the `DATABASE TYPE` command line:

```
DATABASE TYPE = <string>database_type(exodusII)
```

Currently, the Exodus II database and the XDMF database [1] are supported in Presto. Exodus II is more commonly used than XDMF. Other options may be added in the future.

8.3.1.1 Automatic Read and Write of Restart Files

You can use the restart option in an automated fashion by using a combination of the `RESTART` command line in the domain scope and the `DATABASE NAME` command line in the `RESTART DATA` command block. This automated use of restart can best be explained by an example. We will use a two-processor example and assume all files will be in our current directory.

The option of automated restart will not only manage the restart files to prevent overwriting, it will also manage the results files and history files to prevent overwriting. In the example we give, we will assume our run includes a `RESULTS OUTPUT` command block with the command line

```
DATABASE NAME = rslt.e
```

to generate results files with the root file name `rslt.e`. We will also assume a run includes a `HISTORY OUTPUT` command block with the command line

```
DATABASE NAME = hist.h
```

to generate history files with the root file name `hist.h`.

For the first run in our restart sequence, we will have the command line

```
RESTART = AUTOMATIC
```

in the domain scope of our input file. In a `TIME STEPPING` command block, which is embedded in a `TIME CONTROL` command block (Section 3.1.1) in the procedure scope of our input file, we will have the command line

```
START TIME = 0.0 .
```

In the `TIME CONTROL` command block we will have the command line

```
TERMINATION TIME = 2.5E-3
```

to set the limits for the begin and end times of the first restart run. These time-related command lines should not be confused with the similarly named `START TIME` and `TERMINATION TIME` command lines that can be used in the `RESTART DATA` command block.

Finally, for the first run in our restart sequence, the `RESTART DATA` command block in our input file will be as follows:

```
BEGIN RESTART DATA PRESTO_RESTART
  DATABASE NAME = g.rsout
  AT TIME 0.0 INCREMENT = 0.25E-3
END RESTART DATA PRESTO_RESTART
```

In this block, the `DATABASE NAME` command line specifies a root file name for the restart file. The `AT TIME` command line gives the time when we will start to write the restart information and the interval at which the restart information will be written (see Section 8.3.5).

For our first run, the automatic restart option will generate the following restart files:

```
# restart files
g.rsout.2.0
g.rsout.2.1
# results files
rslt.e.2.0
rslt.e.2.1
# history files
hist.h.2.0
hist.h.2.1
```

For the above files, there are extensions on the file names that indicate we have a two-processor run. The 2.0 and 2.1 extensions associate the restart files with the corresponding individual mesh files on each processor. (If our mesh file is `mesh.g`, then our mesh files on the individual processors will be `mesh.g.2.0` and `mesh.g.2.1`.) All restart information in the above files appears at time intervals of 0.25×10^{-3}, and the last restart information is written at time 2.5×10^{-3}. We have also listed the results and history files that will be generated for this run due to the file definitions in the command blocks for the results and history files.

For the second run in our sequence of restart runs, we want to start at the previous termination time, 2.5×10^{-3}, and terminate at time 5.0×10^{-3}. We leave everything in our input file (including the `START TIME = 0.0` command line in the `TIME STEPPING` command block, the `RESTART` command line, and the `RESTART DATA` command block) the same except for the `TERMINATION TIME` command line (in the `TIME CONTROL` command block). The `TERMINATION TIME` command line will now become

```
    TERMINATION TIME = 5.0E-3 .
```

It is important to note here that the actual start time for the second run in our analysis is now set by the last time (2.5×10^{-3}) that restart information was written. The command line `START TIME = 0.0` in the `TIME STEPPING` command block is now superseded as the actual starting time for the second run by the restart commands. Any `START TIME` command line in a `TIME STEPPING` command block is still valid in terms of defining time stepping blocks (these blocks being used to set activation periods), but the restart process sets the actual start time for our analysis. This pattern of control for setting the actual start time holds for any run in our sequence of restart runs.

For the second run in our sequence of restart runs, the restart files will be from time 2.5×10^{-3} to time 5.0×10^{-3}. The restart files in our current directory after the second run will be as follows:

```
   # restart files
   g.rsout.2.0
   g.rsout.2.1
   g.rsout-s0002.2.0
   g.rsout-s0002.2.1
   # results files
   rslt.e.2.0
   rslt.e.2.1
   rslt.e-s0002.2.0
   rslt.e-s0002.2.1
   # history files
   hist.h.2.0
   hist.h.2.1
   hist.h-s0002.2.0
   hist.h-s0002.2.1
```

Notice that we have generated new restart files with a `-s0002` extension in addition to the extension associated with the individual processors. All restart information in the above files with the `-s0002` extension appears at time intervals of 0.25×10^{-3}, the restart information is written between time 2.5×10^{-3} and time 5.0×10^{-3}, and the final restart information is written at time 5.0×10^{-3}. The restart files for the first run in our sequence of restart runs, `g.rsout.2.0` and `g.rsout.2.1`, have been preserved. New results and history files have been created using the same extension, `-s0002`, as that used for the restart files. The original results and history files have been preserved.

Now, we want to do a third run in our sequence of restart runs. For the third run in our sequence of restart runs, we want to start at the previous termination time, 5.0×10^{-3}, and terminate at time 8.5×10^{-3}. We leave everything in our input file (including the `START TIME` command line, the `RESTART` command line, and the `RESTART DATA` command block) the same except for the `TERMINATION TIME` command line. The `TERMINATION TIME` command line (within the `TIME CONTROL` command block) will now become

```
TERMINATION TIME = 8.5E-3 .
```

For the third run in our sequence of restart runs, the restart files will be from time 5.0×10^{-3} to time 8.5×10^{-3}. The restart files in our current directory after the third run will be as follows:

```
# restart files
g.rsout.2.0
g.rsout.2.1
g.rsout-s0002.2.0
g.rsout-s0002.2.1
g.rsout-s0003.2.0
g.rsout-s0003.2.1
# results files
rslt.e.2.0
rslt.e.2.1
rslt.e-s0002.2.0
rslt.e-s0002.2.1
rslt.e-s0003.2.0
rslt.e-s0003.2.1
# history files
hist.h.2.0
hist.h.2.1
hist.h-s0002.2.0
hist.h-s0002.2.1
hist.h-s0003.2.0
hist.h-s0003.2.1
```

Notice that we have generated new restart files with a `-s0003` extension in addition to the extension associated with the individual processors. All restart information in the above files with the `-s0003` extension appears at time intervals of 0.25×10^{-3}, the restart information is written between time 5.0×10^{-3} and time 8.5×10^{-3}, and the final restart information is written at time 8.5×10^{-3}. The restart files for the first and second runs in our sequence of restart runs have been preserved. New results and history files have been created using the same extension, `-s0003`, as that used for the restart files. The original results and history files have been preserved.

The process just described can be continued as long as necessary. We will continue the process of generating new restart files with extensions that indicate their place in the sequence of runs.

8.3.1.2 User-Controlled Read and Write of Restart Files

You can use the restart option and select specific restart times and specific restart files to read from and write to by using a combination of the `RESTART TIME` command line in the domain scope and the `INPUT DATABASE NAME` and `OUTPUT DATABASE NAME` command line in the `RESTART DATA` command block. This "controlled" use of restart can best be explained by an example. We will use a two-processor example and assume all files will be in our current directory. In this example, we will manage the creation of new restart files so as not to overwrite existing restart files. Unlike the automated option for restart, this controlled use of restart requires that the user manage restart file names so as to prevent overwriting previously generated restart files. The same is true for the results and history files. The user will have to manage the creation of new results and history files so as not to overwrite existing results and history files. Creating new results and history files for each run in the sequence of restart runs requires changing the `DATABASE NAME` command line in the `RESULTS OUTPUT` and `HISTORY OUTPUT` command blocks. We will not show examples for use of the `DATABASE NAME` command line in the `RESULTS OUTPUT` and `HISTORY OUTPUT` command blocks here, as the actual use of the `DATABASE NAME` command line in the results and history command blocks would closely parallel the pattern we see for management of the restart file names.

For the first run in our restart sequence, we will have only a `RESTART DATA` command block in the region; there will be no restart-related command line in the domain scope of our input file. We will, however, have a

```
START TIME = 0.0
```

command line in a `TIME STEPPING` command block (within the `TIME CONTROL` command block) and a

```
TERMINATION TIME = 2.5E-3
```

command line within the `TIME CONTROL` command block to set the limits for the begin and end times. The `RESTART DATA` command block in our input file will be as follows:

```
BEGIN RESTART DATA PRESTO_RESTART
  OUTPUT DATABASE NAME = RS1.rsout
  AT TIME 0.0 INCREMENT = 0.5E-3
END RESTART DATA PRESTO_RESTART
```

For our first run, the restart option will generate the following restart files:

```
RS1.rsout.2.0
RS1.rsout.2.1
```

For the above files, the extensions on the file names indicate that we have a two-processor run. The 2.0 and 2.1 extensions associate the restart files with the corresponding individual mesh files on each processor. If our mesh file is `mesh.g`, then our mesh files on the individual processors will be `mesh.g.2.0` and `mesh.g.2.1`. All restart information in the above files appears at time intervals of 0.5×10^{-3}, and the last restart information is written at time 2.5×10^{-3}.

For the second run in our sequence of restart runs, we want to start at the previous termination time, 2.5×10^{-3}, and terminate at time 5.0×10^{-3}. To do this, we must add a

```
RESTART TIME = 2.5E-3
```

command line to the domain scope and set the termination time to 5.0×10^{-3} by using the command line

```
TERMINATION TIME = 5.0E-3
```

within the `TIME CONTROL` command block.

It is important to note here that the actual start time for the second run in our analysis is now set by the restart time set on the `RESTART TIME` command line, 2.5×10^{-3}. The command line `START TIME = 0.0` in the `TIME STEPPING` command block is now superseded as the actual starting time for the second run by the restart commands. Any `START TIME` command line in a `TIME STEPPING` command block is still valid in terms of defining time stepping blocks (these blocks being used to set activation periods), but the restart process sets the actual start time for our analysis. This pattern of control for setting the actual start time holds for any run in our sequence of restart runs.

We also must change the `RESTART DATA` command block to the following:

```
BEGIN RESTART DATA PRESTO_RESTART
  INPUT DATABASE NAME = RS1.rsout
  OUTPUT DATABASE NAME = RS2.rsout
  AT TIME 0.0 INCREMENT = 0.5E-3
END RESTART DATA PRESTO_RESTART
```

For this second run, we will read from the following files:

```
RS1.rsout.2.0
RS1.rsout.2.1
```

And we will write to the following files:

```
RS2.rsout.2.0
RS2.rsout.2.1
```

All restart information in the above output files, `RS2.rsout.2.0` and `RS2.rsout.2.1`, appears at time intervals of 0.5×10^{-3}, restart information is written from time 2.5×10^{-3} to time 5.0×10^{-3}, and the last restart information is written at time 5.0×10^{-3}. Notice that we have preserved the restart files from the first run from our restart sequence of runs because we have specifically given the input and output databases distinct names—`RS2.rsout` for the input file name and `RS1.rsout` for the output file name.

Now, we want to do a third run in our sequence of restart runs. For this third run, we want to start at time 4.5×10^{-3} and terminate at time 8.5×10^{-3}. We do not want to start at the termination time for the previous restart, which is 5.0×10^{-3}; rather, we want to start at time 4.5×10^{-3}. We change the `RESTART TIME` command line to

```
RESTART TIME = 4.5E-3
```

and the `TERMINATION TIME` command line within the `TIME CONTROL` command block to

```
TERMINATION TIME = 8.5E-3 .
```

And we change the `RESTART DATA` command block to the following:

```
BEGIN RESTART DATA PRESTO_RESTART
  INPUT DATABASE NAME = RS2.rsout
  OUTPUT DATABASE NAME = RS3.rsout
  AT TIME 0.0, INCREMENT = 0.5E-3
END RESTART DATA PRESTO_RESTART
```

For this third run, we will read from the following files:

```
RS2.rsout.2.0
RS2.rsout.2.1
```

And we will write to the following files:

```
RS3.rsout.2.0
RS3.rsout.2.1
```

All restart information in the above output files, `RS3.rsout.2.0` and `RS3.rsout.2.1`, appears at time intervals of 0.5×10^{-3}, restart information is written from time 4.5×10^{-3} to time 8.5×10^{-3}, and the last restart information is written at time 8.5×10^{-3}. Notice that we have preserved all restart files from previous runs in our restart sequence of runs because we have specifically given the input and output databases distinct names for this third run.

8.3.1.3 Overwriting Restart Files

If you use the `RESTART TIME` command line in conjunction with the `DATABASE NAME` command line, you will overwrite restart information (unless you have included an `OVERWRITE` command line set to `ON`). As indicated previously, you will probably want to have a restart file (or files in the case of parallel runs) associated with each run in a sequence of restart runs. The example in this section shows how to overwrite restart files if that is an acceptable approach for a particular analysis.

For our first run, we will set a termination time of 1.0×10^{-3} with the command line

```
TERMINATION TIME = 1.0E-3
```

and set the `RESTART DATA` command block as follows:

```
BEGIN RESTART DATA
  DATABASE NAME = RS.out
  AT TIME 0.0 INTERVAL = 0.25E-3
END RESTART DATA
```

Our first run will generate the following restart files:

```
RS.out.2.0
RS.out.2.1
```

All restart information in the above output files, `RS.out.2.0` and `RS.out.2.1`, appears at time intervals of 0.25×10^{-3}, restart information is written from time 0.0 to time 1.0×10^{-3}, and the last restart information is written at time 1.0×10^{-3}.

Suppose for our second run we set the termination time to 2.0×10^{-3} with the command line

```
TERMINATION TIME = 2.0E-3
```

and add the command line

```
RESTART TIME = 1.0E-3
```

to the domain scope. We leave the `RESTART DATA` command block unchanged.

For our second run, restart information is read from the files `RS.out.2.0` and `RS.out.2.1`. These files are then overwritten with new restart information beginning at time 1.0×10^{-3}. The files `RS.out.2.0` and `RS.out.2.1` will have restart information beginning at time 1.0×10^{-3} in intervals of 0.25×10^{-3}. The restart information will terminate at time 2.0×10^{-3}.

Now we want to do a third run with a termination time of 3.0×10^{-3}. We change the termination time by using the command line

```
TERMINATION TIME = 3.0E-3 .
```

And we change the `RESTART TIME` command line so that it is now

```
RESTART TIME = 3.0E-3 .
```

For our third run, restart information is read from the files `RS.out.2.0` and `RS.out.2.1`. These files are then overwritten with new restart information beginning at time 2.0×10^{-3}. The files `RS.out.2.0` and `RS.out.2.1` will have restart information beginning at time 2.0×10^{-3} in intervals of 0.25×10^{-3}. The restart information will terminate at time 3.0×10^{-3}.

8.3.1.4 Recovering from a Corrupted Restart

Suppose you are using the automated option for restart and a system crash occurs when the restart file is being written. The restart file contains a corrupted entry for one of the restart times. In this case, you can continue using the automated option for restart. Restart will detect the corrupted entry and then find an entry previous to the corrupted entry that can be used for restart. This previous entry should be the entry prior to the corrupted entry unless something unusual has occurred. If the first intact restart entry is not the previous entry, restart continues to back up until an intact restart entry is found.

You could do a manual recovery. The manual recovery requires the use of a `RESTART TIME` command line to select some intact restart entry. You will have to use the `INPUT DATABASE NAME` and `OUTPUT DATABASE NAME` command lines to avoid overwriting previous restart files (see Section 8.3.1.2). You will also have to change file names in the results and history command blocks to avoid overwriting previous results and history files. Once you have done the manual recovery, you could then revert to the automatic restart option.

8.3.2 Overwrite Command in Restart

```
OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
  (ON|TRUE|YES)
```

The `OVERWRITE` command line can be used to prevent the overwriting of existing restart files. The use of the automatic read and write option for restart files as described in Section 8.3.1.1 does not require the `OVERWRITE` command line. The automatic read and write option adds extensions to file names and prevents the overwriting of any existing restart files. For the case of a user-controlled read and write of restart files (Section 8.3.1.2), however, the `OVERWRITE` command line is useful for preventing the overwriting of restart files. If the `OVERWRITE` command line is set to `OFF`, `FALSE`, or `NO`, then existing restart files will not be overwritten. Execution of the code will terminate before existing restart files are overwritten. The default option is to overwrite existing restart files. If the `OVERWRITE` command line is not included, or the command line is set to `ON`, `TRUE`, or `YES`, then existing files can be overwritten.

8.3.3 Set Begin Time for Restart Writes

```
START TIME = <real>restart_start_time
```

Using the `START TIME` command line, you can write restarts to the restart file beginning at time `restart_start_time`. No restarts will be written before this time. If other commands set times for restarts (`AT TIME`, `ADDITIONAL TIMES`) that are less than `restart_start_time`, those times will be ignored, and restarts will not be written at those times.

8.3.4 Adjust Interval for Time Steps

```
TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
```

This command line is used to specify that the restarts will be written at exactly the times specified. To hit the restart times exactly in an explicit transient dynamics code, it is necessary to adjust the time step as the time approaches a restart time. The integer value `steps` in the `TIMESTEP ADJUSTMENT INTERVAL` command line specifies the number of time steps to look ahead in order to adjust the time step.

If this command line does not appear, then restarts are written at times closest to the specified restart times.

8.3.5 Restart Interval Specified by Time Increment

```
AT TIME <real>time_begin INCREMENT = <real>time_increment_dt
```

At the time specified by `time_begin`, restarts will be written every time increment given by the real value `time_increment_dt`.

8.3.6 Additional Times for Restart

```
ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any restart times specified by the command line in Section 8.3.5, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional restart times.

8.3.7 Restart Interval Specified by Step Increment

```
AT STEP <integer>step_begin INCREMENT =
  <integer>step_increment
```

At the step specified by `step_begin`, restarts will be written every step increment given by the integer value `step_increment`.

8.3.8 Additional Steps for Restart

```
ADDITIONAL STEPS = <integer>output_step1
  <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.3.7, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of additional restart steps.

8.3.9 Set End Time for Restart Writes

```
TERMINATION TIME = <real>termination_time_value
```

Restarts will not be written to the restart file after time `termination_time_value`. If other commands set times for restarts (`AT TIME`, `ADDITIONAL TIMES`) that are greater than `termination_time_value`, those times will be ignored, and restarts will not be written at those times.

8.3.10 Overlay Count

```
    OVERLAY COUNT = <integer>overlay_count
```

The `OVERLAY COUNT` command line specifies the number of restart output times that will be overlaid on top of the current step before advancing to the next step. For example, suppose that we set the `overlay_count` parameter to 2, and we request that restart information be written every 0.1 second. At time 0.1 second, restart step 1 will be written to the output restart database. At time 0.2 second, restart information will be written over the step 1 information, which originally contained restart information at 0.1 second. At time 0.3 second, restart information will be written over the step 1 information, which last contained information at 0.2 second. At time 0.4 second, we will now write step 2 to the output restart database (step 1 has already been written over twice). At time 0.5 second, restart information will be written over the step 2 information, which originally contained information at 0.4 second. At time 0.6 second, restart information will be written over the step 2 information, which last contained information at 0.5 second. At time 0.7 second, restart step 3 will be written to the output restart database (step 2 has already been written over twice). This pattern continues so that we would build up a sequence of restart information at times 0.3, 0.6, 0.9, . . . second until we reach the termination time for the problem. If there was a problem during the analysis, the last step on the output restart database would be whatever had last been written to the database. If, for example, we had set our termination time to 1.0 second and a problem occurred after restart information had been written at 0.7 second but before we completed the time step at 0.8 second, then the last information on the output restart database would be at 0.7 second.

You can use the `OVERLAY COUNT` command line in conjunction with a `CYCLE COUNT` command line. For a description of the `CYCLE COUNT` command line and its use with the `OVERLAY COUNT` command line, see Section 8.3.11.

8.3.11 Cycle Count

```
    CYCLE COUNT = <integer>cycle_count
```

The `CYCLE COUNT` command line specifies the number of restart steps that will be written to the output restart database before previously written steps are overwritten. For example, suppose we set the `cycle_count` parameter to 5, and we request that restart information be written every 0.1 second. The restart system will write information to the output restart database at times 0.1, 0.2, 0.3, 0.4, and 0.5 second. At time 0.6 second, the information at step 1, originally written at time 0.1 second, will be overwritten with information at time 0.6 second. At time 0.7 second, the information at step 2, originally written at time 0.2 second, will be overwritten with information at time 0.7 second. At time 0.8 second, the output restart database will contain restart information at times 0.6, 0.7, 0.8, 0.4, and 0.5 second. Time will not necessarily be monotonically increasing on a database that uses a `CYCLE COUNT` command line.

If you only want the last step available on the output restart database, set `cycle_count` equal to 1.

The `CYCLE COUNT` and `OVERLAY COUNT` command lines can be used at the same time. For this example, we will combine our example with an overlay count of 2 as given in Section 8.3.10 with our example of a cycle count of 5 as given in this section (Section 8.3.11). Information is written to the output restart database time step every 0.1 second. The output times at which information is written to the output restart database are 0.1, 0.2, 0.3, . . . second. Each of these times corresponds to an output step. Time 0.1 second corresponds to output step 1, time 0.2 second corresponds to output step 2, time 0.3 corresponds to output step 3, and so forth. An output time of $n \times 0.1$ corresponds to output step n. The overlay command will result in information at time 0.3, 0.6, 0.9, 1.2, and 1.5 seconds written as steps 1, 2, 3, 4, and 5 on the output restart database. For times greater than 1.6 seconds, the cycle command will now take effect because we have five steps written on the output restart database. Information at times 1.6, 1.7, and 1.8 seconds will now overwrite the information at step 1, which had information at time 0.3 second. Information at times 1.9, 2.0, and 2.1 seconds will now overwrite the information at step 2, which had information at time 0.6 second. For any output step n, its position, step number n_s, in the restart output database is as follows:

```
$if n_s \neq 0$
    $n_s = int(n/(n_o + 1))\% n_c$
$else$
    $n_s = n_c$
$end$
```

In the above equations, n_c is the cycle count, and n_o is the overlay count. The expression $int(n/(n_o + 1))$ produces an integer arithmetic result. For example, if n is 4 and n_o is 2, then we have 4 divided by 3, and the integer arithmetic result is 1 (any fractional remainder is discarded). The operator % is the modulus operator; the modulus operator gives the modulus of its first operand with respect to its second operand, i.e., it produces the remainder of dividing the first operand by the second operand. The result of 1 % 5 is 1, for example.

8.3.12 Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

In an analysis with multiple regions, it can be difficult to synchronize output such as restart files. To help synchronize output for analyses with multiple regions, you can define an `OUTPUT SCHEDULER` command block at the domain level. The scheduler can then be referenced in the `RESTART DATA` command block via the `USE OUTPUT SCHEDULER` command line. The string `scheduler_name` must match a name used in an `RESTART DATA` command block. See Section 8.4 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.3.13 Write Restart If System Error Encountered

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The `OUTPUT ON SIGNAL` command line is used to initiate the writing of a restart file when the system encounters a type of system error. Only one error type in the list of error types should be entered for this command line. Generally, these system errors cause the code to terminate before the code can add any current restart output (restart output past the last restart output time step) to the restart file. If the code encounters the specified type of error during execution, a restart file will be written before execution is terminated.

This command line can also be used to force the writing of a restart file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

`kill -s SIGUSR2` *pid*

to terminate execution and force the writing of a results file. In the above system command line, *pid* is the process identifier, which is an integer.

The most useful application of the command line is to send a signal via a system command line to write a restart file. Note that the `OUTPUT ON SIGNAL` command line is primarily a debugging tool for code developers.

8.4 Output Scheduler

In an analysis with multiple regions, it can be difficult to synchronize output such as results files, history files, and restart files. To help synchronize output for analyses with multiple regions, you can define an `OUTPUT SCHEDULER` command block at the domain level. This scheduler can then be referenced in several places:

- The scheduler can be referenced in the `RESULTS OUTPUT` command block to control the output of results information.
- The scheduler can be referenced in the `HISTORY OUTPUT` command block to control the output of history information.
- The scheduler can be referenced in the `RESTART DATA` command block to control the writing of restart files.

In summary, the `OUTPUT SCHEDULER` command block is defined in the domain scope. The scheduler is referenced by a `USE OUTPUT SCHEDULER` command line that can appear in a `RESULTS OUTPUT`, `HISTORY OUTPUT`, and `RESTART DATA` command block. Section 8.4.1 describes the `OUTPUT SCHEDULER` command block, and Section 8.4.2 illustrates how this block is referenced with the `USE OUTPUT SCHEDULER` command line.

8.4.1 Output Scheduler Command Block

```
BEGIN OUTPUT SCHEDULER <string>scheduler_name
  START TIME = <real>output_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
END [OUTPUT SCHEDULER <string>scheduler_name]
```

An output scheduler is defined with a command block in the domain scope. The `OUTPUT SCHEDULER` command block begins with the input line

```
BEGIN OUTPUT SCHEDULER <string>scheduler_name
```

and is terminated with the line

```
END OUTPUT SCHEDULER <string>scheduler_name ,
```

where `scheduler_name` is a user-defined name for the command block. All the normal scheduling command lines are valid in an `OUTPUT SCHEDULER` command block.

8.4.1.1 Set Begin Time for Output Scheduler

```
START TIME = <real>output_start_time
```

Using the `START TIME` command line, you can set the start time for a scheduler beginning at time `output_start_time`. The scheduler will not take effect before this time. If other commands set times for scheduling (`AT TIME`, `ADDITIONAL TIMES`) that are less than `output_start_time`, those times will be ignored.

8.4.1.2 Adjust Interval for Time Steps

```
TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
```

This command line is used to specify that, when the scheduler is in effect, output will be at exactly the times specified. To hit the output times exactly in an explicit, transient dynamics code, it is necessary to adjust the time step as the time approaches an output time. The integer value `steps` in the `TIMESTEP ADJUSTMENT INTERVAL` command line specifies the number of time steps to look ahead in order to adjust the time step.

If this command line does not appear, output occurs at times closest to the specified output times.

8.4.1.3 Output Interval Specified by Time Increment

```
AT TIME <real>time_begin INCREMENT = <real>time_increment_dt
```

At the time specified by `time_begin`, output will be scheduled at every time increment given by the real value `time_increment_dt`.

8.4.1.4 Additional Times for Output

```
ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.4.1.3, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional output times.

8.4.1.5 Output Interval Specified by Step Increment

```
AT STEP <integer>step_begin INCREMENT =
  <integer>step_increment
```

At the step specified by `step_begin`, output will be scheduled at every step increment given by the integer value `step_increment`.

8.4.1.6 Additional Steps for Output

```
 ADDITIONAL STEPS = <integer>output_step1
   <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.4.1.5, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of additional output steps.

8.4.1.7 Set End Time for Output Scheduler

```
TERMINATION TIME = <real>termination_time_value
```

Using the `TERMINATION TIME` command line, you can set the termination time for a scheduler beginning at time `termination_time_value`. The scheduler will not be in effect after this time. If other commands set times for scheduling (`AT TIME`, `ADDITIONAL TIMES`) that are greater than `termination_time_value`, those times will be ignored by the scheduler.

8.4.2 Example of Using the Output Scheduler

Once an output scheduler has been defined via the `OUTPUT SCHEDULER` command block, it can be used by inserting a `USE OUTPUT SCHEDULER` command line in any of the following command blocks: `RESULTS OUTPUT`, `HISTORY OUTPUT`, and `RESTART DATA`. The following paragraph provides an example of using output schedulers.

In the domain scope, we define two output schedulers, `Timer` and `Every_Step`:

```
BEGIN OUTPUT SCHEDULER Timer
  AT TIME 0.0 INCREMENT = 10.0e-6
  TIME STEP ADJUSTMENT INTERVAL = 4
END OUTPUT SCHEDULER Timer
#
```

```
BEGIN OUTPUT SCHEDULER Every_Step
  AT STEP 0 INCREMENT = 1
END OUTPUT SCHEDULER Every_Step
```

With the `USE OUTPUT SCHEDULER` command, we reference the scheduler named `Timer` for results output:

```
BEGIN RESULTS OUTPUT Out_Region_1
  .
  USE OUTPUT SCHEDULER Timer
  .
END RESULTS OUTPUT Out_Region_1
```

With the `USE OUTPUT SCHEDULER` command, we reference the scheduler named `Every_STEP` for history output:

```
BEGIN HISTORY OUTPUT Out_Region_2
  .
  USE OUTPUT SCHEDULER Every_Step
  .
END HISTORY OUTPUT Out_Region_2
```

8.5 Registered Variables

This section lists commonly used registered variables that the user can select as output to the results file and the history file. The first part of this section lists global, nodal, and element registered variables. The second part of this section lists registered variables associated with material models.

8.5.1 Global, Nodal, and Element Registered Variables

This section lists commonly used global, nodal, and element registered variables. The registered variables are presented in tables based on use, as follows:

- Table 8.5 Variables Registered on Nodes (Variable and Type)
- Table 8.6 Element Variables Registered for All Elements
- Table 8.7 Element Variables Registered for Energy-Dependent (Equation-of-State) Elements
- Table 8.8 Element Variables Registered for Solid Elements
- Table 8.9 Element Variables Registered for Membranes
- Table 8.10 Nodal Variables Registered for Shells
- Table 8.11 Element Variables Registered for Shells
- Table 8.12 Element Variables Registered for Truss
- Table 8.13 Element Variables Registered for Beam
- Table 8.14 Global Registered Variables
- Table 8.15 Nodal Variables Registered for Spot Welds

The tables provide the following information about each registered variable:

Variable Name. This is the string that will appear on the `GLOBAL VARIABLES`, `NODE VARIABLES`, or `ELEMENT VARIABLES` command line.

Type. This is the variable's type. The various types are denoted with the labels `Integer`, `Real`, `Vector_2D`, `Vector_3D`, `SymTen33`, and `FullTen36`. The type `Integer` indicates the registered variable is an integer; the type `Real` indicates the registered variable is a real. The type `Vector_2D` indicates the registered variable type is a two-dimensional vector. The type `Vector_3D` indicates the registered variable is a three-dimensional vector. For a three-dimensional vector, the variable quantities will be output with suffixes of `_x`, `_y`, and `_z`. For example, if the registered variable displacement is requested to be output as `displ`, the components of the displacement vector on the results file will be `displ_x`, `displ_y`, and `displ_z`. The type `SymTen33` indicates the registered variable is a symmetric 3 × 3 tensor. For a 3 × 3 symmetric tensor, the variable quantities will be output with suffixes of `_xx`, `_yy`, `_zz`, `_xy`, `_yz`, and `_zx`. For example, if the registered variable `rotated_stress` is requested for output as `stress`, the components of the stress tensor on the results file will be `stress_xx`, `stress_yy`, `stress_zz`, `stress_xy`, `stress_yz`, and `stress_zx`. The type `FullTen36` is a full 3 × 3 tensor with three diagonal terms and six off-diagonal terms.

The tables of registered variables follow.

Table 8.5: Variables Registered on Nodes (Variable and Type)

Variable Name	**Type**	**Comments**
model_coordinates	Vector_3D	Original coordinates of nodes.
coordinates	Vector_3D	Current coordinates of nodes.
displacement	Vector_3D	Total displacement.
displacement_increment	Vector_3D	Displacement increment at current time step.
velocity	Vector_3D	
acceleration	Vector_3D	
force_internal	Vector_3D	
force_external	Vector_3D	
force_hourglass	Vector_3D	
force_contact	Vector_3D	
reactions	Vector_3D	
moment_reactions	Vector_3D	
mass	Real	

Table 8.6: Element Variables Registered for All Elements

Variable Name	**Type**	**Comments**
elem_time_step	Real	Critical time step for the element. The element in the model with the smallest time step controls the analysis time step.
element_mass	Real	

Table 8.7: Element Variables Registered for Energy-Dependent ("Equation-of-State") Elements

Variable Name	**Type**	**Comments**
stress	SymTen33	
rotated_stress	SymTen33	
stretch	SymTen33	
rotation	FullTen36	
element_density	Real	
sound_speed	Real	
specific_internal_energy	Real	
artificial_viscosity	Real	
volume	Real	
shrmod	Real	
dilmod	Real	

Table 8.8: Element Variables Registered for Solid Elements

Variable Name	**Type**	**Comments**
stress	SymTen33	
rotated_stress	SymTen33	
stretch	SymTen33	
rotation	FullTen36	
log_strain	SymTen33	
volume	Real	
shrmod	Real	
dilmod	Real	

Table 8.9: Element Variables Registered for Membranes

Variable Name	**Type**	**Comments**
memb_stress	SymTen33	
element_area	Real	
element_thickness	Real	

Table 8.10: Nodal Variables Registered for Shells

Variable Name	Type	Comments
rotational_displacement	Vector_3D	
rotational_velocity	Vector_3D	
rotational_acceleration	Vector_3D	
moment_internal	Vector_3D	
moment_external	Vector_3D	
rotational_mass	Real	

Table 8.11: Element Variables Registered for Shells

Variable Name	Type	Comments
memb_stress	SymTen33	Stress at midplane in global X, Y, and Z coordinates.
bottom_stress	SymTen33	Stress at bottom integration point in global X, Y, and Z coordinates.
top_stress	SymTen33	Stress at top integration point in global X, Y, and Z coordinates.
strain	SymTen33	Integrated strain at midplane in local shell coordinate system.
element_area	Real	
element_thickness	Real	

Table 8.12: Element Variables Registered for Truss

Variable Name	Type	Comments
truss_init_length	Real	
truss_stretch	Real	
truss_stress	Real	
truss_strain_incr	Real	
truss_force	Real	

Table 8.13: Element Variables Registered for Beam

Variable Name	**Type**	**Comments**
beam_strain_inc	Vector_2D	Thirty-two strain increment values are output. Some values may be zero depending on section. Axial strains are 01, 03, 05, . . . Shear strains are 02, 04, 06, . . . See Section 5.2.4 for more details.
beam_stress	Vector_2D	Thirty-two stress values are output. Some values may be zero depending on section. Axial stresses are 01, 03, 05, . . . Shear stresses are 02, 04, 06, . . . See Section 5.2.4 for more details.
beam_stress_axial	Real	Sixteen axial stress values. Some may be zero depending on section.
beam_stress_shear	Real	Sixteen shear stress values. Some may be zero depending on section.
beam_axial_force	Real	Axial force at midpoint.
beam_transverse_force_s	Real	Transverse shear in s-direction at midpoint.
beam_transverse_force_t	Real	Transverse shear in t-direction at midpoint.
beam_moment_r	Real	Torsion at midpoint.
beam_moment_s	Real	Moment about s-direction at midpoint.
beam_moment_t	Real	Moment about t-direction at midpoint.

Table 8.14: Global Registered Variables

Variable Name	**Type**	**Comments**
timestep	Real	
KineticEnergy	Real	
MomentumX	Real	Momentum in global X-direction.
MomentumY	Real	Momentum in global Y-direction.
MomentumZ	Real	Momentum in global Z-direction.

Table 8.15: Nodal Variables Registered for Spot Welds

Variable Name	**Type**	**Comments**
SPOT_WELD%parametric_coords	Vector_2D	Coordinates of node on face.
SPOT_WELD&norm_force_at_death	Real	Value of force normal to face when spot-weld breaks.
SPOT_WELD%tang_force_at_death	Real	Value of force tangential to face when spot-weld breaks.
SPOT_WELD%death_flag	Integer	0 = alive, 1 = dead.
SPOT_WELD%scale_factor	Real	Nodal influence area of current node.
SPOT_WELD%norm_displacement	Real	Current displacement of weld normal to face.
SPOT_WELD%tang_displacement	Real	Current displacement of weld tangential to face.

8.5.2 Registered Variables for Material Models

State Specification. It is possible to output the state variables from the material models. For the elastic material model, there are no state variables. For other models, including the energy-dependent models (Mie-Gruneisen, Mie-Gruneisen Power-Series, JWL, and ideal gas), the associated registered variables are accessed as any other registered variables.

To get the state variables for materials, you should use the `ELEMENT VARIABLES` command line in the `RESULTS OUTPUT` command block in the form

```
ELEMENT VARIABLES = MAT%material_name ,
```

where `material_name` can be some material model such as `elastic_plastic`, `power_law_hardening`, `foam_plasticity`, or `orthotropic_rate`.

For example, if you wanted to get all the state variables for the elastic-plastic material model, you would use

```
ELEMENT VARIABLES = MAT%elastic_plastic
```

or

```
ELEMENT VARIABLES = MAT%elastic_plastic(:)  .
```

In the preceding command line, the parentheses indicate a subset of the state variables, and the colon should be read as "through." The notation (:) implies all the state variables; the notion (`3:7`) would imply state variables 3 through 7.

Depending on the material type, there may be several hundred material state variables. A subset of state variables may be output using a FORTRAN-like array syntax. For example, to output the equivalent plastic strain for the elastic-plastic material and rename it to something meaningful, use

```
ELEMENT VARIABLES = MAT%elastic_plastic(1) as eqps .
```

To output a six-entry symmetric tensor subset of the orthotropic rate material, use

```
ELEMENT VARIABLES = MAT%orthotropic_rate(2:7) .
```

Finally, if a scattered set of material variables is desired, the variables may be selected by using the | array operator, as in the command

```
ELEMENT VARIABLES = MAT%foam_plasticity(1|4:6|9) .
```

This command would output state variables 1, 4, 5, 6, and 9 of the foam plasticity model.

Tables 8.16 through 8.18 list the indexes and corresponding descriptions of the state variables for some of the more commonly used material models. To find the state variable number that corresponds to a quantity of interest for other material models, consult with William Scherzinger (wmscher@sandia.gov), Richard Koteras (jrkoter@sandia.gov), Arne Gullerud (asgulle@sandia.gov), or Nathan Crane (nkcrane@sandia.gov).

Table 8.16: State Variables for Elastic-Plastic Material Model

Index	**Variable Description**
1	Equivalent plastic strain
2	Back stress – xx component
3	Back stress – yy component
4	Back stress – zz component
5	Back stress – xy component
6	Back stress – yz component
7	Back stress – zx component

Table 8.17: State Variables for Elastic-Plastic Power-Law Hardening Material Model

Index	**Variable Description**
1	Equivalent plastic strain
2	Radius of yield surface

Table 8.18: State Variables for Foam Plasticity Material Model

Index	**Variable Description**
1	Iterations
2	Volumetric strain
3	Phi
4	Equivalent plastic strain
5	A
6	B

8.6 References

1. The eXtensible Data Model and Format (XDMF) . . . . http://www.arl.hpc.mil/ice/XdmfUser.html (Accessed April 27, 2007).

2. Sjaardema, G. D. *Overview of the Sandia National Laboratories Engineering Analysis Code Access System*, SAND92-2292. Albuquerque, NM: Sandia National Laboratories, January 1993.

Intentionally Left Blank

Chapter 9

User Subroutines

This chapter describes the use of user-defined subroutines in Presto. In the introductory part of Chapter 9, we first describe, in general, possible applications for the user subroutine functionality in Presto. Then, again in general, we describe the various "pieces" and steps that are required by the user to implement a user subroutine. Subsequently, we focus on various aspects of implementing the user subroutine functionality. Section 9.1 describes the details of the user subroutine. Section 9.2 describes the command lines associated with user subroutines that will appear in a Presto input file. In Section 9.3, we explain how to build and use a version of Presto that incorporates your user subroutine. Finally, Section 9.4 provides examples of actual user subroutines, and Section 9.5 lists some subroutines that are now in the standard user library.

Applications. User subroutines are primarily intended as complex function evaluators that are to be used in conjunction with existing Presto capability (boundary conditions, element death, user output, etc.). For example, suppose we want to have a prescribed displacement boundary condition applied to a set of nodes, and we want the displacement at each node to vary with both time and spatial location of the node. The standard function option associated with the prescribed direction displacement boundary condition in Presto only allows for time variation; i.e., at any given time, the direction and the magnitude of the displacement at each node, regardless of the spatial location of the node, are the same. If we wanted to have a spatial variation of the displacement field in addition to the time variation, it would be necessary to implement a user subroutine for the prescribed direction displacement boundary condition. Other examples of possible uses of user subroutines are as follows:

- Element death is determined by a complex function based on a set of physical parameters and element stress.
- The user wants to compute the total contact force acting on a given surface.

- Element stress information must be transformed to a local coordinate system so that the stress values will be meaningful.
- An aerodynamic pressure based on velocity and surface normal is applied to a specified surface.

Some capability exists for using mesh connectivity. It is possible to compute an element quantity based on values at the element nodes.

Some difficulties might occur in parallel applications. If computations for element A depend on quantities in element B and elements A and B are on different processors, then the computations for A may not have access to quantities in element B. For most computations in user subroutines, however, this should not be a problem.

Implementing completely new capabilities, particularly if these capabilities involve parallel computing, may be difficult or impossible with user subroutines.

General Pieces and Steps. A number of pieces and steps are required to make use of user subroutines. Here, we present a brief description of the pieces and steps that a user will need for user subroutines without going into detail. The details are discussed in later parts of this chapter.

1. You must first determine whether your application fits in the user subroutine format. This can be done by considering the above requirements and examining the description of commands for functionality in Presto. For example, the basic kinematic boundary conditions and force conditions allow for the use of user subroutines. The description of these commands includes a discussion of how a user subroutine could be applied and what command line will invoke a user subroutine.
2. If you determine that your application can make use of the user subroutine functionality in Presto, you will then need to write the subroutine. The parts of the subroutine that interface to Presto have specified formats. The details of these interfaces are described in later sections. One part of the subroutine with a specified format is the call list. Other parts of the subroutine with a specified format are code that will do the following:

   - Read parameters from the Presto input file
   - Access a variety of information—field variables, analysis time, etc.—from Presto
   - Store computed quantities

   Parameters are values they may be passed from the Presto input file to the user subroutine. Suppose that the spatial variation for some quantity in the user subroutine uses some characteristic length and the user wishes to examine results generated by using several different values of the characteristic length. By setting up the characteristic length as a parameter, the value for the parameter in the user subroutine can easily be changed by changing the value for the parameter in the input file. This lets the user change the value for a variable inside the user subroutine without having to recompile the user subroutine.

The portion of your subroutine not built on the Presto specifications will reflect your specific application. The code to implement your application may include a loop over nodes that prescribes a displacement based on the current time for the analysis and the spatial location of the node.

3. After you write the user subroutine, you will need to have a command line in your input file that tells Presto you want to use the user subroutine you have written. For example, if your user subroutine is a specialized prescribed displacement boundary condition, then inside a `PRESCRIBED DISPLACEMENT` command block, you will have a command line of the form

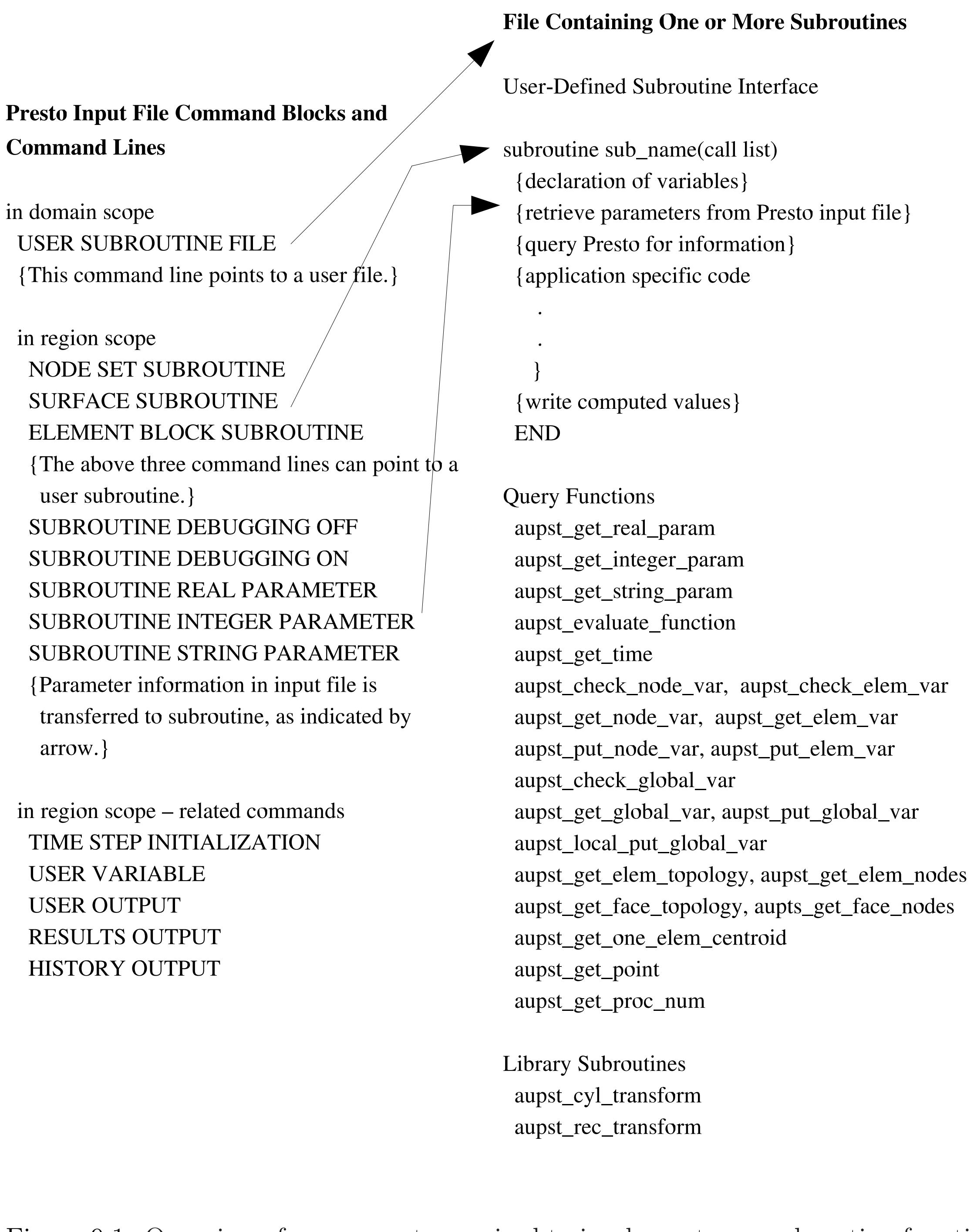
   ```
   NODE SET SUBROUTINE = <string>subroutine_name
   ```

   that provides the name of your user subroutine.

4. Following the invocation of the user subroutine, there may be command lines for various parameters associated with the user subroutine. There may also be some additional command lines in other sections of the code required for your application. For example, you may have to add command lines in the region scope that will create an internal variable associated with a computed quantity so that the computed quantity can be written to the results file.

5. Once you have constructed the user subroutine, which is a FORTRAN file, and the Presto input file, you can build an executable version of Presto that will run your user subroutine. Your Presto run will then incorporate the functionality you have created in your user subroutine.

Figure 9.1 presents a very high-level overview of the various components that work together to implement the user subroutine functionality. The two main components needed for user subroutines, which are commands in the Presto input file and the actual user subroutine, are represented by the two columns in Figure 9.1.



Presto Input File Command Blocks and Command Lines

```
in domain scope
  USER SUBROUTINE FILE
  {This command line points to a user file.}

  in region scope
    NODE SET SUBROUTINE
    SURFACE SUBROUTINE
    ELEMENT BLOCK SUBROUTINE
    {The above three command lines can point to a
      user subroutine.}
    SUBROUTINE DEBUGGING OFF
    SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER
    SUBROUTINE INTEGER PARAMETER
    SUBROUTINE STRING PARAMETER
    {Parameter information in input file is
      transferred to subroutine, as indicated by
      arrow.}

  in region scope – related commands
    TIME STEP INITIALIZATION
    USER VARIABLE
    USER OUTPUT
    RESULTS OUTPUT
    HISTORY OUTPUT
```

File Containing One or More Subroutines

```
User-Defined Subroutine Interface

subroutine sub_name(call list)
  {declaration of variables}
  {retrieve parameters from Presto input file}
  {query Presto for information}
  {application specific code
     .
     .
     }
  {write computed values}
  END

Query Functions
  aupst_get_real_param
  aupst_get_integer_param
  aupst_get_string_param
  aupst_evaluate_function
  aupst_get_time
  aupst_check_node_var,  aupst_check_elem_var
  aupst_get_node_var,  aupst_get_elem_var
  aupst_put_node_var, aupst_put_elem_var
  aupst_check_global_var
  aupst_get_global_var, aupst_put_global_var
  aupst_local_put_global_var
  aupst_get_elem_topology, aupst_get_elem_nodes
  aupst_get_face_topology, aupts_get_face_nodes
  aupst_get_one_elem_centroid
  aupst_get_point
  aupst_get_proc_num

Library Subroutines
  aupst_cyl_transform
  aupst_rec_transform
```

Figure 9.1: Overview of components required to implement user subroutine functionality, excluding compilation and execution commands.

9.1 User Subroutines: Programming

Currently, user subroutines are only supported in FORTRAN 77. Any subroutine that can be compiled with a FORTRAN 77 compiler on the target execution machine can be used. The user should be aware that some computers support different FORTRAN language extensions than others. (In the future, other languages such as FORTRAN 90, C, and C++ may be supported.)

User subroutine variable types must interface directly with the matching variable types used in the main Presto code. Thus, the FORTRAN 77 subroutines should use only integer, double precision, or character types for any data used in the interface or in any query function. Using the wrong data type may yield unpredictable results. The methods used to pass character types from Presto to FORTRAN user subroutines can be machine-dependent, but generally this functionality works quite well.

The basic structure for the user subroutine is as follows:

```
subroutine sub_name(call list)
{declaration of variables}
{retrieve parameters from Presto input file}
{query Presto for information}
{application-specific code
   .
   .
}
{write computed values}
END
```

In general, the user will begin the subroutine with variable declarations. After the variable declarations, the user can then query the Presto input file for parameters. Additional Presto information such as field variables or element topology can then be retrieved from Presto. Once the user has collected all the information for the application, the application-specific portion of the code can be written. After the application-specific code is complete, the user may store computed values.

Section 9.1.1 through Section 9.1.3 describe in detail the format for the interfaces to Presto that will allow the user to make the subroutine call, retrieve information from Presto, and write computed values. In these sections, mesh entities can be a node, an element face, or an element.

9.1.1 Subroutine Interface

The following interface is used for all user subroutines:

```
subroutine sub_name(int  num_objects,
          int  num_values,
          real evaluation_time,
          int  object_ids[],
          real output_values[],
          int  output_flags[],
          int  error_code)
```

The name of the user subroutine, `sub_name`, is selected by the user. Avoid names for the subroutine that are longer than 10 characters. This may cause build problems on some systems.

A detailed description of the input and output parameters is provided in Table 9.1 and Table 9.2.

Table 9.1: Subroutine Input Parameters

Input Parameter	**Data Type**	**Parameter Description**
num_objects	Integer	Number of input mesh entities. For example, if the subroutine is a node set subroutine, this would be the number of nodes on which the subroutine will operate.
num_values	Integer	Number of return values. This is the number of values per mesh entity.
evaluation_time	Real	Time at which the subroutine should be evaluated. This may vary slightly from the current analysis time. Velocities for example are evaluated one-half time step ahead.
object_ids (num_objects)	Integer	Array of mesh-entity identification numbers. The array has a length of num_objects. The input numbers are the global numbers of the input objects. The object identification numbers can be used to query information about a mesh entity.

Table 9.2: Subroutine Output Parameters

Output Parameter	**Data Type**	**Parameter Description**
output_values (num_values, num_objects)	Integer	Array of output values computed by the subroutine. The number of output values will be either the number of mesh entities or some multiple of the number of mesh entities. For example, if there were six nodes (num_objects equals 6) and one value was to be computed per node, the length of output_values would be 6. Similarly, if there were six nodes (num_objects equals 6) and three values were to be computed for each node (as for acceleration, which has X-, Y-, and Z-components), the length of output_values would be 18.
output_flags (num_objects)	Integer	Array of returned flags for each set of data values. When used, this array will generally have a length of num_objects. The usage of the flags depends on subroutine type; the flags are currently used only for element death and for kinematic boundary conditions. For the kinematic boundary conditions (displacement, velocity, acceleration) a flag of –1 means ignore the constraint, a flag of 0 means set the absolute constraint value, and a flag of 1 means set the constraint with direction and distance.
error_code	Integer	Error code returned by the user subroutine. A value of 0 indicates no errors. Any value other than zero is an error. If the return value is nonzero, Presto will report the error code and terminate the analysis.

9.1.2 Query Functions

Presto follows a design philosophy for user subroutines that a minimal amount of information should be passed through the call list. Additional information may be queried from within the subroutine. A user subroutine may query a wide variety of information from Presto.

9.1.2.1 Parameter Query

A number of user subroutine parameters may be set as described in Section 9.2.2.3. These subroutine parameters can be obtained from the Presto input file via the query functions listed below.

```
aupst_get_real_param(string var_name, real var_value,
                     int error_code)

aupst_get_integer_param(string var_name, int var_value,
                        int error_code)

aupst_get_string_param(string var_name, string var_value,
                       int error_code)
```

All three of these subroutine calls are tied to a corresponding "parameter" command line that will appear in the Presto input file. The parameter command lines are described in Section 9.2.2.3. These command lines are named based on the type of value they store, i.e., `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`.

We will use the example of a real parameter to show how the subroutine call works in conjunction with the `SUBROUTINE REAL PARAMETER` command line. Suppose we have a real parameter radius that is set to a value of 2.75 on the `SUBROUTINE REAL PARAMETER` command line:

```
SUBROUTINE REAL PARAMETER: radius = 2.75
```

Also suppose we have a call to `aupst_get_real_parameter` in the user subroutine:

```
call aupst_get_real_parameter(‘‘radius’’,cyl_radius,error_code)
```

In the call to `aupst_get_real_parameter`, we have `var_name` set to `radius` and `var_value` defined as the real FORTRAN variable `cyl_radius`. The call to `aupst_get_real_parameter` will assign the value 2.75 to the FORTRAN variable `cyl_radius`. A similar pattern is followed for integer and string parameters.

The arguments for the parameter-related query functions are described in Table 9.3, Table 9.4, and Table 9.5. The function is repeated prior to each table for easy reference.

```
aupst_get_real_param(string var_name, real var_value,
                     int error_code)
```

Table 9.3: aupst_get_real_param Arguments

Parameter	**Usage**	**Data Type**	**Description**
var_name	Input	String	Name of a real-valued subroutine parameter, as defined in the Presto input file via the `SUBROUTINE REAL PARAMETER` command line.
var_value	Output	Real	Name of a real variable to be used in the FORTRAN subroutine. The FORTRAN variable var_value will be set to the value specified by the `SUBROUTINE REAL PARAMETER` command line.
error_code	Output	Integer	Error code indicating status of retrieving the parameter value from the input file. If the retrieval is successful, error_code is set to 0. If the parameter is not found or is the wrong type, error_code is set to a value other than 0.

```
aupst_get_integer_param(string var_name, int var_value,
                        int error_code)
```

Table 9.4: aupst_get_integer_param Arguments

Parameter	**Usage**	**Data Type**	**Description**
var_name	Input	String	Name of an integer-valued subroutine parameter, as defined in the Presto input file via the `SUBROUTINE INTEGER PARAMETER` command line.
var_value	Output	Integer	Name of an integer variable to be used in the FORTRAN subroutine. The FORTRAN variable var_value will be set to the value specified by the `SUBROUTINE INTEGER PARAMETER` command line.
error_code	Output	Integer	Error code indicating status of retrieving the parameter value from the input file. If the retrieval is successful, error_code is set to 0. If the parameter is not found or is the wrong type, error_code is set to a value other than 0.

```
aupst_get_string_param(string var_name, string var_value,
                       int error_code)
```

Table 9.5: aupst_get_string_param Arguments

Parameter	**Usage**	**Data Type**	**Description**
var_name	Input	String	Name of a string-valued subroutine parameter, as defined in the Presto input file via the `SUBROUTINE STRING PARAMETER` command line.
var_value	Output	String	Name of a string variable to be used in the FORTRAN subroutine. The FORTRAN variable var_value will be set to the value specified by the `SUBROUTINE STRING PARAMETER` command line.
error_code	Output	Integer	Error code indicating status of retrieving the parameter value from the input file. If the retrieval is successful, error_code is set to 0. If the parameter is not found or is the wrong type, error_code is set to a value other than 0.

9.1.2.2 Function Data Query

The function data query routine listed below may be used for extracting data from a function that is defined in a `DEFINITION FOR FUNCTION` command block in the Presto input file. This query allows the user to directly access information stored in a function defined in the Presto input file.

```
aupst_evaluate_function(string func_name, real
input_times[], int num_times, real output_data[])
```

The arguments for this function are described in Table 9.6.

Table 9.6: aupst_evaluate_function Arguments

Parameter	**Usage**	**Data Type**	**Description**
func_name	Input	String	Name of the function to look up.
input_times (num_times)	Input	Real	Array of times used to extract values of the function.
num_times	Input	Integer	Length of the array input_times.
output_data (num_times)	Output	Real	Array of output values of the named function at the specified times.

9.1.2.3 Time Query

The time query function can be used to determine the current analysis time. This is the time associated with the new time step. This time may not be equivalent to the `evaluation_time` argument passed into the subroutine (see Section 9.1.1, Table 9.1) as some boundary conditions need to be evaluated at different times than others. The parameter of the time query function listed below is given in Table 9.7.

```
aupst_get_time(real time)
```

Table 9.7: aupst_get_time Argument

Parameter	**Usage**	**Data Type**	**Description**
time	Output	Real	Current analysis time.

9.1.2.4 Field Variables

Field variables (displacements, stresses, etc.) may be defined on groups of mesh entities. A number of queries are available for getting and putting field variables. These queries involve passing in a set of mesh-entity identification numbers to receive field values on the mesh entities. There are query functions to check for the existence and size of a field, functions to retrieve the field values, and functions to store new variables in a field. The field query functions listed below can be used to extract any registered nodal or element variable field.

```
aupst_check_node_var(int num_nodes, int num_components,
                     int node_ids[], string var_name,
                     int error_code)

aupst_check_elem_var(int num_elems, int num_components,
                     int elem_ids[], string var_name,
                     int error_code)

aupst_get_node_var(int num_nodes, int num_components,
                   int node_ids[], real return_data[],
                   string var_name, int error_code)

aupst_get_elem_var(int num_elems, int num_components,
                   int elem_ids[], real return_data[],
                   string var_name, int error_code)

aupst_put_node_var(int num_nodes, int num_components,
                   int node_ids[], real new_data[],
                   string var_name, int error_code)

aupst_put_elem_var(int num_elems, int num_components,
                   int elem_ids[], real new_data[],
                   string var_name, int error_code)
```

The arrays where data are stored are static arrays. These arrays of a set size will be declared at the beginning of a user subroutine. The query functions to check for the existence and size of a field can be used to ensure that the size of the array of information being returned from Presto does not exceed the size of the array allocated by the user.

The arguments to field query functions are defined in Table 9.8 through Table 9.13. The function is repeated before each table for easy reference.

```
aupst_check_node_var(int num_nodes, int num_components,
                     int node_ids[], string var_name,
                     int error_code)
```

Table 9.8: aupst_check_node_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_nodes	Input	Integer	Number of nodes used to extract field information.
num_componen ts	Output	Integer	Number of components in the field information. A displacement field at a node has three components, for example.
node_ids (num_nodes)	Input	Integer	Array of size num_nodes listing the node identification number for each node where field information will be retrieved.
var_name	Input	String	Name of the field variable. The field variable must be a registered Presto variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_check_elem_var(int num_elems, int num_components,
                     int elem_ids[], string var_name,
                     int error_code)
```

Table 9.9: aupst_check_elem_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements used to extract field information.
num_componen ts	Output	Integer	Number of components in the field information. A stress field for a an eight-node hexahedron element has six components, for example.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
var_name	Input	String	Name of the field variable. The field variable must be a registered Presto variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_get_node_var(int num_nodes, int num_components,
                   int node_ids[], real return_data[],
                   string var_name, int error_code)
```

Table 9.10: aupst_get_node_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_nodes	Input	Integer	Number of nodes used to extract field information.
num_componen ts	Input	Integer	Number of components in the field information. A displacement field at a node has three components, for example.
node_ids (num_nodes)	Input	Integer	Array of size num_nodes listing the node identification number for each node where field information will be retrieved.
return_data (num_components, num_nodes)	Output	Real	Array of size num_components $\times$ num_nodes containing the field data at each node.
var_name	Input	String	Name of the field variable. The field variable must be a registered Presto variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_get_elem_var(int num_elems, int num_components,
                   int elem_ids[], real return_data[],
                   string var_name, int error_code)
```

Table 9.11: aupst_get_elem_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements used to extract field information.
num_componen ts	Input	Integer	Number of components in the field information. A stress field for a an eight-node hexahedron element has six components, for example.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
return_data (num_components, num_elems)	Output	Real	Array of size num_components $\times$ num_elems containing the field data for each element.
var_name	Input	String	Name of the field variable. The field variable must be a registered Presto variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_put_node_var(int num_nodes, int num_components,
                   int node_ids[], real new_data[],
                   string var_name, int error_code)
```

Table 9.12: aupst_put_node_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_nodes	Input	Integer	Number of nodes for which the user will specify the field data.
num_components	Input	Integer	Number of components in the field information. A displacement field at a node has three components, for example.
node_ids (num_nodes)	Input	Integer	Array of size num_nodes listing the node identification number for each node where field information will be retrieved.
new_data (num_compon ents, num_nodes)	Input	Real	Array of size num_components $\times$ num_nodes containing the new data for the field.
var_name	Input	String	Name of the field variable. The field variable must be a registered Presto variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_put_elem_var(int num_elems, int num_components,
                   int elem_ids[], real new_data[],
                   string var_name, int error_code)
```

Table 9.13: aupst_put_elem_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements for which the user will specify the field data.
num_componen ts	Input	Integer	Number of components in the field information. A stress field for a an eight-node hexahedron element has six components, for example.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
new_data (num_compon ents, num_elems)	Input	Real	Array of size num_components $\times$ num_elems containing the new data for the field.
var_name	Input	String	Name of the field variable. The field variable must be a registered Presto variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

9.1.2.5 Global Variables

Global variables may be extracted or set from user subroutines. A global variable has a single value for a given region.

Global variables have limited support for parallel operations. There are two subroutines to perform parallel modification of global variables: `aupst_put_global_var` and `aupst_local_put_global_var`.

- The subroutine `aupst_local_put_global_var` only modifies a temporary local copy of the global variable. The local copies on the various processors are reduced to create the true global value at the end of the time step. Global variables set with `aupst_local_put_global_var` do not have the single processor value available immediately. The true global variable will not be available through the `aupst_get_global_var` routine until the next time step.

- The subroutine `aupst_put_global_var` attempts to immediately modify and perform a parallel reduction of the value of a global variable. Care must be taken to call this routine on all processors at the same time with the same arguments. Failure to call the routine from all processors will result in the code hanging. For some types of subroutines this is not possible or reliable. For example, a boundary condition subroutine may not be called at all on a processor that contains no nodes in the set of nodes assigned to the boundary condition. It is recommended that `aupst_local_put_global_var` only be used in conjunction with a user subroutine referenced in a `USER OUTPUT` command block (Section 8.1.2).

Only user-defined global variables may be modified by the user subroutine (see Section 9.2.4). However, any global variable that exists on the region may be checked or extracted. The following subroutine calls pertain to global variables:

```
aupst_get_global_var(int num_comp, real return_data,
                     string var_name, int error_code)

aupst_put_global_var(int num_comp, real input_data,
                     string reduction_type,
                     string var_name, int error_code)

aupst_local_put_global_var(int num_comp, real input_data,
                     string var_name, string reduction_type,
                     int error_code)
```

The arguments for subroutine calls pertaining to global variables are defined in Table 9.14 through Table 9.17. The call is repeated before each table for easy reference.

```
aupst_check_global_var(int num_comp, string var_name
                       int error_code)
```

Table 9.14: aupst_check_global_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_comp	Output	Integer	Number of components of the global variable.
var_name	Input	String	Name of the global variable.
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist or in some way cannot be accessed.

```
aupst_get_global_var(int num_comp, real return_data,
                     string var_name, int error_code)
```

Table 9.15: aupst_get_global_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_comp	Input	Integer	Number of components of the global variable.
return_data	Output	Real	Value of the global variable.
var_name	Input	String	Name of the global variable.
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist or in some way cannot be accessed.

```
aupst_put_global_var(int num_comp, real input_data,
                     string reduction_type,
                     string var_name, int error_code)
```

Table 9.16: aupst_put_global_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_comp	Input	Integer	Number of components of the global variable.
input_data	Input	Real	New value of the global variable.
reduction_type	Input	String	Type of parallel reduction to perform on the variable. Options are “sum”, “min”, “max”, and “none”.
var_name	Input	String	Name of the global variable.
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist, in some way cannot be accessed, or may not be overwritten.

```
aupst_local_put_global_var(int num_comp, real input_data,
                           string var_name,
                           string reduction_type,
                           int error_code)
```

Table 9.17: aupst_local_put_global_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_comp	Input	Integer	Number of components of the global variable.
input_data	Input	Real	New value of the global variable.
reduction_type	Input	String	Type of parallel reduction to perform on the variable. Options are "sum", "min", and "max". The operation type specified here must match the operation type given to the user-defined global variable when it is defined in the Presto input file.
var_name	Input	String	Name of the global variable.
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist, in some way cannot be accessed, or may not be overwritten.

9.1.2.6 Topology Extraction

The element and surface subroutines operate on groups of elements or faces. The elements and faces may have a variety of topologies. Topology queries can be used to get topological data about elements and faces. The topology of an object is represented by an integer. The integer is formed from a function of the number of dimensions, vertices, and nodes of an object. The topology of an object is given by

```
topology = num_node + 100 * num_vert + 10000 * num_dim .
```

In a FORTRAN routine, the number of nodes can easily be extracted with the mod function:

```
num_node = mod(topo,100)
num_vert = mod(topo / 100, 100)
num_dim  = mod(topo / 10000, 100)
```

Table 9.18 lists the topologies currently in use by Presto.

Table 9.18: Topologies Used by Presto

Topology	**Element / Face Type**
00101	One-node particle
10202	Two-node beam, truss, or damper
20404	Four-node quadrilateral
20303	Three-node triangle
20304	Four-node triangle
20306	Six-node triangle
30404	Four-node tetrahedron
30408	Eight-node tetrahedron
30410	Ten-node tetrahedron
30808	Eight-node hexahedron

The following topology query functions are available in Presto:

```
aupst_get_elem_topology(int num_elems, int elem_ids[],
                        int topology[], int error_code)

aupst_get_elem_nodes(int num_elems, int elem_ids[],
                     int elem_node_ids[], int error_code)

aupst_get_face_topology(int num_faces, int face_ids[],
                        int topology[], int error_code)

aupst_get_face_nodes(int num_faces, int face_ids[],
                     int face_node_ids[], int error_code)
```

The arguments for the topology extraction functions are defined in Table 9.19 through Table 9.22. The function is repeated before each table for easy reference.

```
aupst_get_elem_topology(int num_elems, int elem_ids[],
                        int topology[], int error_code)
```

Table 9.19: aupst_get_elem_topology Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements from which the topology will be extracted.
elem_ids (num_elems)	Input	Integer	Array of length num_elems listing the element identification for each element from which the topology will be extracted.
topology (num_elems)	Output	Integer	Array of length num_elems that has the topology for each element. See Table 8.18.
error_code	Output	Integer	Error code indicating status of retrieving the element identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the element identification numbers is not valid.

```
aupst_get_elem_nodes(int num_elems, int elem_ids[],
                     int elem_node_ids[], int error_code)
```

Table 9.20: aupst_get_elem_nodes Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements from which the topology will be extracted.
elem_ids (num_elems)	Input	Integer	Array of length num_elems listing the element identification for each element from which the topology will be extracted.
elem_node_ids (number of nodes for element type $\times$ num_elems)	Output	Integer	Array containing the node identification numbers for each element requested. The length of the array is the total number of nodes contained in all elements. If the elements are eight-node hexahedra, then the number of nodes will be $8 \times$ num_elems. The first set of eight entries in the array will be the eight nodes defining the first element. The second set of eight entries will be the eight nodes defining the second element, and so on.
error_code	Output	Integer	Error code indicating status of retrieving the element identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the element identification numbers is not valid.

```
aupst_get_face_topology(int num_faces, int face_ids[],
                        int topology[], int error_code)
```

Table 9.21: aupst_get_face_topology Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_faces	Input	Integer	Number of faces from which the topology will be extracted.
face_ids (num_faces)	Input	Integer	Array of length num_faces listing the face identification for each face from which the topology will be extracted.
topology (num_faces)	Output	Integer	Array of length num_faces containing the output topologies of each face.
error_code	Output	Integer	Error code indicating status of retrieving the face identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the face identification numbers is not valid.

```
aupst_get_face_nodes(int num_faces, int face_ids[],
                     int face_node_ids[], int error_code)
```

Table 9.22: aupst_get_face_nodes Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_faces	Input	Integer	Number of faces from which the topology will be extracted.
face_ids (num_faces)	Input	Integer	Array of length num_faces listing the face identification for each face from which the topology will be extracted.
face_node_ids (number of nodes for face type × num_faces)	Output	Integer	Array containing the node identification numbers for each face requested. The length of the array is the total number of nodes contained in all faces. If the faces are four-node quadrilaterals, then the number of nodes will be 4 × num_faces. The first set of four entries in the array will be the four nodes defining the first face. The second set of four entries will be the four nodes defining the second face, and so on.
error_code	Output	Integer	Error code indicating status of retrieving the face identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the face identification numbers is not valid.

9.1.3 Miscellaneous Query Functions

A number of miscellaneous query functions are available for computing some commonly used quantities.

```
aupst_get_one_elem_centroid(int num_elems, int elem_ids[],
                     real centroids, int error_code)

aupst_get_point(string point_name, real point_coords,
                     int error_code)

aupst_get_proc_num(proc_num)
```

The arguments for the miscellaneous query functions are defined in Table 9.23 through Table 9.25. The function is repeated before each table for easy reference.

```
aupst_get_one_elem_centroid(int num_elems, int elem_ids[],
                     real centroids[], int error_code)
```

Table 9.23: aupst_get_one_elem_centroid Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements for which to extract the topology.
elem_ids (num_elems)	Input	Integer	Array of length num_elems listing the element identification for each element for which the centroid will be computed.
centroids (3, num_elems)	Output	Real	Array of length 3 $\times$ num_elems containing the centroid of each element.
error_code	Output	Integer	Error code indicating status of retrieving the element identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the element identification numbers is not valid.

```
aupst_get_point(string point_name, real point_coords,
                        int error_code)
```

Table 9.24: aupst_get_point Arguments

Parameter	**Usage**	**Data Type**	**Description**
point_name	Input	String	SIERRA name for a given point.
point_coords (3)	Output	Real	Array of length 3 containing the x, y, and z coordinates of the point.
error_code	Output	Integer	Error code indicating status of retrieving the point. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if the point cannot be found

```
aupst_get_proc_num(proc_num)
```

Table 9.25: aupst_get_proc_num Arguments

Parameter	**Usage**	**Data Type**	**Description**
proc_num	Output	Integer	Processor number of the calling process. This number can be used for informational purposes. A common example is that output could only be written by a single processor, e.g., processor 0, rather than by all processors.

9.2 User Subroutines: Command File

In addition to the actual user subroutine, you will need to add command lines to your input file to make use of your user subroutine. This section describes the command lines that are used in conjunction with user subroutines. This section also describes two additional command blocks, `TIME STEP INITIALIZATION` and `USER VARIABLE`. The `TIME STEP INITIALIZATION` command block lets you execute a user subroutine at the beginning of a time step as opposed to some later time. The `USER VARIABLE` command block can be used in conjunction with user subroutines or for user-defined output.

9.2.1 Subroutine Identification

As described in Section 2.1.4, there is one command line associated with the user subroutine functionality that must be provided in the domain scope:

```
USER SUBROUTINE FILE = <string>file_name
```

The named file may contain one or more user subroutines. The file must have an extension of ".F", as in blast.F.

9.2.2 User Subroutine Command Lines

```
{begin command block}
  NODE SET SUBROUTINE = <string>subroutine name |
    SURFACE SUBROUTINE = <string>subroutine_name |
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
{end command block}
```

A number of user subroutine command lines will appear in some Presto command block. User subroutine commands can appear in boundary condition, element death, user output, and state initialization command blocks. The possible command lines are shown above. The following sections describe the command lines related to user subroutines.

9.2.2.1 Type

User subroutines are currently available in three general types: node set, surface, and element.

Node set subroutines operate on groups of nodes. The command line for defining a node set subroutine is

```
NODE SET SUBROUTINE = <string>subroutine_name ,
```

where `subroutine_name` is the name of the user subroutine. The name is case sensitive. A node set subroutine will operate on all nodes contained in an associated mechanics instance.

Surface subroutines work on groups of surfaces. A surface may be an external face of a solid element or the face of a shell element associated with either the positive or negative normal for the surface of the shell. The command line for defining a surface subroutine is

```
SURFACE SUBROUTINE = <string>subroutine_name .
```

Element block subroutines work on groups of elements. The command line for defining an element block subroutine is

```
ELEMENT BLOCK SUBROUTINE = <string>subroutine_name .
```

An element may be a solid element such as a hexahedron or a two-dimensional element such as a shell.

Different Presto features may accept one or more types of user subroutines. Only one subroutine is allowed per command block.

9.2.2.2 Debugging

Subroutines may be run in a special debugging mode to help catch memory errors. For example, there is a potential for a user subroutine to write outside of its allotted data space by writing beyond the bounds of an input or output array. Generally, this causes Presto to crash, but it also has the potential to introduce other very hard-to-trace bugs into the Presto analysis. Subroutines run in debug mode require more memory and more processing time than subroutines not run in debug mode.

Subroutine debugging is on by default in debug executables. It can be turned off with the following command line:

```
SUBROUTINE DEBUGGING OFF
```

Subroutine debugging is off by default in optimized executables. It can be turned on with the following command line:

```
SUBROUTINE DEBUGGING ON
```

9.2.2.3 Parameters

All user subroutines have the ability to use parameters. Parameters are defined in the input file and are quickly accessible by the user subroutine during run time. Parameters are a way of making a single user subroutine much more versatile. For example, a user subroutine could be written to define a periodic loading on a structure. A parameter for the subroutine could be defined specifying the frequency of the function. In this way, the same subroutine can be used in different parts of the model, and the subroutine behavior can be modified without recompiling the program. These command lines are placed within the scope of the command block in which the user subroutine is specified.

Real-valued parameters can be stored with the following command line:

```
SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
```

Integer-valued parameters can be stored with the following command line:

```
SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
```

String-valued parameters can be stored with the following command line:

```
SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
```

Any number of subroutine parameters may be defined. The subroutine parameters may be defined in any order within the command block. The user subroutine may request the values of the parameters but is not required to use them or even have any knowledge of their existence. An example of subroutine usage with parameters is as follows:

```
BEGIN PRESSURE
  SURFACE = surface_1
  SURFACE SUBROUTINE = blast_pressure
  SUBROUTINE REAL PARAMETER: blast_time = 1.2
  SUBROUTINE REAL PARAMETER: blast_power = 1.3e+07
  SUBROUTINE STRING PARAMETER: formulation = alpha
  SUBROUTINE INTEGER PARAMETER: decay_exponent = 2
  SUBROUTINE DEBUGGING ON
END PRESSURE
```

In the above example, four parameters are associated with the subroutine `blast_pressure`. Two of the parameters are real (`blast_time` and `blast_power`), one of the parameters is a string (`formulation`), and one of the parameters is an integer (`decay_exponent`). To access the parameters in the user subroutine, the user will need to include interface calls described in previous sections.

9.2.3 Time Step Initialization

```
BEGIN TIME STEP INITIALIZATION
  # {mesh-entity set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list> surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name |
    SURFACE SUBROUTINE = <string>sub_name |
    ELEMENT BLOCK SUBROUTINE = <string>sub_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional command
  ACTIVE PERIODS = <string list>period_names
END TIME STEP INITIALIZATION
```

The `TIME STEP INITIALIZATION` command block, which appears in the region scope, is used to flag a user subroutine to run at the beginning of every time step. This subroutine can be used to compute quantities used by other command types. For example, if the traction on a surface was dependent on the area, the time step initialization subroutine could be used to calculate the area, and that area could be stored and later read when calculating the traction. The user initialization subroutine will pass the specified mesh objects to the subroutine for use in calculating some value.

The `TIME STEP INITIALIZATION` command block contains two groups of commands—mesh entity set and user subroutine. In addition to the command lines in the these command groups, there is an additional command line: `ACTIVE PERIODS`. The `ACTIVE PERIODS` command line is used to activate or deactivate the running of the user subroutine at the beginning of every time step for certain time periods. Following are descriptions of the different command groups and the `ACTIVE PERIODS` command lines.

9.2.3.1 Mesh-Entity Set Commands

The {`mesh-entity set commands`} portion of the `TIME STEP INITIALIZATION` command block specifies the nodes, element faces, or elements associated with the particular subroutine that will be run at the beginning of the applicable time steps. This portion of the command block can include some combination of the following command lines:

```
NODE SET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list> surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes, element faces, or elements. See Section 5.1 for more information about the use of these command lines for mesh entities. There must be at least one `NODE SET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

9.2.3.2 User Subroutine Commands

The following command lines are related to the user subroutine specification:

```
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

Only one of the first three command lines listed above can be specified in the command block. The particular command line selected depends on the mesh-entity type of the variable being initialized. For example, variables associated with nodes would be initialized if you are using the `NODE SET SUBROUTINE` command line, variables associated with faces if you are using the `SURFACE SUBROUTINE` command line, and variables associated with elements if you are using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the selected subroutine command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 9.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 9.

9.2.3.3 Additional Command

The `ACTIVE PERIODS` command line can appear as an option in the `TIME STEP INITIALIZATION` command block:

```
ACTIVE PERIODS = <string list>period_names
```

This command line determines when time step initialization by a user subroutine is active. See Section 2.5 for more information about this optional command line.

9.2.4 User Variables

```
BEGIN USER VARIABLE <string>var_name
  TYPE = <string>NODE|ELEMENT|GLOBAL
    [<string>REAL|INTEGER LENGTH = <integer>length]|
    [<string>SYM_TENSOR|FULL_TENSOR|VECTOR]
  GLOBAL OPERATOR = <string>SUM|MIN|MAX]
  INITIAL VALUE = <real list>values
  USE WITH RESTART
END [USER VARIABLE <string>var_name]
```

The `USER VARIABLE` command block is used to create a user-defined variable. This kind of variable may be used for scratch space in a user subroutine or for some user-defined output. A user-defined variable may be output to the results file or the history file just like any registered variable; i.e., a user-defined variable once defined by the `USER VARIABLE` command block can be specified in a `USER OUTPUT` command block, a `RESULTS OUTPUT` command block, and a `HISTORY OUTPUT` command block.

User-defined variables are associated with mesh entities. For example, a node variable will exist at every node of the model. An element variable will exist on every element of the model. A global variable will have a single value for the entire model.

If the user-defined variable functionality is used in conjunction with restart, the `USE WITH RESTART` command line must be included.

The `USER VARIABLE` command block is placed within a Presto region. The command block begins with the input line

```
BEGIN USER VARIABLE <string>var_name
```

and ends with the input line

```
END [USER VARIABLE <string>var_name] ,
```

where `var_name` is a user-selected name for the variable.

In the above command block:

- A user-defined variable has an associated type that is specified by the `TYPE` command line, which itself contains several parameters. The `TYPE` command line is required.

  1. The variable must be a nodal quantity, an element quantity, or a global quantity. The options `NODE`, `ELEMENT`, and `GLOBAL` determine whether the variable will be a nodal, element, or global quantity. One of these options must appear on the `TYPE` command line.
  2. The user-defined variable can be either an integer or a real, as specified by the `INTEGER` or `REAL` option.

3. The length of the variable must be set by using one of the options SYM_TENSOR, FULL_TENSOR, VECTOR, or LENGTH = <integer>length. If the LENGTH option is used, the user must specify whether the variable is an integer number or a real number by using the INTEGER or REAL option. If the SYM_TENSOR option is used, the variable has six real components. If the FULL_TENSOR is used, the variable has nine real components. If the VECTOR option is used, the variable has three real components. The three options SYM_TENSOR, FULL_TENSOR, and VECTOR all imply real numbers, and thus the REAL option need not be included in the command line when one of these three options is specified.

Some examples of the TYPE command line follow:

```
type = global real length = 1
type = element tensor
type = element real length = 3
type = node sym_tensor
type = node vector
```

- If you use the GLOBAL option on the TYPE command line, a global variable is created, and this global variable must be given an associated reduction type, which is specified by the GLOBAL OPERATOR command line. The reduction type tells Presto how to reduce the individual values stored on each processor to a mesh global value. Global reductions are performed at the end of each time step. Any modifications to a global variable made by an aupst_local_put_global_var call (see Section 9.1.2.5) will not be seen until the next time step after the user-defined global variables have been updated and reduced. The SUM operator sums all processor variable contributions. The MAX operator takes the maximum value of the aupst_local_put_global_var calls. The MIN operator takes the minimum value of the aupst_local_put_global_var calls.

- One or more initial values may be specified for the user-defined variable in the INITIAL VALUE command line. The number of initial values specified should be the same as the length of the variable, as specified in the TYPE command line either explicitly via the LENGTH option or implicitly via the SYM_TENSOR, FULL_TENSOR, or VECTOR option. The initial values will be copied to the variable space on every mesh object on which the variable is defined. Only real type variables may be given initial values at this time.

- All intrinsic type options such as REAL, INTEGER, SYM_TENSOR, FULL_TENSOR, VECTOR and the LENGTH option can be used with any of the mesh entity options (NODE, ELEMENT, GLOBAL).

- As indicated previously, if the user-defined variable functionality is used in conjunction with restart, the `USE WITH RESTART` command line must be included.

9.3 User Subroutines: Compilation and Execution

Running a code with user subroutines is a two-step process. First, you must create an executable version of Presto that recognizes the user subroutines. Next, you must use this version of Presto for an actual Presto run with an input file that incorporates the proper user subroutine command lines.

How the above two steps are carried out is site-specific. The actual process will depend on how Presto is set up at your installation. We will give an example that shows how the process is carried out on various systems at Sandia using SIERRA command lines. SIERRA is a general code framework and code management system at Sandia.

For the first step, you will need the user subroutine, in a FORTRAN file, and a Presto input file that makes use of the user subroutine. You will use a system command line of the general form shown below.

```
% sierra presto -i <string>input_file_name --make
```

Suppose that you have a subdirectory in your area called `test` and you wish to incorporate a user subroutine called `blast_load`. The actual user subroutine will be in a file called `blast_load.F`, and the associated input file will be called `blast_load_1.i`. Both of these files will be in the directory `test`. In the input file, you will have the following command line in the domain scope:

```
USER SUBROUTINE FILE = blast_load.F
```

You will also have some subset of the command lines described in the previous section in your Presto input file. The specific form of the system command line to execute the first step of the user subroutine process is shown below.

```
% sierra presto -i blast_load_1.i --make
```

The above command will create a local version of Presto in a local directory named `UserSubsProject`. The system command line to run the local version of Presto is shown below.

```
% sierra presto -i <string>input_file_name
      -x UserSubsProject
```

The specific form of the system command line you will execute in the subdirectory test is shown below.

```
% sierra presto -i blast_load_1.i -x UsersSubsProjects
```

The second command line runs Presto using `blast_load_1.i` as an input file and utilizes the user subroutines in the process. Again, all of this is a site-specific example. You must determine how Presto is set up at your installation to determine what system command lines are necessary to build Presto with user subroutines and then use this version of Presto.

9.4 User Subroutines: Examples

9.4.1 Pressure as a Function of Space and Time

The following code is an example of a user subroutine to compute blast pressures on a group of faces. The blast pressure simulates a blast occurring at a specified position and time. The blast wave radiates out from the center of the blast and dissipates over time. This subroutine is included in the Presto input file as follows:

```
#In the domain scope:
user subroutine file = blast_load.F

#In the region scope:
begin pressure
  surface = surface_1
  surface subroutine = blast_load
  subroutine real parameter: pos_x = 5.0
  subroutine real parameter: pos_y = 5.0
  subroutine real parameter: pos_z = 1.6
  subroutine real parameter: wave_speed = 1.5e+02
  subroutine real parameter: blast_time = 0.0
  subroutine real parameter: blast_energy = 1.0e+09
  subroutine real parameter: blast_wave_width = 0.75
end pressure
```

The FORTRAN 77 subroutine listing follows. Note that it would be possible to increase the speed of this subroutine by calling the topology functions (see Section 9.1.2.6) on groups of elements, though this would increase subroutine complexity.

```
c
c  Subroutine to simulate a blast load on a surface
c
      subroutine blast_load(num_faces, num_vals,
     &  eval_time, faceID, pressure, flags, err_code)

      implicit none
c
c  Subroutine input arguments
c
      integer num_faces
      double precision eval_time
      integer faceID(num_faces)
      integer num_vals
```

```
c
c  Subroutine output arguments
c
      double precision pressure(num_vals, num_faces)
      integer flags(num_faces)
      integer err_code
c
c  Variables to hold the subroutine parameters
c
      double precision pos_x, pos_y, pos_z, wave_speed,
     &                 blast_time, blast_energy,
     &                 blast_wave_width
c
c  Local variables
c
      integer iface, inode
      integer cur_face_id, face_topo, num_nodes
      integer num_comp_check
      double precision dist, blast_o_rad, blast_i_rad
      double precision blast_volume, blast_pressure
      integer query_error
      double precision face_center(3)
c
c  Create some static variables to hold queried
c  information.  Assume no face has more than 10
c  nodes
c
      double precision face_nodes(10)
      double precision face_coords(3, 10)
c
c  Extract the subroutine parameters
c
      call aupst_get_real_param("pos_x",pos_x,query_error)
      call aupst_get_real_param("pos_y",pos_y,query_error)
      call aupst_get_real_param("pos_z",pos_z,query_error)
      call aupst_get_real_param("wave_speed",wave_speed,
     &                          query_error)
      call aupst_get_real_param("blast_energy",
     &                          blast_energy,query_error)
      call aupst_get_real_param("blast_time",
     &                          blast_time,query_error)
      call aupst_get_real_param("blast_wave_width",
     &                  blast_wave_width, query_error)
c
```

```
c  Determine the outer radius of the blast wave
c
      blast_o_rad = (eval_time - blast_time) * wave_speed
      if(blast_o_rad .le. 0.0) return;
c
c  Determine the inner radius of the blast wave
c
      blast_i_rad = blast_o_rad - blast_wave_width
      if(blast_i_rad .le. 0.0) blast_i_rad = 0.0
c
c  Determine the total volume the blast wave occupies
c
      blast_volume = 3.1415 * (4.0/3.0) *
     &               (blast_o_rad**2 - blast_i_rad**2)
c
c  Determine the total pressure on faces inside the
c  blast wave
c
      blast_pressure = blast_energy / blast_volume
c
c  Loop over all faces in the set
c
      do iface = 1, num_faces
c
c  Extract the topology of the current face
c
        cur_face_id = faceID(iface)
        call aupst_get_face_topology(1, cur_face_id,
     &                          face_topo, query_error)
c
c  Determine the number of nodes of the current face
c
        num_nodes = mod(face_topo,100)
c
c  Extract the node ids for nodes contained in the current
c  face
c
        call aupst_get_face_nodes(1, cur_face_id,
     &                           face_nodes, query_error)
c
c  Extract the nodal coordinates of the face nodes
c
        call aupst_get_node_var(num_nodes, 3, face_nodes,
     &          face_coords, "coordinates", query_error)
c
```

```
c  Compute the centroid of the face
c
         face_center(1) = 0.0
         face_center(2) = 0.0
         face_center(3) = 0.0
         do inode = 1, num_nodes
           face_center(1) = face_center(1) +
     &                      face_coords(1,inode)
           face_center(2) = face_center(2) +
     &                      face_coords(2,inode)
           face_center(3) = face_center(3) +
     &                      face_coords(3,inode)
         enddo
         face_center(1) = face_center(1)/num_nodes
         face_center(2) = face_center(2)/num_nodes
         face_center(3) = face_center(3)/num_nodes
c
c  Determine the distance from the current face
c  to the blast center
c
         dist = sqrt((face_center(1) - pos_x)**2 +
     &               (face_center(2) - pos_y)**2 +
     &               (face_center(3) - pos_z)**2)
c
c  Apply pressure to the current face if it falls within
c  the blast wave
c
         if(dist .ge. blast_i_rad .and.
     &       dist .le. blast_o_rad) then
           pressure(1,iface) = blast_pressure
         else
           pressure(1,iface) = 0.0
         endif
       enddo
       err_code = 0
       end
```

9.4.2 Error Between a Computed and an Analytic Solution

The following code is a user subroutine to compute the error between a Presto-computed subroutine and an expected analytic manufactured solution result. This subroutine is called by a `USER OUTPUT` command block immediately prior to producing an output Exodus file. The error for the mesh is computed by taking the squared difference between the computed and analytic displacements at every node. Finally, a global sum of the error is produced along with the square root norm of the error.

This user subroutine requires a user variable, which is defined in the Presto input file. The command block for the user variable specified in this user subroutine is as follows:

```
begin user variable conv_error
 type = global real length = 1
  global operator = sum
  initial value = 0.0
end user variable conv_error
```

The subroutine is called in the Presto input file as follows:

```
begin user output
  node set = nodelist_10
  node set subroutine = conv0_error
  subroutine real parameter: char_length = 1.0
  subroutine real parameter: char_time   = 1.0e-3
  subroutine real parameter: x_offset    = 0.0
  subroutine real parameter: y_offset    = 0.0
  subroutine real parameter: z_offset    = 0.0
  subroutine real parameter: t_offset    = 0.0
  subroutine real parameter: u0          = 0.01
  subroutine real parameter: v0          = 0.02
  subroutine real parameter: w0          = 0.03
  subroutine real parameter: alpha       = 1.0
  subroutine real parameter: youngs_modulus = 10.0e6
  subroutine real parameter: poissons_ratio = 0.3
  subroutine real parameter: density        = 0.0002588
  subroutine real parameter: num_nodes      = 125.0
end user output
```

The FORTRAN listing for the subroutine is as follows:

```
      subroutine conv0_error(num_nodes, num_vals,
     &  eval_time, nodeID, values, flags, ierror)
      implicit none

      integer num_nodes
      integer num_vals
      double precision eval_time
      integer nodeID(num_nodes)
      double precision values(1)
      integer flags(1)
      integer ierror
c
c     Local vars
c
      integer inode
      integer error_code
      double precision clength, ctime, xoff, yoff, zoff, toff
      double precision zero, one, two, three, four, nine
      double precision mod_coords(3,3000)
      double precision cdispl(3,3000)
      integer num_comp_check
      double precision expat
      double precision x, y, z, t
      double precision u0, v0, w0, alpha
      double precision pi
      double precision half
      double precision mdisplx, mdisply, mdisplz
      double precision xdiff, ydiff, zdiff
      double precision conv_error
      double precision numnod

      pi    = 3.141592654
      half  = 0.5
      zero  = 0.0
      one   = 1.0
      two   = 2.0
      three = 3.0
      four  = 4.0
      nine  = 9.0
c
c  Check that the nodal coordinates will fit into the
c  statically allocated array
c
      if(num_nodes .gt. 3000) then
        write(6,*) ERROR in sphere disp, ,
     &  num_nodes exceeds static array size
        ierror = 1
        return
      endif
c
c  Extract the model coordinates for all nodes
c
      call aupst_check_node_var(num_nodes, num_comp_check,
     &                          nodeID, "model_coordinates",
     &                          ierror)
      if(ierror .ne. 0) return
      if(num_comp_check .ne. 3) return
      call aupst_get_node_var(num_nodes, num_comp_check,
     &       nodeID, mod_coords, "model_coordinates",
     &       ierror)
c
c  Extract the computed displacements for all nodes
c
      call aupst_check_node_var(num_nodes, num_comp_check,
     &                          nodeID, "displacement",
     &                          ierror)
      if(ierror .ne. 0) return
      if(num_comp_check .ne. 3) return
      call aupst_get_node_var(num_nodes, num_comp_check,
     &       nodeID, cdispl, "displacement",
     &       ierror)
c
c  Extract the subroutine parameters.
c
      call aupst_get_real_param("char_length",
     &                          clength,error_code)
      call aupst_get_real_param("char_time",
     &                          ctime,error_code)
      call aupst_get_real_param("x_offset",xoff,error_code)
      call aupst_get_real_param("y_offset",yoff,error_code)
      call aupst_get_real_param("z_offset",zoff,error_code)
      call aupst_get_real_param("t_offset",toff,error_code)
      call aupst_get_real_param("u0",u0,error_code)
      call aupst_get_real_param("v0",v0,error_code)
      call aupst_get_real_param("w0",w0,error_code)
      call aupst_get_real_param("alpha",alpha,error_code)
      call aupst_get_real_param("num_nodes",
     &                          numnod,error_code)
c
c  Calculate a solution scaling factor
c
      expat = half * ( one - cos( pi * eval_time / ctime ) )
c
c  Compute the expected solution at each node and do a
c  sum of the differences from the analytic solution
c
      conv_error = zero
      do inode = 1, num_nodes
c
```

```
c  Set the displacement value from the manufactured solution
c
        x = ( mod_coords(1,inode) - xoff ) / clength
        y = ( mod_coords(2,inode) - yoff ) / clength
        z = ( mod_coords(3,inode) - zoff ) / clength
c
        mdisplx = u0 * sin(x) * cos(two*y) * cos(three*z)
     *               * expat
        mdisply = v0 * cos(three*x) * sin(y) * cos(two*z)
     *               * expat
        mdisplz = w0 * cos(two*x) * cos(three*y) * sin(z)
     *               * expat
c
        xdiff = mdisplx - cdispl(1,inode)
        ydiff = mdisply - cdispl(2,inode)
        zdiff = mdisplz - cdispl(3,inode)
        conv_error = conv_error + xdiff*xdiff
     *                          + ydiff*ydiff
     *                          + zdiff*zdiff
c
      enddo
c
      ierror = 0
c
c  Do a parallel sum of the squared errors and extract
c  the total summed value on all processors
c
      call aupst_put_global_var(1,conv_error,
     &                          "conv_error","sum",ierror)
      call aupst_get_global_var(1,conv_error,
     &                          "conv_error",ierror)
c
c  Take the square root of the errors and store that as
c  the net error norm
c
      conv_error = sqrt(conv_error) / sqrt(numnod)
      call aupst_put_global_var(1,conv_error,
     &                          "conv_error","none",ierror)
c
      return
      end
```

9.4.3 Transform Output Stresses to a Cylindrical Coordinate System

The following code is a user subroutine to transform element stresses in global x, y, and z coordinates to a global cylindrical coordinate system. This subroutine could be used to transform the relatively meaningless shell stress in x, y, and z coordinates to more meaningful tangential, hoop, and radial stresses. The subroutine is called from a `USER OUTPUT` command block. It reads in the old stresses, transforms them, and writes them back out to a user-created scratch variable, defined via a `USER VARIABLE` command block, for output.

```
begin user variable cyl_stress
  type = element sym_tensor length = 1
  initial value = 0.0 0.0 0.0 0.0 0.0 0.0
end user variable

begin user output
  block = block_1
  element block subroutine = aupst_cyl_transform
  subroutine string parameter: origin_point = Point_O
  subroutine string parameter: z_point      = Point_Z
  subroutine string parameter: xz_point     = Point_XZ
  subroutine string parameter: input_stress = memb_stress
  subroutine string parameter: output_stress = cyl_stress
end user output
```

The FORTRAN listing for the subroutine is as follows:

```
      subroutine aupst_cyl_transform(num_elems, num_vals,
     *  eval_time, elemID, values, flags, ierror)
      implicit none
#include<framewk/Fmwk_type_sizes_decl.par>
#include<framewk/Fmwk_type_sizes.par>
c
c  Subroutine Arguments
c
c  num_elems: Input: Number of elements to calculate on
c  num_vals : Input: Ignored
c  eval_time: Input: Time at which to evaluate the stress.
c  elemID   : Input: Global sierra IDs of the input elements
c  values   : I/O  : Ignored, stress will be stored manually
c  flags    : I/O  : Ignored
c  ierror   :Output: Returns non-zero if an error occurs
```

```
c
      integer num_elems
      integer num_vals
      double precision eval_time
      integer elemID(num_elems)
      double precision values(1)
      integer flags(1)
      integer ierror
c
c  Fortran cannot dynamically allocate memory, thus worksets
c  will be iterated over by  chucks each of size chunk_size.
c
      integer chunk_size
      parameter (chunk_size = 100)
      integer chunk_ids(chunk_size)
c
c  Subroutine parameter data
c
      character*80     origin_point_name
      double precision origin_point(3)
      character*80     z_point_name
      double precision z_point(3)
      character*80     xz_point_name
      double precision xz_point(3)
      character*80     input_stress_name
      character*80     output_stress_name
c
c  Local element data for centroids and rotation vectors
c
      double precision cent(3)
      double precision centerline_pos(3)
      double precision dot_prod
      double precision z_vec(3)
      double precision r_vec(3)
      double precision theta_vec(3)
      double precision rotation_tensor(9)
c
c  Chunk data storage
c
      double precision elem_centroid(3, chunk_size)
      double precision input_stress_val(6, chunk_size)
      double precision output_stress_val(6, chunk_size)
c
c  Simple iteration variables
c
```

```
      integer error_code
      integer ichunk, ielem
      integer zero_elem, nel
c
c  Extract the current subroutine parameters.  origin_point
c  is the origin of the coordinate system
c  z_point is a point on the z axis of the coordinate system
c  xz_point is a point on the xz plane
c
      call aupst_get_string_param("origin_point",
     &                            origin_point_name,
     &                            error_code)
      call aupst_get_string_param("z_point",
     &                            z_point_name,
     &                            error_code)
      call aupst_get_string_param("xz_point",
     &                            xz_point_name,
     &                            error_code)
      call aupst_get_string_param("input_stress",
     &                            input_stress_name,
     &                            error_code)
      call aupst_get_string_param("output_stress",
     &                            output_stress_name,
     &                            error_code)
c
c  Use the point names to look up the coordinates of each
c  relevant point
c
      call aupst_get_point(origin_point_name, origin_point,
     &                     error_code)
      call aupst_get_point(z_point_name, z_point,
     &                     error_code)
      call aupst_get_point(xz_point_name, xz_point,
     &                     error_code)
c
c  Compute the z axis vector
c
      z_vec(1) = z_point(1) - origin_point(1)
      z_vec(2) = z_point(2) - origin_point(2)
      z_vec(3) = z_point(3) - origin_point(3)
c
c  Transform z_vec into a unit vector, abort if it is invalid
c
      call aupst_unitize_vector(z_vec, ierror)
      if(ierror .ne. 0) return
```

```
c
c  Loop over chunks of the data arrays
c
      do ichunk = 1, (num_elems/chunk_size + 1)
c
c  Determine the first and last element number for the
c  current chunk of elements
c
       zero_elem = (ichunk-1) * chunk_size
       if((zero_elem + chunk_size) .gt. num_elems) then
        nel = num_elems - zero_elem
       else
        nel = chunk_size
       endif
c
c  Copy the elemIDs for all elems in the current chunk to a
c  temporary array
c
       do ielem = 1, nel
       chunk_ids(ielem) = elemID(zero_elem + ielem)
       enddo
c
c  Extract the element centroids and stresses
c
       call aupst_get_elem_centroid(nel, chunk_ids,
     &                              elem_centroid,
     &                              ierror)
       call aupst_get_elem_var(nel, 6, chunk_ids,
     &                         input_stress_val,
     &                         input_stress_name, ierror)
c
c  Loop over each element in the current chunk
c
       do ielem = 1, nel
c
c  Find the closest point on the cylinder centerline axis
c  to the element centroid
c
        cent(1) = elem_centroid(1, ielem) - origin_point(1)
        cent(2) = elem_centroid(2, ielem) - origin_point(2)
        cent(3) = elem_centroid(3, ielem) - origin_point(3)
        dot_prod = cent(1) * z_vec(1) +
     &             cent(2) * z_vec(2) +
     &             cent(3) * z_vec(3)
        centerline_pos(1) = z_vec(1) * dot_prod
```

```
         centerline_pos(2) = z_vec(2) * dot_prod
         centerline_pos(3) = z_vec(3) * dot_prod
c
c  Compute the current normal radial vector
c
         r_vec(1) = cent(1) - centerline_pos(1)
         r_vec(2) = cent(2) - centerline_pos(2)
         r_vec(3) = cent(3) - centerline_pos(3)
         call aupst_unitize_vector(r_vec, ierror)
         if(ierror .ne. 0) return
c
c  Compute the current hoop vector
c
         theta_vec(1) = z_vec(2)*r_vec(3) - r_vec(2)*z_vec(3)
         theta_vec(2) = z_vec(3)*r_vec(1) - r_vec(3)*z_vec(1)
         theta_vec(3) = z_vec(1)*r_vec(2) - r_vec(1)*z_vec(2)
c
c  The r, theta, and z vectors describe the new stress
c  coordinate system, Transform the input stress tensor
c  in x,y,z coords to the output stress tensor in r, theta,
c  and z coords use the unit vectors to create a rotation
c  tensor
c
         rotation_tensor(k_f36xx) = r_vec(1)
         rotation_tensor(k_f36yx) = r_vec(2)
         rotation_tensor(k_f36zx) = r_vec(3)
         rotation_tensor(k_f36xy) = theta_vec(1)
         rotation_tensor(k_f36yy) = theta_vec(2)
         rotation_tensor(k_f36zy) = theta_vec(3)
         rotation_tensor(k_f36xz) = z_vec(1)
         rotation_tensor(k_f36yz) = z_vec(2)
         rotation_tensor(k_f36zz) = z_vec(3)
c
c  Rotate the current stress tensor to the new configuration
c
         call fmth_rotate_symten33(1, 1, 0, rotation_tensor,
      &                           input_stress_val(1,ielem),
      &                          output_stress_val(1,ielem))
        enddo
c
c  Store the new stress
c
         call aupst_put_elem_var(nel, 6, chunk_ids,
      &                          output_stress_val,
      &                          output_stress_name, ierror)
      enddo
      ierror = 0
      end
```

9.5 User Subroutines: Library

A number of user subroutines are used commonly and have been permanently incorporated into the code. These subroutines are used just like any other subroutines, but they do not need to be compiled into the code. (The user need be concerned only about the Presto command lines.) This section describes the usage of each of these subroutines.

9.5.1 aupst_cyl_transform

Author: Nathan Crane

Purpose:

The purpose of this subroutine is to transform element stresses from a global rectangular coordinate system to a local cylindrical coordinate system. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = aupst_cyl_transform
  subroutine string parameter: origin_point = Point_O
  subroutine string parameter: z_point      = Point_Z
  subroutine string parameter: xz_point     = Point_XZ
  subroutine string parameter: input_stress = memb_stress
  subroutine string parameter: output_stress = cyl_stress
end user output
```

Requirements:

This subroutine requires a tensor variable to store the cylindrical stress into a registered variable for each element. The registered variable is created by the following command block in the Presto region:

```
begin user variable cyl_stress
   type = element sym_tensor length = 1
   initial value = 0.0 0.0 0.0 0.0 0.0 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
origin_point	String	Name of the point at the cylinder origin.
z_point	String	Point on the cylinder axis.
xz_point	String	Point on the line that passes through theta = 0 on the cylinder.
input_stress	String	Name of the Presto internal input stress tensor variable.
output_stress	String	Name of the Presto internal output stress tensor variable.

9.5.2 aupst_rec_transform

Author: Daniel Hammerand

Purpose:

The purpose of this subroutine is to transform element stresses from a global rectangular coordinate system to a different local rectangular coordinate system. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = aupst_rec_transform
  subroutine string parameter: origin_point = Point_O
  subroutine string parameter: z_point      = Point_Z
  subroutine string parameter: xz_point     = Point_XZ
  subroutine string parameter: input_stress = memb_stress
  subroutine string parameter: output_stress = new_stress
end user output
```

Requirements:

This subroutine requires a tensor variable to store the new stress into a registered variable for each element. The registered variable is created by the following command block in the Presto region:

```
begin user variable new_stress
  type = element sym_tensor length = 1
  initial value = 0.0 0.0 0.0 0.0 0.0 0.0
end user variable
```

Parameters:

Parameter Name	**Usage**	**Description**
origin_point	String	Name of the point at the cylinder origin.
z_point	String	Point on the cylinder axis.
xz_point	String	Point on the line that passes through theta = 0 on the cylinder.
input_stress	String	Name of the Presto internal input stress tensor variable.
output_stress	String	Name of the Presto internal output stress tensor variable.

Chapter 10

Shared Interface Models (SIMOD)

SIMOD is a modular third-party library that allows for the addition of surface physics in a modular fashion. As such, the emphasis of the library is not upon providing a comprehensive set of surface-physics models, but rather upon providing a place to "plug-in" new models that can be used by different application codes and their various "interface descriptors" (e.g., a contact algorithm or interface elements). As used here, surface physics implies the description of the "mechanical interaction" between two surfaces. A very simple surface-physics scheme is the linear elastic model that relates each component of the interfacial traction to its work-conjugate relative displacement through a single constant of proportionality (stiffness). A more-detailed surface-physics scheme could involve electrostatic attraction or even a model with history dependence.

SIMOD is used to construct various surface-physics models for contact, mixed boundary conditions, interface elements, and continuum elements with surfaces of discontinuity. Currently, for Presto, users can access SIMOD as one of several options for the traction force boundary condition described in Section 6.5.2. In the future, SIMOD will be accessible in conjunction with contact (Chapter 7).

The set of SIMOD interface models available for use in Presto is under development. These models are divided into two sets: test models and core models. Some of the test models are under development; others are being used to verify the generality of the SIMOD library (e.g., a model that is two-dimensional and includes internal state data). The core models are supported for production application and are listed as model options in Section 10.3. Users should contact James V. Cox (505-284-4816, jvcox@sandia.gov) for a complete list of interface models undergoing testing and development.

10.1 Defining a SIMOD Model

To use a SIMOD interface model, you must include a `SIMOD MODEL` command block in the domain scope. The SIMOD model can then be referenced in a traction force boundary condition via the `TRACTION` command block. The format of the `SIMOD MODEL` command block follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = <string>model_name
  BOOLEAN PARAMETER: <string>bool_name = <boolean>bool_val
  CHARACTER PARAMETER: <string>char_name = <string>char_val
  REAL PARAMETER: <string>real_name = <real>real_val
  INTEGER PARAMETER: <string>int_name = <integer>int_val
  STRING PARAMETER: <string>string_name = <string>str_name
  MODEL PARAMETER: <string>param_name = <string>str_name
  FUNCTION PARAMETER: <string>func_name =
    <string>sierra_function
END [SIMOD MODEL <string>some_name]
```

The string `some_name` is a user-selected name for the SIMOD model. A particular interface model is selected by using the `MODEL TYPE` command line. The current list of choices for the model type (of core models only) are

- `Composite2d_via_1d`,
- `Composite3d_via_1d`,
- `Concrete_Exp1d`,
- `Elastic_IM`,
- `Electrostatic_ParallelPL`,
- `Hamaker_ParallelPL`,
- and `null_IM`.

Each interface model has a specific set of parameters that are used to define the model. The parameters are set by some combination of the parameter command lines in the above command block. The various parameter names correspond to SIMOD names. For example, if you selected model type `Elastic_IM` and that model type required the real parameter `k`, the `SIMOD MODEL` command block would appear as follows:

```
BEGIN SIMOD MODEL elastic_im_bond1
  MODEL TYPE = Elastic_IM
  REAL PARAMETER: k = 1.0e6
END SIMOD MODEL
```

10.2 Use of a SIMOD Model with the Traction Boundary Condition

SIMOD models referenced for traction force boundary conditions must be fully three-dimensional. The dimension of an interface model is defined as the number of traction components that are related to their work-conjugate kinematic variables (e.g., relative displacements). A three-dimensional model can be built up from one-dimensional models via a SIMOD composite model. Please reference the SIMOD Application Programmer Interface [1] regarding the use of composite models. A null SIMOD model named `null` is automatically defined by SIERRA for use with traction force boundary conditions. For example, by using the `Composite3d_via_1d` model you may use the adhesion model (`Hamaker_ParallelPL`) for the normal response and two null models for the tangential response components.

When using the traction boundary condition option, the surface interaction is between some user-defined plane and a surface on the mesh. There may be a preferential direction in the user-defined plane. For example, a stick-slip condition interface may show certain behavior along the preferential direction, a different behavior along a direction normal to the preferential direction, and some combination of the two (preferential and normal) behaviors for any direction in between.

The command block for the traction boundary conditions appears in the region scope. If we want to use a SIMOD model, it is necessary to introduce a `TRACTION` command block of the following form:

```
BEGIN TRACTION
  USE SIMOD MODEL = <string>some_name
  SURFACE = <string list>surface_names
  REFERENCE PLANE AXIS = <string>axis_name
  REFERENCE PLANE T1 DIRECTION = <string>t1_direction
END [TRACTION]
```

The specific model is selected with the `USE SIMOD MODEL` command line. The string `some_name` on this command line references the string used to name a `SIMOD MODEL` command block. The surface on the mesh is defined with the `SURFACE` command line. The plane with which the surface of the mesh interacts is defined with the `REFERENCE PLANE AXIS` command line. The string `axis_name` uses an `axis_name` that is defined in the domain scope via a `DEFINE AXIS` command line. An axis definition (rather than a simple direction definition) is required to define the plane. An axis definition includes both a point and a direction. The point lies in the plane, and the direction defines the outward normal to the plane. If there is a preferential direction for model behavior in the plane, this direction is defined with the `REFERENCE`

`PLANE T1` command line. The string `t1_direction` uses a `direction_name` that is defined in the domain scope via a `DEFINE DIRECTION` command line.

10.3 SIMOD Core Models

The specific interface models that are currently available to users (the core models) are described in the following subsections in alphabetical order. Each model type listed includes a complete list of the required parameters. Again, other models (the test models) are available, but their support is limited. As mentioned previously, all these core models use the `SIMOD MODEL` command block as their format. The type of the model must be specified in the `MODEL TYPE` command line.

The units for various parameters in the SIMOD models are dependent upon the units you have selected for your analysis model. In the following model description, F represents force, L represents length, and V represents voltage.

10.3.1 Composite2d_via_1d

The `Composite2d_via_1d` model is a two-dimensional model that is composed of two one-dimensional models: a tangent response model and a normal response model. By definition these two components of the response are uncoupled, i.e., the normal response has no dependence upon the tangent response and visa versa. The format of the `SIMOD MODEL` command block for the `Composite2d_via_1d` model is as follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Composite2d_via_1d
  MODEL PARAMETER: tangent_model = <string>name_tangent
  MODEL PARAMETER: normal_model = <string>name_normal
END [SIMOD MODEL <string>some_name]
```

In the above command block:

- The string `some_name` is a user-selected name for the model.
- The string `name_tangent` specifies the name of a one-dimensional SIMOD model that will be applied to define the tangent response. This string can be (1) the name of a previously user-defined one-dimensional SIMOD model or (2) the name of the default null model, i.e., `null`.
- The string `name_normal` specifies the name of a SIMOD model that will be applied to define the normal response. This string can be (1) the name of a previously user-defined one-dimensional SIMOD model or (2) the name of the default null model, i.e., `null`.

If a `MODEL PARAMETER` command line is omitted, the `null` model is assumed. See Section 10.4 for an example of constructing a composite model with the SIMOD library.

10.3.2 Composite3d_via_1d

The `Composite3d_via_1d` model is a three-dimensional model that is composed of three one-dimensional models: a tangent response model for the $\mathtt{t}_1$ direction, a tangent response model for the $\mathtt{t}_2$ direction (where $\mathtt{t}_1 \perp \mathtt{t}_2$), and a normal response model. By definition, these three components of the response are uncoupled. The format of the `SIMOD MODEL` command block for the `Composite3d_via_1d` model is as follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Composite3d_via_1d
  MODEL PARAMETER: tangent1_model = <string>name_tangent1
  MODEL PARAMETER: tangent2_model = <string>name_tangent2
  MODEL PARAMETER: normal_model = <string>name_normal
END [SIMOD MODEL <string>some_name]
```

In the above command block:

- The string `some_name` is a user-selected name for the model.
- The string `name_tangent1` specifies the name of a one-dimensional SIMOD model that will be applied to define the tangent response in the $\mathtt{t}_1$ direction. This string can be (1) the name of a previously user-defined one-dimensional SIMOD model or (2) the name of the default null model, i.e., `null`.
- The string `name_tangent2` specifies the name of a one-dimensional SIMOD model that will be applied to define the tangent response in the $\mathtt{t}_2$ direction. This string can be (1) the name of a previously user-defined one-dimensional SIMOD model or (2) the name of the default null model, i.e., null.
- The string `name_normal` specifies the name of a one-dimensional SIMOD model that will be applied to define the normal response. This string can be (1) the name of a previously user-defined one-dimensional SIMOD model or (2) the name of the default null model, i.e., `null`.

If a `MODEL PARAMETER` command line is omitted, the null model is assumed. See Section 10.4 for an example of constructing a composite model with the SIMOD library.

10.3.3 Concrete_Exp1d

The `Concrete_Exp1d` model is a one-dimensional model for concrete tensile response across a surface, typically used in cohesive zone modeling of concrete fracture. The underlying assumptions are that (1) failure is by mode I fracture, (2) compression and elastic tensile loading can be represented by an elastic constant, and (3) the process zone can be mathematically represented by a strong discontinuity. The character string "Exp" in the model name denotes that an exponential function is used to represent the traction-separation softening response. The format of the `SIMOD MODEL` command block for the `Concrete_Exp1d` model is as follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Concrete_Exp1d
  REAL PARAMETER: Ec = <real>real_val
  REAL PARAMETER: sigt = <real>real_val
  REAL PARAMETER: wo = <real>real_val
  REAL PARAMETER: Gf = <real>real_val
  REAL PARAMETER: c2 = <real>real_val
END [SIMOD MODEL <string>some_name]
```

In the above command block:

- The name `some_name` is a user-selected name for the model.
- The parameter `Ec` is the elastic constant for the interface, in units of F/L^3.
- The parameter `sigt` is the tensile failure stress, in units of F/L^2.
- The parameter `wo` is the relative displacement at which the traction unloads to zero, in units of L.
- The parameter `Gf` is the fracture energy—the energy required to create a unit area of "unbonded surface," in units of $FL/L^2 = F/L$. Graphically, it is the area under the curve for the traction-separation law.
- The parameter `c2` is the exponential softening parameter—a unitless user-defined fitting parameter that affects the shape of the traction-separation curve.

10.3.4 Elastic_IM

The `Elastic_IM` model is a one-, two-, or three-dimensional, linear, isotropic, elastic interface model. Each relative displacement component multiplied by the elastic constant gives the corresponding work-conjugate traction. The format of the `SIMOD MODEL` command block for the `Elastic_IM` model is as follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Elastic_IM
  REAL PARAMETER: k = <real>real_val
END [SIMOD MODEL <string>some_name]
```

In the above command block:

- The name `some_name` is a user-selected name for the model.
- The parameter `k` is the elastic constant—the interface stiffness—in units of F/L^3.

10.3.5 Electrostatic_ParallelPL

The `Electrostatic_ParallelPL` model is a simple parallel plate model for electrostatic attraction or repulsion. The model is one-dimensional and relates traction to the distance of separation. The format of the `SIMOD MODEL` command block for the `Electrostatic_ParallelPL` model is as follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Electrostatic_ParallelPL
  REAL PARAMETER: zero_offset = <real>real_val
  REAL PARAMETER: clip_distance = <real>real_val
  REAL PARAMETER: scale_factor = <real>real_val
  REAL PARAMETER: permittivity = <real>real_val
  FUNCTION PARAMETER: voltage_function =
    <string>sierra_function
END [SIMOD MODEL <string>some_name]
```

In the above command block:

- The name `some_name` is a user-selected name for the model.
- The parameter `zero_offset` is the position of the reference plane for zero gap, in units of L. This parameter has two purposes: it avoids singularity of the rational function and sets the adhesion energy. Warning: The sign of this term may change in a future release.
- The parameter `clip_distance` is the distance to clip of a singular function, in units of L. This distance value must be less than the value of the parameter `zero_offset`.
- The parameter `scale_factor` (unitless) indicates attraction or repulsion, where a positive number implies attraction and a negative number implies repulsion.
- The parameter `permittivity` is the permittivity constant of the dielectric, in units of F/V^2.
- The parameter `voltage_function` refers to the name of a SIERRA function that defines voltage as a function of time. This function must be defined in the domain scope in a `DEFINITION FOR FUNCTION` command block.

10.3.6 Hamaker_ParallelPL

The `Hamaker_ParallelPL` model is a simple parallel plate model for adhesion associated with van der Waals forces. The model is one-dimensional and relates traction to the distance of separation. The format of the `SIMOD MODEL` command block for the `Hamaker_ParallelPL` model is as follows:

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Hamaker_ParallelPL
  REAL PARAMETER: zero_offset = <real>real_val
  REAL PARAMETER: clip_distance = <real>real_val
  REAL PARAMETER: A = <real>real_val
  REAL PARAMETER: permittivity = <real>real_val
  FUNCTION PARAMETER: time_function =
    <string>sierra_function
END [SIMOD MODEL <string>some_name]
```

In the above command block:

- The name `some_name` is a user-selected name for the model.
- The parameter `zero_offset` is the position of the reference plane for zero gap, in units of L. This parameter has two purposes: it avoids singularity of the rational function and sets the adhesion energy. Warning: The sign of this term may change in a future release.
- The parameter `clip_distance` is the distance to clip of a singular function in units of L. This distance value must be less than the value of the parameter `zero_offset`.
- The parameter `A` is the Hamaker constant, in units of FL.
- The parameter `time_function` refers to the name of a SIERRA function that defines a scalar multiplier as a function of time. This function must be defined in the domain scope in a `DEFINITION FOR FUNCTION` command block. The time function is used to facilitate convergence in quasi-static analyses. For explicit, transient dynamic analyses, the time function should reference a function with a value of unity for all times.

10.3.7 null_IM

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = null_IM
END [SIMOD MODEL <string>some_name]
```

In addition to the default null model `null`, you can also define a null model using a model type of `null_IM`. The `null_IM` model is a one-dimensional model that always gives a value of zero traction. The `null_IM` model may be referenced in a `MODEL PARAMETER` command line in the composite models. The model will be referenced by using the value `some_name` you have specified on the `BEGIN SIMOD MODEL` command line.

10.4 Example of Constructing a Composite Model

This section explains how to construct a `Composite2d_via_1d` model. This explanation also applies to the construction of a `Composite3d_via_1d` model, though the particulars of the model parameters may differ in the `Composite3d_via_1d` model.

For reference, the format of the `SIMOD MODEL` command block for constructing the `Composite2d_via_1d_model` is as follows:

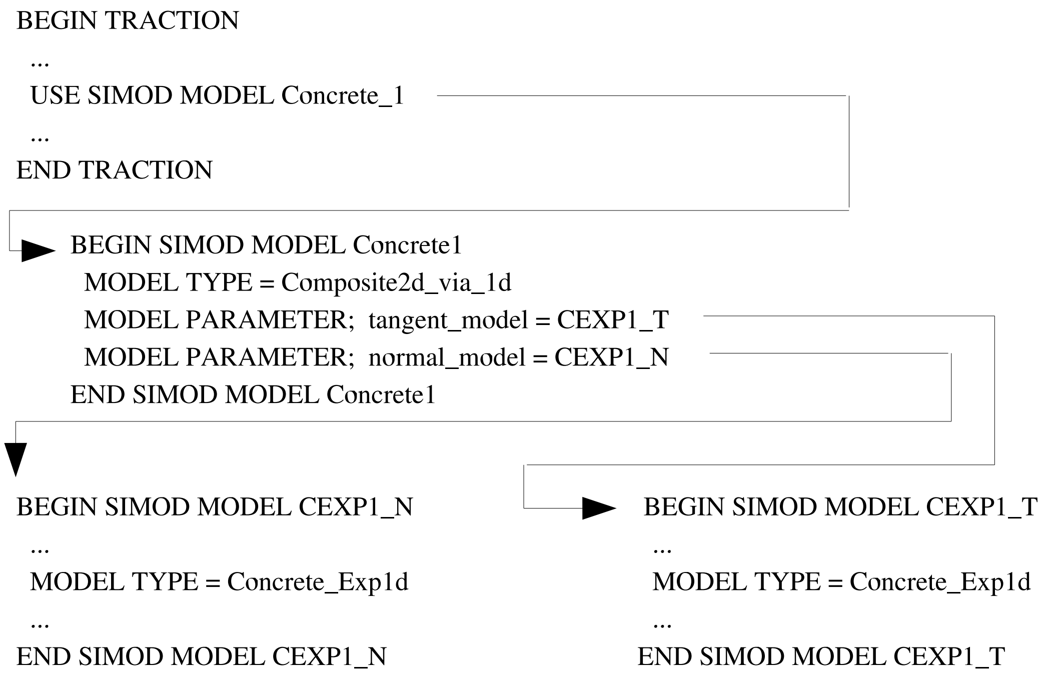
```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Composite2d_via_1d
  MODEL PARAMETER: tangent_model = <string>name_tangent
  MODEL PARAMETER: normal_model = <string>name_normal
END [SIMOD MODEL <string>some_name]
```

The previously defined SIMOD models are models that have been defined by the user with a `SIMOD MODEL` command block. The string `name_tangent` or `name_normal` in a `MODEL PARAMETER` command line will reference the user-defined name, `some_name`, of another `SIMOD MODEL` command block. For example, suppose we are constructing a `Composite2d_via_1d` model that is composed of two `Concrete_Exp1d` models. One of the models, which we will assign the user-defined name of `CEXP1_N`, will define behavior in the normal direction; the other model, which we will assign the user-defined name of `CEXP1_T`, will define behavior in the tangential direction. If the name of our composite model is `Concrete1`, then our command block for this composite model is as follows:

```
BEGIN SIMOD MODEL Concrete1
  MODEL TYPE = Composite2d_via_1d
  MODEL PARAMETER: tangent_model = CEXP1_T
  MODEL PARAMETER: normal_model = CEXP1_N
END SIMOD MODEL Concrete1
```

The user-defined name `Concrete1` will be referenced by a traction boundary condition specified with a `TRACTION` command block in the region.

Figure 10.1 highlights the relationships involved in the construction of this type of composite model.



```
BEGIN TRACTION
  ...
  USE SIMOD MODEL Concrete_1
  ...
END TRACTION

    BEGIN SIMOD MODEL Concrete1
      MODEL TYPE = Composite2d_via_1d
      MODEL PARAMETER;  tangent_model = CEXP1_T
      MODEL PARAMETER;  normal_model = CEXP1_N
    END SIMOD MODEL Concrete1

BEGIN SIMOD MODEL CEXP1_N                    BEGIN SIMOD MODEL CEXP1_T
  ...                                          ...
  MODEL TYPE = Concrete_Exp1d                  MODEL TYPE = Concrete_Exp1d
  ...                                          ...
END SIMOD MODEL CEXP1_N                      END SIMOD MODEL CEXP1_T
```

Figure 10.1: Relationships among command blocks in constructing a composite SIMOD model.

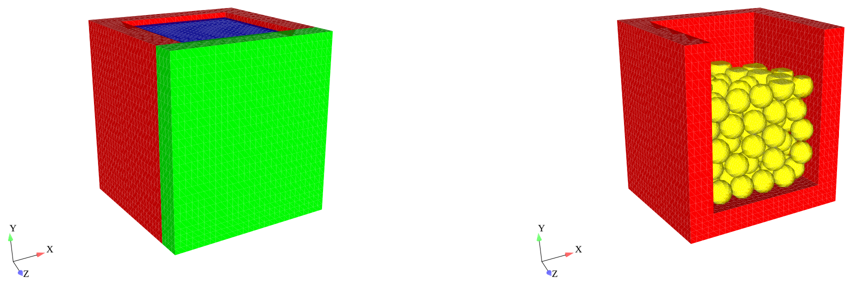
10.5 References

1. Cox, J. V. “Shared Interface Models - SIMOD Application Programmer Interface.” Draft Report, Department 9123, Sandia National Laboratories.

Chapter 11

Example Problem

This chapter provides an example problem to illustrate the construction of an input file for an analysis. The example problem consists of 124 spheres made of lead enclosed in a steel box. The steel box has an open top into which a steel plate is placed (see Figure 11.1). A prescribed velocity is then applied on the steel plate, pushing it into the box and crushing the spheres contained within using frictionless contact. This problem is a severe test for the contact algorithms as the spheres crush into a nearly solid block. See Figure 11.2 for results of this problem.

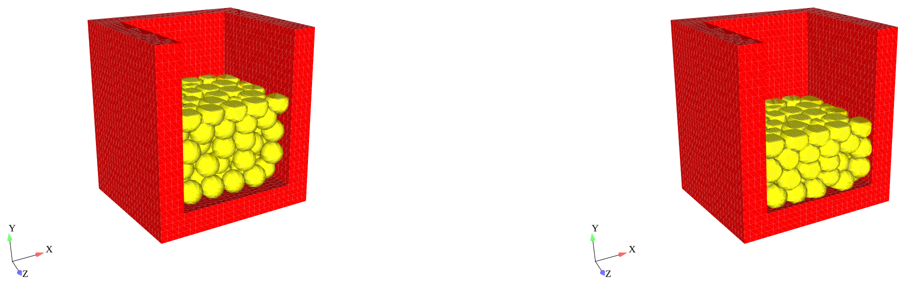


(a) Undeformed Mesh (b) Initial Crush of Spheres

Figure 11.1: Mesh for example problem: (a) box (red and green surfaces) with plate in top (blue surface) and (b) mesh with blue and green surfaces removed to show internal spheres (yellow) with initial crush.

The input file is described below, with comments to explain every few lines. Following the description, the full input file is listed again. Most of the key words in this example are all lowercase, which is different from the convention we have used to describe the command lines in this document. However, all the lowercase usage in the following example is an acceptable format in Presto.

The input file starts with a begin sierra statement (i.e., the first line of the `SIERRA`



(a) Additional Crush of Spheres (b) Final Deformed Configuration

Figure 11.2: Mesh with blue and green surfaces removed to show internal spheres (yellow) after initial crush shown in Figure 11.1 (b).

command block), as is required for all input files:

```
begin sierra crush_124_spheres
```

We now need to define the functions used with this problem. The boundary conditions require a function for the initial velocity, as follows:

```
begin definition for function constant_velocity
  type is piecewise linear
  ordinate is velocity
  abscissa is time
  begin values
    0.0 30.0
    1.0 30.0
  end values
end definition for function constant_velocity
```

To define the boundary conditions, we need to define the direction for the initial velocity—this is in the y-direction. We could also choose to simply specify the Y component for the initial condition, but this input file uses directions.

```
define direction y_axis with vector 0.0 1.0 0.0
```

Next we define the material models that will be used for this analysis. There are two materials in this problem: steel for the box, and lead for the spheres. Both materials use the elastic-plastic material model (denoted as `elastic_plastic`).

```
begin property specification for material steel
  density = 7871.966988

  begin parameters for model elastic_plastic
    youngs modulus = 1.999479615e+11
    poissons ratio = 0.33333
    yield stress = 275790291.7
    hardening modulus = 275790291.7
    beta = 1.0
  end parameters for model elastic_plastic

end property specification for material steel

begin property specification for material lead
  density = 11253.30062

  begin parameters for model elastic_plastic
    youngs modulus = 1.378951459e+10
    poissons ratio = 0.44
    yield stress = 13789514.59
    hardening modulus = 0.0
    beta = 1.0
  end parameters for model elastic_plastic

end property specification for material lead
```

Now we define the finite element mesh. This includes specification of the file that contains the mesh, as well as a list of all the element blocks we will use from the mesh and the material associated with each block. The name of the file is `crush_124_spheres.g`. The specification of the database type is optional—ExodusII is the default. Currently, each element block must be defined individually. For this particular problem, all the spheres are the same element block. Each sphere is a distinct geometry entity, but all spheres constitute one element block in the Exodus II database. Note that the three element blocks that make up the box and lid all reference the same material description. The material description is *not* repeated three times. The material description for steel appears once and is then referenced three times.

```
begin finite element model mesh1
  Database Name = crush_124_spheres.g
  Database Type = exodusII

  begin parameters for block block_1
    material linear_elastic_steel
```

```
      solid mechanics use model elastic_plastic
    end parameters for block block_1

    begin parameters for block block_2
      material linear_elastic_steel
      solid mechanics use model elastic_plastic
    end parameters for block block_2

    begin parameters for block block_3
      material linear_elastic_steel
      solid mechanics use model elastic_plastic
    end parameters for block block_3

    begin parameters for block block_4
      material linear_elastic_lead
      solid mechanics use model elastic_plastic
    end parameters for block block_4

  end finite element model mesh1
```

As an alternative to referencing the material description for steel three times as done above, you could define multiple element blocks simultaneously on the same command line. Thus, the three element block specifications with the material `linear_elastic_steel` could be consolidated into one, as follows:

```
begin parameters for block block_1 block 2 block 3
  material linear_elastic_steel
  solid mechanics use model elastic_plastic
end parameters for block block_1 block 2 block 3
```

At this point we have finished specifying physics-independent quantities. We now want to set up the Presto procedure and region, along with the `time control` command block. We start by defining the beginning of the procedure scope, the `time control` command block, and the beginning of the region scope. Only one `time stepping block` command block is needed for this analysis. The termination time is set to 7×10^{-4}.

```
begin presto procedure Apst_Procedure

begin time control
  begin time stepping block p1
    start time = 0.0
    begin parameters for presto region presto
```

```
        time step scale factor = 1.0
        time step increase factor = 2.0
        step interval = 25
      end parameters for presto region presto
    end time stepping block p1

    termination time = 7.0e-4
  end time control

    begin presto region presto
```

Next we associate the finite element model we defined above (mesh1) with this presto region.

```
      use finite element model mesh1
```

Now we define the boundary conditions on the problem. We prescribe the velocity on the top surface of the box (`nodelist_100`) to crush the spheres, and we confine the bottom surface of the box (`nodelist_200`) not to move. Note that although we use node sets to define these boundary conditions, we could have used the corresponding side sets.

```
      begin prescribed velocity
        node set = nodelist_100
        direction = y_axis
        function = constant_velocity
        scale factor = -1.0
      end

      begin fixed displacement
        node set = nodelist_200
        components = Y
      end
```

Now we define the contact for this problem. For this problem, we want all four element blocks to be able to contact each other, with a normal tolerance of 0.0001 and a tangential tolerance of 0.0005. In this case, we simply define the same contact characteristics for all interactions. However, we could also specify tolerances and kinematic partition factors for individual interactions. Since no friction model is defined in the block below, the contact defaults to frictionless contact. (There are numerous options you can use to control the contact algorithm. The options you choose will affect contact algorithm efficiency and solution accuracy. Consult with Chapter 7 to determine how to set input for the `CONTACT DEFINITION` command block to obtain the best level of efficiency and accuracy for your particular problem.)

```
      begin contact definition
        skin all blocks = on
         begin search options
           normal tolerance = 0.0001
           tangential tolerance = 0.00005
         end
         begin interaction defaults
           general contact = on
           self contact = on
         end
       end
```

Now we define what variables we want in the results file, as well as how often we want this file to be written. Here we request files written every 7×10^{-6} sec of analysis time. This will result in results output at one hundred time steps (plus the zero time step) since the termination time is set to 7×10^{-4} sec. The output file will be called `crush_124_spheres.e`, and it will be an Exodus II file (the database type command is optional; it defaults to ExodusII). The variables we are requesting are the displacements and external forces at the nodes, the rotated stresses for the elements, the time-step increment, and the kinetic energy.

```
       begin Results Output output_presto
         Database Name = crush_124_spheres.e
         Database Type = exodusII
         At Time 0.0, Increment = 7.0e-6
         nodal Variables = displacement as displ
         nodal Variables = force_external as fext
         element Variables = rotated_stress as stress
         global Variables = KineticEnergy as KE
         global Variables = timestep
       end
```

Now we end the presto region, presto procedure, and sierra blocks to complete the input file.

```
     end presto region presto
   end presto procedure Apst_Procedure
 end sierra crush_124_spheres
```

Here is the resulting full input file for this problem:

```
begin sierra crush_124_spheres
  begin definition for function constant_velocity
    type is piecewise linear
    ordinate is velocity
    abscissa is time
    begin values
      0.0 30.0
      1.0 30.0
    end values
  end definition for function constant_velocity
  define direction y_axis with vector 0.0 1.0 0.0

  begin property specification for material steel
    density = 7871.966988

    begin parameters for model elastic_plastic
      youngs modulus = 1.999479615e+11
      poissons ratio = 0.33333
      yield stress = 275790291.7
      hardening modulus = 275790291.7
      beta = 1.0
    end parameters for model elastic_plastic

  end property specification for material steel

  begin property specification for material lead
    density = 11253.30062

    begin parameters for model elastic_plastic
      youngs modulus = 1.378951459e+10
      poissons ratio = 0.44
      yield stress = 13789514.59
      hardening modulus = 0.0
      beta = 1.0
    end parameters for model elastic_plastic

  end property specification for material lead

  begin finite element model mesh1
    Database Name = crush_124_spheres.g
    Database Type = exodusII

    begin parameters for block block_1
      material linear_elastic_steel
      solid mechanics use model elastic_plastic
    end parameters for block block_1

    begin parameters for block block_2
      material linear_elastic_steel
      solid mechanics use model elastic_plastic
    end parameters for block block_2

    begin parameters for block block_3
      material linear_elastic_steel
      solid mechanics use model elastic_plastic
    end parameters for block block_3

    begin parameters for block block_4
      material linear_elastic_lead
      solid mechanics use model elastic_plastic
    end parameters for block block_4

  end finite element model mesh1

  begin presto procedure Apst_Procedure

    begin time control
      begin time stepping block p1
        start time = 0.0
        begin parameters for presto region presto
          time step scale factor = 1.0
          time step increase factor = 2.0
          step interval = 25
        end parameters for presto region presto
      end time stepping block p1

      termination time = 7.0e-4
    end time control

    begin presto region presto

      use finite element model mesh1

      begin prescribed velocity
        node set = nodelist_100
        direction = y_axis
        function = constant_velocity
        scale factor = -1.0
      end prescribed velocity
      begin fixed displacement
        node set = nodelist_200
        components = Y
      end fixed displacement

     begin contact definition
       skin all blocks = on
        begin search options
          normal tolerance = 0.0001
          tangential tolerance = 0.00005
        end
        begin interaction defaults
          general contact = on
          self contact = on
        end
      end

      begin Results Output output_presto
        Database Name = crush_124_spheres.e
        Database Type = exodusII
        At Time 0.0, Increment = 7.0e-6
        nodal Variables = displacement as displ
        nodal Variables = force_external as fext
        element Variables = rotated_stress as stress
        global Variables = KineticEnergy as KE
        global Variables = timestep
      end results output output_presto

    end presto region presto
  end presto procedure Apst_Procedure
end sierra crush_124_spheres
```

Intentionally Left Blank

Chapter 12

Command Summary

This chapter gives all of the Presto commands in the proper scope.

```
# Domain specification
BEGIN SIERRA <string>name

  # Title

    TITLE = <string list>title

  # Restart time

    RESTART TIME = <real>restart_time
    RESTART = AUTOMATIC

  # User subroutine file

  USER SUBROUTINE FILE = <string>file name

  # Function definition

  BEGIN DEFINITION FOR FUNCTION <string>function_name
    TYPE = <string>CONSTANT|PIECEWISE LINEAR|
      ANALYTIC
    ABSCISSA = <string>abscissa_label
    ORDINATE = <string>ordinate_label
    BEGIN VALUES
      <real>value_1   [<real>value_2
      <real>value_3    <real>value_4
      ...              <real>value_n]
    END [VALUES]
    EVALUATE EXPRESSION = <string>analytic_expression1;
      analytic_expression2;...
  END [DEFINITION FOR FUNCTION <string>function_name]

  # Definitions

  DEFINE POINT <string>point_name WITH COORDINATES
    <real>value_1 <real>value_2 <real>value_3

  DEFINE DIRECTION <string>direction_name WITH VECTOR
    <real>value_1 <real>value_2 <real>value_3

  DEFINE AXIS <string>axis_name WITH POINT
    <string>point_1 POINT <string>point_2

  DEFINE AXIS <string>axis_name WITH POINT
    <string>point_name DIRECTION <string>direction

  # Local coordinate system

  BEGIN ORIENTATION <string>orientation_name
    SYSTEM = <string>RECTANGULAR|Z RECTANGULAR|CYLINDRICAL|
      SPHERICAL(RECTANGULAR)
    #
    POINT A = <real>global_ax <real>global_ay <real>global_az
    POINT B = <real>global_bx <real>global_by <real>global_bz
    #
    ROTATION ABOUT <integer> 1|2|3(1) = <real>theta(0.0)
  END [ORIENTATION <string>orientation_name]

  # Error estimator controller

  BEGIN ERROR ESTIMATION CONTROLLER <string>err_name
    ERROR ESTIMATOR = <string>DISTORTION
    COMPUTE METRIC = <string>ASPECT_RATIO/SOLID_ANGLE
      /PERIMETER_RATIO
    COMPUTE STEP INTERVAL = <integer>step_int
    COMPUTE AT OUTPUT
  END [ERROR ESTIMATION CONTROLLER <string>name]

  # Elastic material

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL ELASTIC
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
    END [PARAMETERS FOR MODEL ELASTIC]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Elastic fracture material

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL ELASTIC_FRACTURE
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      MAX STRESS = <real>max_stress
      CRITICAL STRAIN = <real>critical_strain
    END [PARAMETERS FOR MODEL ELASTIC_FRACTURE]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

```
# Elastic-plastic material

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD STRESS = <real>yield_stress
    HARDENING MODULUS = <real>hardening_modulus
    BETA = <real>beta_parameter(1.0)
  END [PARAMETERS FOR MODEL ELASTIC_PLASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Elastic-plastic power-law hardening

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL EP_POWER_HARD
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
```

```
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      YIELD STRESS = <real>yield_stress
      HARDENING CONSTANT = <real>hardening_constant
      HARDENING EXPONENT = <real>hardening_exponent
      LUDERS STRAIN = <real>luders_strain
    END [PARAMETERS FOR MODEL EP_POWER_HARD]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Elastic plastic power-law hardening with failure

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL DUCTILE_FRACTURE
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      YIELD STRESS = <real>yield_stress
      HARDENING CONSTANT = <real>hardening_constant
      HARDENING EXPONENT = <real>hardening_exponent
      LUDERS STRAIN <real>luders_strain
      CRITICAL TEARING PARAMETER = <real>crit_tearing
      CRITICAL CRACK OPENING STRAIN = <real>crit_crack
    END [PARAMETERS FOR MODEL DUCTILE_FRACTURE]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Multilinear elastic plaster power-law hardening with
  # failure

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL ML_EP_FAIL
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      YIELD STRESS = <real>yield_stress
      BETA = <real>beta_parameter(1.0)
      HARDENING FUNCTION = <real>hardening_function_name
      YOUNGS MODULUS FUNCTION = <real>ym_function_name
      POISSONS RATIO FUNCTION = <real>pr_function_name
      YIELD STRESS FUNCTION =
        <real>yield_stress_function_name
      CRITICAL TEARING PARAMETER = <real>crit_tearing
      CRITICAL CRACK OPENING STRAIN = <real>crit_crack
    END [PARAMETERS FOR MODEL ML_EP_FAIL]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # BCJ plasticity

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL BCJ
```

```
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      C1 = <real>c1
      C2 = <real>c2
      C3 = <real>c3
      C4 = <real>c4
      C5 = <real>c5
      C6 = <real>c6
      C7 = <real>c7
      C8 = <real>c8
      C9 = <real>c9
      C10 = <real>c10
      C11 = <real>c11
      C12 = <real>c12
      C13 = <real>c13
      C14 = <real>c14
      C15 = <real>c15
      C16 = <real>c16
      C17 = <real>c17
      C18 = <real>c18
      C19 = <real>c19
      C20 = <real>c20
      DAMAGE EXPONENT = <real>damage_exponent
      INITIAL ALPHA_XX = <real>alpha_xx
      INITIAL ALPHA_YY = <real>alpha_yy
      INITIAL ALPHA_ZZ = <real>alpha_zz
      INITIAL ALPHA_XY = <real>alpha_xy
      INITIAL ALPHA_YZ = <real>alpha_yz
      INITIAL ALPHA_XZ = <real>alpha_xz
      INITIAL KAPPA = <real>initial_kappa
      INITIAL DAMAGE = <real>initial_damage
      YOUNGS MODULUS FUNCTION = <string>ym_function_name
      POISSONS RATIO FUNCTION = <string>pr_function_name
      SPECIFIC HEAT = <real>specific_heat
      THETA OPT = <integer>theta_opt
      FACTOR = <real>factor
      RHO = <real>rho
      TEMP0 = <real>temp0
    END [PARAMETERS FOR MODEL BCJ]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Soil and crushable foam
```

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL SOIL_FOAM
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    A0 = <real>const_coeff_yieldsurf
    A1 = <real>lin_coeff_yieldsurf
    A2 = <real>quad_coeff_yieldsurf
    PRESSURE CUTOFF = <real>pressure_cutoff
    PRESSURE FUNCTION = <string>function_press_volstrain
  END [PARAMETERS FOR MODEL SOIL_FOAM]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name

# Foam plasticity

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL FOAM_PLASTICITY
    YOUNGS MODULUS = <real>youngs_modulus
```

```
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      PHI = <real>phi
      SHEAR STRENGTH  = <real>shear_strength
      SHEAR HARDENING = <real>shear_hardening
      SHEAR EXPONENT  = <real>shear_exponent
      HYDRO STRENGTH  = <real>hydro_strength
      HYDRO HARDENING = <real>hydro_hardening
      HYDRO EXPONENT  = <real>hydro_exponent
      BETA = <real>beta
    END [PARAMETERS FOR MODEL FOAM_PLASTICITY]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Elastic three-dimensional orthotropic
  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
      YOUNGS MODULUS AA = <real>Eaa_value
      YOUNGS MODULUS BB = <real>Ebb_value
      YOUNGS MODULUS CC = <real>Ecc_value
      POISSONS RATIO AB = <real>NUab_value
      POISSONS RATIO BC = <real>NUbc_value
      POISSONS RATIO CA = <real>NUca_value
      SHEAR MODULUS AB = <real>Gab_value
      SHEAR MODULUS BC = <real>Gbc_value
      SHEAR MODULUS CA = <real>Gca_value
      COORDINATE SYSTEM = <string>coordinate_system_name
      DIRECTION FOR ROTATION = <real>1|2|3
      ALPHA = <real>alpha_in_degrees
      SECOND DIRECTION FOR ROTATION = <real>1|2|3
      SECOND ALPHA = <real>second_alpha_in_degrees
      THERMAL STRAIN AA FUNCTION = <string>ethaa_function_name
      THERMAL STRAIN BB FUNCTION = <string>ethbb_function_name
      THERMAL STRAIN CC FUNCTION = <string>ethcc_function_name
    END [PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Orthotropic crush

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      YIELD STRESS = <real>yield_stress
      EX  = <real>modulus_x
      EY  = <real>modulus_y
      EZ  = <real>modulus_z
      GXY = <real>shear_modulus_xy
      GYZ = <real>shear_modulus_yz
      GZX = <real>shear_modulus_zx
      VMIN = <real>min_crush_volume
      CRUSH XX = <string>stress_volume_xx_function_name
      CRUSH YY = <string>stress_volume_yy_function_name
      CRUSH ZZ = <string>stress_volume_zz_function_name
      CRUSH XY =
        <string>shear_stress_volume_xy_function_name
      CRUSH YZ =
        <string>shear_stress_volume_yz_function_name
      CRUSH ZX =
        <string>shear_stress_volume_zx_function_name
    END [PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Orthotropic rate

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    DENSITY = <real>density_value
    #
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_RATE
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      SHEAR MODULUS = <real>shear_modulus
      BULK MODULUS = <real>bulk_modulus
      LAMBDA = <real>lambda
      YIELD STRESS = <real>yield_stress
      MODULUS TTTT = <real>modulus_tttt
      MODULUS TTLL = <real>modulus_ttll
      MODULUS TTWW = <real>modulus_ttww
      MODULUS LLLL = <real>modulus_llll
      MODULUS LLWW = <real>modulus_llww
      MODULUS WWWW = <real>modulus_wwww
      MODULUS TLTL = <real>modulus_tltl
      MODULUS LWLW = <real>modulus_lwlw
      MODULUS WTWT = <real>modulus_wtwt
      TX = <real>tx
      TY = <real>ty
      TZ = <real>tz
      LX = <real>lx
      LY = <real>ly
      LZ = <real>lz
      MODULUS FUNCTION = <string>modulus_function_name
      RATE FUNCTION = <string>rate_function_name
      T FUNCTION = <string>t_function_name
      L FUNCTION = <string>l_function_name
      W FUNCTION = <string>w_function_name
      TL FUNCTION = <string>tl_function_name
      LW FUNCTION = <string>lw_function_name
      WT FUNCTION = <string>wt_function_name
    END [PARAMETERS FOR MODEL ORTHOTROPIC_RATE]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Mie-Gruneisen

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL MIE_GRUNEISEN
      RHO_0 = <real>density
      C_0 = <real>sound_speed
      SHUG = <real>const_shock_velocity
      GAMMA_0 = <real>ambient_gruneisen_param
      POISSR = <real>poissons_ratio
      Y_0 = <real>yield_strength
      PMIN = <real>mean_stress(REAL_MAX)
    END [PARAMETERS FOR MODEL MIE_GRUNEISEN]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

  # Mie-Gruneisen power series

  BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
    # {thermal strain option}
    THERMAL STRAIN FUNCTION = <string>thermal_strain_function
    # or all three of the following
    THERMAL STRAIN X FUNCTION =
      <string>thermal_strain_x_function
    THERMAL STRAIN Y FUNCTION =
      <string>thermal_strain_y_function
    THERMAL STRAIN Z FUNCTION =
      <string>thermal_strain_z_function
    #
    BEGIN PARAMETERS FOR MODEL MIE_GRUNEISEN_POWER_SERIES
      RHO_0 = <real>density
      C_0 = <real>sound_speed
      K1 = <real>power_series_coeff1
      K2 = <real>power_series_coeff2
      K3 = <real>power_series_coeff3
      K4 = <real>power_series_coeff4
      K5 = <real>power_series_coeff5
      GAMMA_0 = <real>ambient_gruneisen_param
      POISSR = <real>poissons_ratio
      Y_0 = <real>yield strength
      PMIN = <real>mean_stress(REAL_MAX)
    END [PARAMETERS FOR MODEL MIE_GRUNEISEN_POWER_SERIES]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

```
# JWL (Jones-Wilkins-Lee)

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL JWL
    RHO_0 = <real>initial_density
    D = <real>detonation_velocity
    E_0 = <real>init_chem_energy
    A = <real>jwl_const_pressure1
    B = <real>jwl_const_pressure2
    R1 = <real>jwl_const_nondim1
    R2 = <real>jwl_const_nondim2
    OMEGA = <real>jwl_const_nondim3
    XDET = <real>x_detonation_point
    YDET = <real>y_detonation_point
    ZDET = <real>z_detonation_point
    TDET = <real>time_of_detonation
    B5 = <real>burn_width_const(2.5)
  END [PARAMETERS FOR MODEL JWL]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Ideal gas

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  # {thermal strain option}
  THERMAL STRAIN FUNCTION = <string>thermal_strain_function
  # or all three of the following
  THERMAL STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  #
  BEGIN PARAMETERS FOR MODEL IDEAL_GAS
    RHO_0 = <real>initial_density
    C_0 = <real>initial_sound_speed
    GAMMA = <real>ratio_specific_heats
  END [PARAMETERS FOR MODEL IDEAL_GAS]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Elastic laminate

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BEGIN PARAMETERS FOR MODEL ELASTIC_LAMINATE
    A11 = <real>a11_value
    A12 = <real>a12_value
    A16 = <real>a16_value
    A22 = <real>a22_value
    A26 = <real>a26_value
    A66 = <real>a66_value
    A44 = <real>a44_value
    A45 = <real>a45_value
    A55 = <real>a55_value
    B11 = <real>b11_value
    B12 = <real>b12_value
    B16 = <real>b16_value
    B22 = <real>b22_value
    B26 = <real>b26_value
    B66 = <real>b66_value
    D11 = <real>d11_value
    D12 = <real>d12_value
    D16 = <real>d16_value
    D22 = <real>d22_value
    D26 = <real>d26_value
    D66 = <real>d66_value
    COORDINATE SYSTEM = <string>coord_sys_name
    DIRECTION FOR ROTATION = 1|2|3
    ALPHA = <real>alpha_value_in_degrees
    THETA = <real>theta_value_in_degrees
    NTH11 FUNCTION = <string>nth11_function_name
    NTH22 FUNCTION = <string>nth22_function_name
    NTH12 FUNCTION = <string>nth12_function_name
    MTH11 FUNCTION = <string>mth11_function_name
    MTH22 FUNCTION = <string>mth22_function_name
    MTH12 FUNCTION = <string>mth12_function_name
  END [PARAMETERS FOR MODEL ELASTIC_LAMINATE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Define mesh
```

```
BEGIN FINITE ELEMENT MODEL <string>mesh_descriptor
  DATABASE NAME = <string>mesh_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  ALIAS <string>mesh_identifier AS <string>user_name
  BEGIN PARAMETERS FOR BLOCK <string list>block_names
    MATERIAL <string>material_name
    SOLID MECHANICS USE MODEL <string>model_name
    SECTION = <string>section_id
    LINEAR BULK VISCOSITY =
      <real>linear_bulk_viscosity_value(0.06)
    QUADRATIC BULK VISCOSITY =
      <real>quad_bulk_viscosity_value(1.20)
    HOURGLASS STIFFNESS =
      <real>hour_glass_stiff_value(solid = 0.05,
      shell/membrane = 0.0)
    HOURGLASS VISCOSITY =
      <real>hour_glass_visc_value(solid = 0.0,
      shell/membrane = 0.0)
    EFFECTIVE MODULI MODEL = <string>PRESTO|PRONTO|
      CURRENT|ELASTIC(PRONTO)
    ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD)
    ACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
      <string list>period_names
    RIGID BODY = <string>rb_name
  END [PARAMETERS FOR BLOCK <string list>block_names]
END [FINITE ELEMENT MODEL <string>mesh_descriptor]

# Element sections

BEGIN SOLID SECTION <string>solid_section_name
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC(MEAN QUADRATURE)
  DEVIATORIC PARAMETER = <real>deviatoric_param
  STRAIN INCREMENTATION = <string>MIDPOINT_INCREMENT|
    STRONGLY_OBJECTIVE|NODE_BASED(MIDPOINT_INCREMENT)
  NODE BASED ALPHA FACTOR =
    <real>bulk_stress_weight(0.01)
  NODE BASED BETA FACTOR =
    <real>shear stress_weight(0.35)
END [SOLID SECTION <string>solid_section_name]

BEGIN SHELL SECTION <string>shell_section_name
  THICKNESS = <real>shell_thickness
  THICKNESS MESH VARIABLE =
    <string>THICKNESS|<string>var_name
  THICKNESS TIME STEP = <real>time_value
  THICKNESS SCALE FACTOR = <real>thick_scale_factor(1.0)
  INTEGRATION RULE = TRAPEZOID|GAUSS|LOBATTO|SIMPSONS|
    USER(TRAPEZOID)
  NUMBER OF INTEGRATION POINTS =
    <integer>num_int_points(5)
  BEGIN USER INTEGRATION RULE
    <real>location_1 <real>weight_1
    <real>location_2 <real>weight_2
    .
    .
    <real>location_n <real>weight_n
  END [USER INTEGRATION RULE]
  LOFTING FACTOR = <real>lofting_factor(0.5)
  ORIENTATION = <string>orientation_name
END [SHELL SECTION <string>shell_section_name]

BEGIN MEMBRANE SECTION <string>membrane_section_name
  THICKNESS = <real>mem_thickness
  THICKNESS MESH VARIABLE =
    <string>THICKNESS|<string>var_name
  THICKNESS TIME STEP = <real>time_value
  THICKNESS SCALE FACTOR = <real>thick_scale_factor(1.0)
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC(MEAN QUADRATURE)
  DEVIATORIC PARAMETER = <real>deviatoric_param
  LOFTING FACTOR = <real>lofting_factor(0.5)
END [MEMBRANE SECTION <string>membrane_section_name]

BEGIN BEAM SECTION <string>beam_section_name
  SECTION = <string>ROD|TUBE|BAR|BOX|I
  WIDTH = <real>section_width
  HEIGHT = <real>section_width
  WALL THICKNESS = <real>wall_thickness
  FLANGE THICKNESS = <real>flange_thickness
  T AXIS = <real>tx <real>ty <real>tz (0 0 1)
  REFERENCE AXIS = <string>CENTER|RIGHT|
    TOP|LEFT|BOTTOM (CENTER)
  AXIS OFFSET = <real>s_offset <real>t_offset
END [BEAM SECTION <string>beam_section_name]

BEGIN TRUSS SECTION <string>truss_section_name
  AREA = <real>cross_sectional_area
  INITIAL LOAD = <real>initial_load
```

```
    PERIOD = <real>period
  END [TRUSS SECTION <string>truss_section_name]

  BEGIN DAMPER SECTION <string>damper_section_name
    AREA = <real>damper_cross_sectional_area
  END [DAMPER SECTION <string>damper_section_name]

  BEGIN POINT MASS SECTION <string>pointmass_section_name
    VOLUME = <real>volume
  END [POINT MASS SECTION <string>pointmass_section_name]

  BEGIN SPH SECTION <string>sph_section_name
    RADIUS MESH VARIABLE =
      <string>var_name|<string>attribute|SPHERE INITIAL
      RADIUS = <real>rad
    RADIUS MESH VARIABLE TIME STEP = <string>time
    PROBLEM DIMENSION = <integer>1|2|3(3)
    CONSTANT SPHERE RADIUS
  END [SPH SECTION <string>sph_section_name]

  # SPH utility commands

  SPH SYMMETRY PLANE <string>+X|+Y|+Z|-X|-Y|-Z
    <real>position_on_axis(0.0)
  SPH DECOUPLE STRAINS: <string>material1 <string>material2

  # Zoltan parameters

  BEGIN ZOLTAN PARAMETERS <string>parameter_name
    LOAD BALANCING METHOD  = <string>recursive coordinate
      bisection|recursive inertial bisection|hilbert space
      filling curve|octree
    DETERMINISTIC DECOMPOSITION = <string>false|true
    IMBALANCE TOLERANCE = <real>imb_tol
    OVER ALLOCATE MEMORY = <real>over_all_mem
    REUSE CUTS = <string>false|true
    ALGORITHM DEBUG LEVEL = <integer>alg_level
      #  0<=(alg_level)<=3
    CHECK GEOMETRY = <string>false|true
    KEEP CUTS = <string>false|true
    LOCK RCB DIRECTIONS = <string>false|true
    SET RCB DIRECTIONS = <string>do not order cuts|xyz|xzy|
      yzx|yxz|zxy|zyx
    RECTILINEAR RCB BLOCKS = <string>false|true
    RENUMBER PARTITIONS = <string>false|true
    OCTREE DIMENSION = <integer>oct_dimension
    OCTREE METHOD = <string>morton indexing|grey code|hilbert
    OCTREE MIN OBJECTS = <integer>min_obj  #  1<=(min_obj)
    OCTREE MAX OBJECTS = <integer>max_obj  #  1<=(max_obj)
    ZOLTAN DEBUG LEVEL = <integer>zoltan_level
      #  0<=(zoltan_level)<=10
    DEBUG PROCESSOR NUMBER = <integer>proc  #  1<=proc
    TIMER = <string>wall|cpu
    DEBUG MEMORY = <integer>dbg_mem  #  0<=(dbg_mem)<=3
  END [ZOLTAN PARAMETERS <string>parameter_name]

  # Output scheduler

  BEGIN OUTPUT SCHEDULER <string>scheduler_name
    START TIME = <real>output_start_time
    TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
    AT TIME <real>time_begin INCREMENT =
      <real>time_increment_dt
    ADDITIONAL TIMES = <real>output_time1
      <real>output_time2 ...
    AT STEP <integer>step_begin INCREMENT =
      <integer>step_increment
    ADDITIONAL STEPS = <integer>output_step1
      <integer>output_step2 ...
    TERMINATION TIME = <real>termination_time_value
   END [OUTPUT SCHEDULER <string>scheduler_name]

  # SIMOD Composite2d_via_1d model

  BEGIN SIMOD MODEL <string>some_name
    MODEL TYPE = Composite2d_via_1d
    MODEL PARAMETER: tangent_model = <string>name_tangent
    MODEL PARAMETER: normal_model = <string>name_normal
  END [SIMOD MODEL <string>some_name]

  # SIMOD Composite3d_via_1d model

  BEGIN SIMOD MODEL <string>some_name
    MODEL TYPE = Composite3d_via_1d
    MODEL PARAMETER: tangent1_model = <string>name_tangent1
    MODEL PARAMETER: tangent2_model = <string>name_tangent2
    MODEL PARAMETER: normal_model = <string>name_normal
  END [SIMOD MODEL <string>some_name]

  # SIMOD Concrete_Exp1d model
```

```
BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Concrete_Exp1d
  REAL PARAMETER: Ec = <real>real_val
  REAL PARAMETER: sigt = <real>real_val
  REAL PARAMETER: wo = <real>real_val
  REAL PARAMETER: Gf = <real>real_val
  REAL PARAMETER: c2 = <real>real_val
END [SIMOD MODEL <string>some_name]

# SIMOD Elastic_IM model

BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Elastic_IM
  REAL PARAMETER: k = <real>real_val
END [SIMOD MODEL <string>some_name]

# SIMOD Electrostatic_ParallelPL model

BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Electrostatic_ParallelPL
  REAL PARAMETER: zero_offset = <real>real_val
  REAL PARAMETER: clip_distance = <real>real_val
  REAL PARAMETER: scale_factor = <real>real_val
  REAL PARAMETER: permittivity = <real>real_val
  FUNCTION PARAMETER: voltage_function =
    <string>sierra_function
END [SIMOD MODEL <string>some_name]

# SIMOD Hamaker_ParallelPL model

BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = Hamaker_ParallelPL
  REAL PARAMETER: zero_offset = <real>real_val
  REAL PARAMETER: clip_distance = <real>real_val
  REAL PARAMETER: A = <real>real_val
  REAL PARAMETER: permittivity = <real>real_val
  FUNCTION PARAMETER: time_function =
    <string>sierra_function
END [SIMOD MODEL <string>some_name]

# SIMOD null_IM model

BEGIN SIMOD MODEL <string>some_name
  MODEL TYPE = null_IM
```

```
END [SIMOD MODEL <string>some_name]

# Begin Procedure scope

BEGIN PRESTO PROCEDURE <string>presto_procedure_name

  # Time block

  BEGIN TIME CONTROL
    BEGIN TIME STEPPING BLOCK <string>time_block_name
      START TIME = <real>start_time_value
      BEGIN PARAMETERS FOR PRESTO REGION
        <string>region_name
        INITIAL TIME STEP = <real>initial_time_step_value
        TIME STEP SCALE FACTOR =
          <real>time_step_scale_factor(1.0)
        TIME STEP INCREASE FACTOR =
          <real>time_step_increase_factor(1.1)
        STEP INTERVAL = <integer>nsteps(100)
      END [PARAMETERS FOR PRESTO REGION
        <string>region_name]
    END [TIME STEPPING BLOCK <string>time_block_name]
    TERMINATION TIME = <real>termination_time
  END TIME CONTROL

  # Begin Region scope

  BEGIN PRESTO REGION <string>presto_region_name

    USE FINITE ELEMENT MODEL <string>model_name

    # Error controller

    USE ERROR ESTIMATION CONTROLLER <string>err_name

    # Time step control using Lanczos

    BEGIN LANCZOS PARAMETERS <string>lanczos_name
      NUMBER EIGENVALUES = <integer>num_eig(20)
      STARTING VECTOR = <string>STRETCH_X|STRETCH_Y|STRETCH_Z
        (STRETCH_X)
      VECTOR SCALE = <real>vec_scale(1.0e-5)
      TIME SCALE = <real>time_scale(0.9)
      STEP INTERVAL = <integer>step_int(500)
      INCREMENT INTERVAL = <integer>incr_int(5)
      TIME STEP LIMIT = <real>step_lim(0.10)
    END [LANCZOS PARAMETERS <string>lanczos_name]

    # Time step control using nodes

    BEGIN NODE BASED TIME STEP PARAMETERS <string>nbased_name
      INCREMENT INTERVAL = <integer>incr_int(5)
      STEP INTERVAL = <integer>step_int(500)
      TIME STEP LIMIT = <real>step_lim(0.10)
    END [NODE BASED TIME STEP PARAMETERS <string>nbased_name]

    # Mass scaling

    BEGIN MASS SCALING
      # {node set commands}
      NODE SET = <string list>nodelist_names
      SURFACE = <string list>surface_names
      BLOCK = <string list>block_names
      INCLUDE ALL BLOCKS
      REMOVE NODE SET = <string list>nodelist_names
      REMOVE SURFACE = <string list>surface_names
      REMOVE BLOCK = <string list>block_names
      #
      TARGET TIME STEP = <real>target_time_step
      ALLOWABLE MASS INCREASE RATIO
        = <real>mass_increase_ratio
      #
      # additional command
      ACTIVE PERIODS = <string list>periods
    END MASS SCALING

    # Energy deposition

    BEGIN PRESCRIBED ENERGY DEPOSITION
      # {block set commands}
      BLOCK = <string_list>block_names
      INCLUDE ALL BLOCKS
      REMOVE BLOCK
      #
      # function commands
      T FUNCTION = <string>t_func_name
      X FUNCTION = <string>x_func_name
      Y FUNCTION = <string>y_func_name
      Z FUNCTION = <string>z_func_name
      #
      # input mesh command
      READ VARIABLE = <string>mesh_var_name
      #
      # user subroutine commands
      ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
      SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
      SUBROUTINE REAL PARAMETER: <string>param_name
        = <real>param_value
      SUBROUTINE INTEGER PARAMETER: <string>param_name
        = <integer>param_value
      SUBROUTINE STRING PARAMETER: <string>param_name
        = <string>param_value
    END [PRESCRIBED ENERGY DEPOSITION]

    # Torsional spring

    BEGIN TORSIONAL SPRING MECHANISM <string>spring_name
      NODE SETS = <string>nodelist_int1
        <string>nodelist_int2
        <string>nodelist_int3 <string>nodelist_int4
      TORSIONAL STIFFNESS = <real>stiffness
      INITIAL TORQUE = <real>init_load
      PERIOD = <real>time_period
      ACTIVE PERIODS = <string list>period_names
    END [TORSIONAL SPRING MECHANISM <string>spring_name]

    # Rigid body

    BEGIN RIGID BODY <string>rb_name
      POINT INERTIA = <real>Ixx <real>Iyy <real>Izz <real>Ixy
        <real>Iyz <real>Izx
      MAGNITUDE = <real>magnitude_of_velocity
      DIRECTION = <string>direction_definition
      ANGULAR VELOCITY = <real>omega
      CYLINDRICAL AXIS = <string>axis_definition
    END [RIGID BODY <string>rb_name]

    # Mass property calculations

    BEGIN MASS PROPERTIES
      # {block set commands}
      BLOCK = <string list>block_names
      INCLUDE ALL BLOCKS
      REMOVE BLOCK = <string list>block_names
      #
```

```
    # structure command
    STRUCTURE NAME = <string>structure_name
  END [MASS PROPERTIES]

  # Element death

  BEGIN ELEMENT DEATH <string>death_name
    # {block set commands}
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE BLOCK = <string list>block_names
    #
    # criteria commands
    CRITERION IS AVG|MAX|MIN NODAL VALUE OF
      <string>var_name[(<integer>component_num)]
      <|<=|=|>=|> <real>tolerance
    CRITERION IS ELEMENT VALUE OF
      <string>var_name[(<integer>component_num)]
      <|<=|=|>=|> <real>tolerance |
      <string>derived_quantity[(<integer>component_num)]
      <|<=|=|>=|> <real>tolerance
    CRITERION IS GLOBAL VALUE OF
      <string>var_name[(<integer>component_num)]
      <|<=|=|>=|> <real>tolerance
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    MATERIAL CRITERION
      = <string list>material_model_names
    #
    # evaluation commands
    CHECK STEP INTERVAL = <integer>num_steps
    CHECK TIME INTERVAL = <real>delta_t
    DEATH START TIME = <real>time
    #
    # miscellaneous option commands
    DEATH STEPS = <integer>death_steps(1)
    FORCE VALID ACME CONNECTIVITY
    #
    # additional command
    ACTIVE PERIODS = <string list>period_names
  END [ELEMENT DEATH <string>death_name]

  BEGIN DERIVED OUTPUT
    COMPUTE AND STORE STRESS VARIABLE =
      <string>derived_quantity_name
  END DERIVED OUTPUT
  #
  BEGIN DERIVED STRAIN OUTPUT
    COMPUTE AND STORE STRAIN VARIABLE =
      <string>derived_quantity_name
  END DERIVED OUTPUT
  #
  BEGIN DERIVED LOG STRAIN OUTPUT
    COMPUTE AND STORE LOG STRAIN VARIABLE =
      <string>derived_quantity_name
  END DERIVED OUTPUT

  # Mesh rebalance

  BEGIN REBALANCE
    INITIAL REBALANCE = ON|OFF(OFF)
    PERIODIC REBALANCE = ON|OFF|AUTO(OFF)
    STEP INTERVAL = <integer>step_interval
    COMMUNICATION RATIO THRESHOLD = <real>ratio
    ZOLTAN PARAMETERS = <string>parameter_name
  END REBALANCE

  # Initial condition

  BEGIN INITIAL CONDITION
    # {mesh-entity set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # variable identification commands
    INITIALIZE VARIABLE NAME = <string>var_name
    VARIABLE TYPE = [NODE|EDGE|FACE|ELEMENT|GLOBAL]
    #
    # constant magnitude command
```

```
    MAGNITUDE = <real list>initial_values
    #
    # input mesh commands
    READ VARIABLE = <string>mesh_var_name
    TIME = <real>time
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name |
      SURFACE SUBROUTINE = <string>subroutine_name |
      ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional command
    SCALE FACTOR = <real>scale_factor(1.0)
  END [INITIAL CONDITION]

  # Boundary conditions

  BEGIN FIXED DISPLACEMENT
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # component commands
    COMPONENT = <string>X/Y/Z | COMPONENTS =
      <string>X/Y/Z
    #
    # additional command
    ACTIVE PERIODS = <string list>period_names
  END [FIXED DISPLACEMENT]

  BEGIN PRESCRIBED DISPLACEMENT
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # function commands
    DIRECTION = <string>defined_direction |
      COMPONENT = <string>X|Y|Z |
      CYLINDRICAL AXIS = <string>defined_axis |
      RADIAL AXIS = <string>defined_axis
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED DISPLACEMENT]

  BEGIN PRESCRIBED VELOCITY
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # function commands
    DIRECTION = <string>defined_direction |
      COMPONENT = <string>X|Y|Z |
      CYLINDRICAL AXIS = <string>defined_axis |
      RADIAL AXIS = <string>defined_axis
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED VELOCITY]

  BEGIN PRESCRIBED ACCELERATION
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # function commands
    DIRECTION = <string>defined_direction |
      COMPONENT = <string>X|Y|Z
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED ACCELERATION]
```

```
BEGIN FIXED ROTATION
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # component commands
  COMPONENT = <string>X/Y/Z | COMPONENTS =
    <string>X/Y/Z
  #
  # additional command
  ACTIVE PERIODS = <string list>periods_names
END [FIXED ROTATION]

BEGIN PRESCRIBED ROTATION
  # {node set commands}
  NODE SET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # function commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
```

```
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED ROTATION]

  BEGIN INITIAL VELOCITY
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # direction commands
    COMPONENT = <string>X|Y|Z |
      DIRECTION = <string>defined_direction
    MAGNITUDE = <real>magnitude_of_velocity
    #
    # angular velocity commands
    CYLINDRICAL AXIS = <string>defined_axis
    ANGULAR VELOCITY = <real>angular_velocity
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
  END [INITIAL VELOCITY]

  BEGIN PRESSURE
    # {surface set commands}
    SURFACE = <string list>surface_names
    REMOVE SURFACE = <string list>surface_names
    #
    # function command
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    SURFACE SUBROUTINE = <string>subroutine_name |
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # external pressure sources
    READ VARIABLE = <string>variable_name
    OBJECT TYPE = <string>NODE|FACE(NODE)
    TIME = <real>time
    FIELD VARIABLE = <string>field_variable
    #
    # output external forces from pressure
    EXTERNAL FORCE CONTRIBUTION OUTPUT NAME
      = <string>variable_name
    #
    # additional commands
    USE DEATH = <string>death_name
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESSURE]

  BEGIN TRACTION
    # {surface set commands}
    SURFACE = <string list>surface_names
    REMOVE SURFACE = <string list>surface_names
    #
    # function commands
    DIRECTION = <string>direction_name
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # SIMOD commands
    USE SIMOD MODEL = <string>Simod_model_name
    REFERENCE PLANE AXIS = <string>axis_direction
    REFERENCE PLANE T1 DIRECTION = <string>t1_direction
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [TRACTION]

  BEGIN PRESCRIBED FORCE
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # function commands
    DIRECTION = <string>defined_direction |
      COMPONENT = <string>X|Y|Z
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED FORCE]

  BEGIN PRESCRIBED MOMENT
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # function commands
    DIRECTION = <string>defined_direction |
      COMPONENT = <string>X|Y|Z
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED MOMENT]

  # Standard temperature boundary condition

  BEGIN PRESCRIBED TEMPERATURE
    # {block set commands}
    BLOCK = <string_list>block_names
    INCLUDE ALL BLOCKS
    REMOVE BLOCK
    #
    # function command
    FUNCTION = <string>function_name
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # read variable commands
    READ VARIABLE = <string>variable_name
    TIME = <real>time
    #
    # additional commands
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [PRESCRIBED TEMPERATURE]

  # Specialized boundary conditions

  BEGIN GRAVITY
    # {node set commands}
    NODE SET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    #
    DIRECTION = <string>defined_direction
    FUNCTION = <string>function_name
    GRAVITATIONAL CONSTANT = <real>g_constant
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
  END [GRAVITY]

  BEGIN CAVITY EXPANSION
    EXPANSION RADIUS = <string>SPHERICAL|CYLINDRICAL
      (spherical)
    SURFACE = <string list>surface_ids
    REMOVE SURFACE = <string list>surface_ids
    FREE SURFACE = <real>top_surface_zcoord
      <real>bottom_surface_zcoord
    NODE SETS TO DEFINE BODY AXIS =
      <string>nodelist_1 <string>nodelist_id2
    TIP RADIUS = <real>tip_radius
    BEGIN LAYER <string>layer_name
      LAYER SURFACE = <real>top_layer_zcoord
        <real>bottom_layer_zcoord
      PRESSURE COEFFICIENTS = <real>c0 <real>c1
        <real>c2
      SURFACE EFFECT = <string>NONE|SIMPLE_ON_OFF(NONE)
      FREE SURFACE EFFECT COEFFICIENTS = <real>coeff1
        <real>coeff2
    END [LAYER <string>layer_name]
    ACTIVE PERIODS = <string list>period_names
  END [CAVITY EXPANSION]

  BEGIN SILENT BOUNDARY
    SURFACE = <string list>surface_names
    REMOVE SURFACE = <string list>surface_names
    ACTIVE PERIODS = <string list>period_names
  END [SILENT BOUNDARY]

  BEGIN SPOT WELD
    NODE SET = <string list>nodelist_ids
    REMOVE NODE SET = <string list>nodelist_ids
    SURFACE = <string list>surface_ids
    REMOVE SURFACE = <string list>surfac_ids
    SECOND SURFACE = <string>surface_id
    NORMAL DISPLACEMENT FUNCTION =
      <string>function_nor_disp
    NORMAL DISPLACEMENT SCALE FACTOR =
      <real>scale_nor_disp[1.0]
    TANGENTIAL DISPLACEMENT FUNCTION =
      <string>function_tang_disp
    TANGENTIAL DISPLACEMENT SCALE FACTOR =
      <real>scale_tang_disp[1.0]
    FAILURE ENVELOPE EXPONENT = <real>exponent
    FAILURE FUNCTION = <string>fail_func_name
    FAILURE DECAY CYCLES = <integer>number_decay_cycles
    SEARCH TOLERANCE = <real>search_tolerance
    ACTIVE PERIODS = <string list>period_names
  END [SPOT WELD]

  BEGIN LINE WELD
    SURFACE = <string list> surface_names
    REMOVE SURFACE = <string list> surface_names
    BLOCK = <string list> block_names
    REMOVE BLOCK = <string list>block_names
    SEARCH TOLERANCE = <real>search_tolerance
    R DISPLACEMENT FUNCTION =
      <string>r_disp_fucntion_name
    R DISPLACEMENT SCALE FACTOR = <real>r_disp_scale
    S DISPLACEMENT FUNCTION =
      <string>s_disp_function_name
    S DISPLACEMENT SCALE FACTOR = <real>s_disp_scale
    T DISPLACEMENT FUNCTION =
      <string>t_disp_function_name
    T DISPLACEMENT SCALE FACTOR = <real>t_disp_scale
    R ROTATION FUNCTION =
      <string>r_rotation_function_name
    R ROTATION SCALE FACTOR = <real>r_rotation_scale
    S ROTATION FUNCTION =
      <string>s_rotation_function_name
    S ROTATION SCALE FACTOR = <real>s_rotation_scale
    T ROTATION FUNCTION =
      <string>t_rotation_function_name
    T ROTATION SCALE FACTOR = <real>t_rotation_scale
    FAILURE ENVELOPE EXPONENT = <real>k
    FAILURE DECAY CYCLES = <integer>number_decay_cycles
    ACTIVE PERIODS = <string list>period_names
  END [LINE WELD]

  BEGIN VISCOUS DAMPING <string>damp_name
    # {block set commands}
    BLOCK = <string list>block_names
    INCLUDE ALL BLOCKS
    REMOVE BLOCK = <string list>block_names
    #
    MASS DAMPING COEFFICIENT = <real>mass_damping
    STIFFNESS DAMPING COEFFICIENT = <real>stiff_damping
    #
    # additional command
    ACTIVE PERIODS = <string list>period names
  END [VISCOUS DAMPING <string>damp_name]

  # Contact

  BEGIN CONTACT DEFINITION <string>name
    #
    # contact surface and node set definition
    CONTACT SURFACE <string>name
    CONTAINS <string list>surface_names
    #
    SKIN ALL BLOCKS = <string>ON|OFF(OFF)
      [EXCEPT <string list> block_names]
    #
    BEGIN CONTACT SURFACE <string>name
      BLOCK = <string list>block_names
      SURFACE = <string list>surface_names
      NODE SET = <string list>node_set_names
      REMOVE BLOCK = <string list>block_names
      REMOVE SURFACE = <string list>surface_names
      REMOVE NODE SET = <string list>nodelist_names
    END [CONTACT SURFACE <string>name]
    #
    CONTACT NODE SET <string>surface_name
      CONTAINS <string>nodelist_names
    #
    # analytic surfaces
    BEGIN ANALYTIC PLANE <string>name
      NORMAL = <string>defined_direction
      POINT = <string>defined_point
    END [ANALYTIC PLANE <string>name]
    #
    BEGIN ANALYTIC CYLINDER <string>name
      CENTER = <string>defined_point
      AXIAL DIRECTION = <string>defined_axis
      RADIUS = <real>cylinder_radius
      LENGTH = <real>cylinder_length
      CONTACT NORMAL = <string>OUTSIDE|INSIDE
    END [ANALYTIC CYLINDER <string>name]
    #
    BEGIN ANALYTIC SPHERE <string>name
      CENTER = <string>defined_point
      RADIUS = <real>sphere_radius
    END [ANALYTIC SPHERE <string>name]
    # end contact surface and node set definition
    #
    UPDATE ALL SURFACES FOR ELEMENT DEATH = <string>ON|OFF(ON)
    #
    BEGIN REMOVE INITIAL OVERLAP
      OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
      OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
      SHELL OVERLAP ITERATIONS = <integer>max_iter(10)
      SHELL OVERLAP TOLERANCE = <real>shell_over_tol(0.0)
    END [REMOVE INITIAL OVERLAP]
    #
    MULTIPLE INTERACTIONS = <string>ON|OFF(ON)
    MULTIPLE INTERACTIONS WITH ANGLE = <real>angle(60.0)
    #
    SURFACE NORMAL SMOOTHING = <string>ON|OFF(OFF)
    #
    ERODED FACE TREATMENT = <string>NONE|ALL(ALL)
    #
    # shell lofting
    BEGIN SHELL LOFTING
      LOFTING ALGORITHM = <string>ON|OFF(ON)
      COINCIDENT SHELL TREATMENT = <string>DISALLOW|IGNORE|
        SIMPLE(DISALLOW)
      COINCIDENT SHELL HEX TREATMENT = <string>DISALLOW|
        IGNORE|TAPERED|EMBEDDED(DISALLOW)
    END [SHELL LOFTING]
    #
    # contact output
    CONTACT VARIABLES = <string>ON|OFF(OFF)
    #
    # surface-physics models
    BEGIN FRICTIONLESS MODEL <string>name
    END [FRICTIONLESS MODEL <string>name]
    #
    BEGIN CONSTANT FRICTION MODEL <string>name
      FRICTION COEFFICIENT = <real>coeff
    END [CONSTANT FRICTION MODEL <string>name]
    #
    BEGIN TIED MODEL <string>name
    END [TIED MODEL <string>name]
    #
    BEGIN SPRING WELD MODEL <string>name
      NORMAL DISPLACEMENT FUNCTION = <string>func_name
      NORMAL DISPLACEMENT SCALE FACTOR =
        <real>scale_factor(1.0)
      TANGENTIAL DISPLACEMENT FUNCTION = <string>func_name
      TANGENTIAL DISPLACEMENT SCALE FACTOR =
        <real>scale_factor(1.0)
      FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
      FAILURE DECAY CYCLES = <integer>num_cycles(1)
      FAILED MODEL = <string>failed_model_name|FRICTIONLESS
        (FRICTIONLESS)
    END [SPRING WELD MODEL <string>name]
    #
    BEGIN SURFACE WELD MODEL <string>name
      NORMAL CAPACITY = <real>normal_cap
      TANGENTIAL CAPACITY = <real>tangential_cap
      FAILURE DECAY CYCLES = <integer>num_cycles(1)
      FAILED MODEL = <string>failed_model_name|FRICTIONLESS
        (FRICTIONLESS)
    END [SURFACE WELD MODEL <string>name]
    #
    BEGIN AREA WELD MODEL <string>name
      NORMAL CAPACITY = <real>normal_cap
      TANGENTIAL CAPACITY = <real>tangential_cap
      FAILURE DECAY CYCLES = <integer>num_cycles(1)
      FAILED MODEL = <string>failed_model_name|FRICTIONLESS
        (FRICTIONLESS)
    END [AREA WELD MODEL <string>name]
    #
    BEGIN ADHESION MODEL <string>name
      ADHESION FUNCTION = <string>func_name
      ADHESION SCALE FACTOR = <real>scale_factor(1.0)
    END [ADHESION MODEL <string>name]
    #
    BEGIN COHESIVE ZONE MODEL <string>name
      TRACTION DISPLACEMENT FUNCTION = <string>func_name
      TRACTION DISPLACEMENT SCALE FACTOR =
        <real>scale_factor(1.0)
      CRITICAL NORMAL GAP = <real>crit_norm_gap
      CRITICAL TANGENTIAL GAP = <real>crit_tangential_gap
    END [COHESIVE ZONE MODEL <string>name]
    #
    BEGIN JUNCTION MODEL <string>name
      NORMAL TRACTION FUNCTION = <string>func_name
      NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
      TANGENTIAL TRACTION FUNCTION = <string>func_name
      TANGENTIAL TRACTION SCALE FACTOR =
        <real>scale_factor(1.0)
      NORMAL CUTOFF DISTANCE FOR TANGENTIAL TRACTION =
        <real>distance
    END [JUNCTION MODEL <string>name]
    #
    BEGIN THREADED MODEL <string>name
      NORMAL TRACTION FUNCTION = <string>func_name
      NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
      TANGENTIAL TRACTION FUNCTION = <string>func_name
      TANGENTIAL TRACTION SCALE FACTOR =
        <real>scale_factor(1.0)
      TANGENTIAL TRACTION GAP FUNCTION = <string>func_name
      TANGENTIAL TRACTION GAP SCALE FACTOR =
        <real>scale_factor(1.0)
      NORMAL CAPACITY = <real>normal_cap
      TANGENTIAL CAPACITY = <real>tangential_cap
      FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
      FAILURE DECAY CYCLES = <integer>num_cycles(1)
      FAILED MODEL = <string>failed_model_name|FRICTIONLESS
        (FRICTIONLESS)
    END [THREADED MODEL <string>name]
    #
    BEGIN PV_DEPENDENT MODEL <string>name
```

```
    STATIC COEFFICIENT = <real>stat_coeff
    DYNAMIC COEFFICIENT = <real>dyn_coeff
    VELOCITY DECAY = <real>vel_decay
    REFERENCE PRESSURE = <real>p_ref
    OFFSET PRESSURE = <real>p_off
    PRESSURE EXPONENT = <real>p_exp
  END [PV_DEPENDENT MODEL <string>name]
  #
  BEGIN SIMOD SHARED MODEL <string>name
    USE SIMOD MODEL = <string>name
    FAILED MODEL = <string>name|FRICTIONLESS
      (FRICTIONLESS)
  END [SIMOD SHARED MODEL <string>name]
  #
  BEGIN SIMOD UNKNOWN MODEL <string>name
    USE SIMOD MODEL = <string>name
    FAILED MODEL = <string>name|FRICTIONLESS
      (FRICTIONLESS)
  END [SIMOD UNKNOWN MODEL <string>name]
  # end surface physics models
  #
  BEGIN USER SUBROUTINE MODEL <string>name
    INITIALIZE MODEL SUBROUTINE = <string>init_model_name
    INITIALIZE TIME STEP SUBROUTINE = <string>init_ts_name
    INITIALIZE NODE STATE DATA SUBROUTINE =
      <string>init_node_data_name
    LIMIT FORCE SUBROUTINE = <string>limit_force_name
    ACTIVE SUBROUTINE = <string>active_name
    INTERACTION TYPE SUBROUTINE = <string>interaction_name
  END [USER SUBROUTINE MODEL <string>name]
  #
  # search options command block
  BEGIN SEARCH OPTIONS [<string>name]
    GLOBAL SEARCH INCREMENT = <integer>num_steps(1)
    GLOBAL SEARCH ONCE = <string>ON|OFF(OFF)
    SEARCH TOLERANCE = <string>AUTOMATIC|USER_DEFINED
      (AUTOMATIC)
    NORMAL TOLERANCE = <real>norm_tol
    TANGENTIAL TOLERANCE = <real>tang_tol
    SECONDARY DECOMPOSITION = <string>ON|OFF(ON)
  END [SEARCH OPTIONS <string>name]
  #
  # enforcement
  BEGIN ENFORCEMENT OPTIONS [<string>name]
    ENFORCEMENT ALGORITHM = <string>MOMENTUM_BALANCE|
      PENALTY(MOMENTUM_BALANCE)
    MOMENTUM BALANCE ITERATIONS = <integer>num_iter(5)
  END [ENFORCEMENT OPTIONS <string>name]
  #
  BEGIN INTERACTION DEFAULTS [<string>name]
    SURFACES = <string list>surface_names
    SELF CONTACT = <string>ON|OFF(OFF)
    GENERAL CONTACT = <string>ON|OFF(OFF)
    AUTOMATIC KINEMATIC PARTITION = <string>ON|OFF(OFF)
    INTERACTION BEHAVIOR = <string>SLIDING|
      INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
    FRICTION MODEL = <string>friction_model_name|
      FRICTIONLESS(FRICTIONLESS)
  END [INTERACTION DEFAULTS <string>name]
  #
  BEGIN INTERACTION [<string>name]
    SURFACES = <string>surface1 <string>surface2
    MASTER = <string>surface
    SLAVE = <string>surface
    KINEMATIC PARTITION = <real>kin_part
    NORMAL TOLERANCE = <real>norm_tol
    TANGENTIAL TOLERANCE = <real>tang_tol
    OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
    OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
    FRICTION MODEL = <string>friction_model_name|
      FRICTIONLESS(FRICTIONLESS)
    AUTOMATIC KINEMATIC PARTITION
    INTERACTION BEHAVIOR = <string>SLIDING|
      INFINITESIMAL_SLIDING|NO_INTERACTION(SLIDING)
  END [INTERACTION <string>name]
  # end enforcement
  #
END [CONTACT DEFINITION <string>name]

# Results specification

BEGIN RESULTS OUTPUT <string>results_name
  DATABASE NAME = <string>results_file_name
  DATABASE TYPE =
    <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  TITLE <string>user_title
  NODE VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODAL VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]
  NODESET VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODESET VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>nodelist_names
        ... <string>variable_name
        [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
        <string list>nodelist_names
  FACE VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | FACE VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>surface_names
        ...  <string>variable_name
        [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
        <string list>surface_names
  ELEMENT VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | ELEMENT VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>block_names
        ... <string>variable_name
        [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
        <string list>block_names
  COMPONENT SEPARATOR CHARACTER = <string>character|NONE
  GLOBAL VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
  START TIME = <real>output_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
  USE OUTPUT SCHEDULER <string>scheduler name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
END [RESULTS OUTPUT <string>results_name]

# User output

BEGIN USER OUTPUT
  # {mesh-entity set commands}
  NODE SET = <string_list>nodeset_names
  SURFACE = <string_list>surface_names
  BLOCK = <string_list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list> surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # compute global result command
  COMPUTE GLOBAL <string>results_var_name AS
    <string>SUM|AVERAGE|MAX|MIN OF <string>NODAL|
    ELEMENT <string>value_var_name
    [(<integer>component_num)]
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name |
    SURFACE SUBROUTINE = <string>subroutine_name |
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # copy command
  COPY ELEMENT VARIABLE <string>ev_name TO NODAL
```

```
      VARIABLE <string>nv_name
    #
    # compute for element death
    COMPUTE AT EVERY TIME STEP
    #
    # additional command
    ACTIVE PERIODS = <string list>period_names
  END [USER OUTPUT]

  # Time step initialization

  BEGIN TIME STEP INITIALIZATION
    # {mesh-entity set commands}
    NODE SET = <string_list>nodeset_names
    SURFACE = <string_list>surface_names
    BLOCK = <string_list>block_names
    INCLUDE ALL BLOCKS
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list> surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name |
      SURFACE SUBROUTINE = <string>sub_name |
      ELEMENT BLOCK SUBROUTINE = <string>sub_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional command
    ACTIVE PERIODS = <string list>period_names
  END TIME STEP INITIALIZATION

  # User variable

  BEGIN USER VARIABLE <string>var_name
    TYPE = <string>NODE|ELEMENT|GLOBAL
      [<string>REAL|INTEGER LENGTH = <integer>length]|
      [<string>SYM_TENSOR|FULL_TENSOR|VECTOR]
    GLOBAL OPERATOR = <string>SUM|MIN|MAX]
    INITIAL VALUE = <real list>values
    USE WITH RESTART
  END [USER VARIABLE <string>var_name]

  # History specification

  BEGIN HISTORY OUTPUT <string>history_name
    DATABASE NAME = <string>history_file_name
    DATABASE TYPE =
      <string>database_type(exodusII)
    OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
     (ON|TRUE|YES)
    TITLE <string>user_title
    #
    # for global variables
    VARIABLE = GLOBAL
      <string>variable_name
      [AS <string>history_variable_name]
    #
    # for mesh entity - node, edge, face,
    # element - variables
    VARIABLE =
      NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
      AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
      [AS <string>history_variable_name]
    #
    # for nearest point output of mesh entity - node,
    # edge, face, element - variables
    VARIABLE =
      NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
      NEAREST LOCATION <real>global_x,
        <real>global_y>, <real>global_z
      [AS <string>history_variable_name]
    START TIME = <real>output_start_time
    TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
    AT TIME <real>time_begin INCREMENT =
      <real>time_increment_dt
    ADDITIONAL TIMES = <real>output_time1
      <real>output_time2 ...
    AT STEP <integer>step_begin INCREMENT =
      <integer>step_increment
    ADDITIONAL STEPS = <integer>output_step1
      <integer>output_step2 ...
    TERMINATION TIME = <real>termination_time_value
    USE OUTPUT SCHEDULER <string>scheduler_name
    OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|
          SIGHUP|SIGINT|SIGPIPE|SIGQUIT|SIGTERM|
          SIGUSR1|SIGUSR2|SIGABRT|
          SIGKILL|SIGILL|SIGSEGV
      END [HISTORY OUTPUT <string>history_name]

      # Restart specification

      BEGIN RESTART DATA <string>restart_name
        DATABASE NAME = <string>restart_file_name
        INPUT DATABASE NAME = <string>restart_input_file
        OUTPUT DATABASE NAME =
          <string>restart_output_file
        DATABASE TYPE =
          <string>database_type(exodusII)
        OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
          (ON|TRUE|YES)
        START TIME = <real>restart_start_time
        TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
        AT TIME <real>time_begin INCREMENT =
          <real>time_increment_dt
        ADDITIONAL TIMES = <real>output_time1
          <real>output_time2 ...
        AT STEP <integer>step_begin INCREMENT =
          <integer>step_increment
        ADDITIONAL STEPS = <integer>output_step1
          <integer>output_step2 ...
        TERMINATION TIME = <real>termination_time_value
        OVERLAY COUNT = <integer>overlay_count
        CYCLE COUNT = <integer>cycle_count
        USE OUTPUT SCHEDULER <string>scheduler_name
        OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
          SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
          SIGKILL|SIGILL|SIGSEGV
      END [RESTART DATA <string>restart_name]

    END [PRESTO REGION <string>presto_region_name]

  END [PRESTO PROCEDURE <string>presto_procedure_name]

END [SIERRA <string>name]
```

Intentionally Left Blank

Index

Distribution

2	0325	Duong, Henry, 2627
1	0372	Gwinn, Kenneth W., 1524
1	0372	Hammerand, Daniel C., 1524
1	0380	Koteras, James R., 1542
1	0380	Reinert, Rhonda K., 1542
1	0847	Fulcher, Clay W., 1526
1	0886	Cox, James V., 1526
1	1185	Bessette, Gregory C.
1	KCP/1D40	Neilsen, Michael K., 1526
2	9018	Central Technical Files, 8944
2	0899	Technical Library, 4536